BERN-NN: Tight Bound Propagation For Neural Networks Using Bernstein Polynomial Interval Arithmetic

Wael Fatnassi*
University of California, Irvine
Dept. of Electrical Engineering and Computer Science
wfatnass@uci.edu

Haitham Khedr*
University of California, Irvine
Dept. of Electrical Engineering and Computer Science
hkhedr@uci.edu

Valen Yamamoto
University of California, Irvine
Dept. of Electrical Engineering and Computer Science
vyamamot@uci.edu

Yasser Shoukry
University of California, Irvine
Dept. of Electrical Engineering and Computer Science
yshoukry@uci.edu

ABSTRACT

In this paper, we present BERN-NN as an efficient tool to perform bound propagation of Neural Networks (NNs). Bound propagation is a critical step in wide range of NN model checkers and reachability analysis tools. Given a bounded input set, bound propagation algorithms aim to compute tight bounds on the output of the NN. So far, linear and convex optimizations have been used to perform bound propagation. Since neural networks are highly non-convex, state-of-the-art bound propagation techniques suffer from introducing large errors. To circumvent such drawback, BERN-NN approximates the bounds of each neuron using a class of polynomials called Bernstein polynomials. Bernstein polynomials enjoy several interesting properties that allow BERN-NN to obtain tighter bounds compared to those relying on linear and convex approximations. BERN-NN is efficiently parallelized on graphic processing units (GPUs). Extensive numerical results show that bounds obtained by BERN-NN are orders of magnitude tighter than those obtained by state-of-the-art verifiers such as linear programming and linear interval arithmetic. Moreoveer, BERN-NN is both faster and produces tighter outputs compared to convex programming approaches like alpha-CROWN.

KEYWORDS

Neural Networks, Bernstein Polynomials, Abstraction Refinement

ACM Reference Format:
Wael Fatnassi, Haitham Khedr, Valen Yamamoto, and Yasser Shoukry. 2023. BERN-NN: Tight Bound Propagation For Neural Networks Using Bernstein Polynomial Interval Arithmetic. In *Proceedings of the 26th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '23), May 09–12, 2023, San Antonio, TX, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3575870.3587126

*Both authors contributed equally to the paper

This work is licensed under a Creative Commons Attribution International 4.0 License.

HSCC '23, May 09–12, 2023, San Antonio, TX, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0033-0/23/05.
https://doi.org/10.1145/3575870.3587126

1 INTRODUCTION

Neural Networks (NNs) have become an increasingly central component of modern, safety-critical, cyber-physical systems like autonomous driving, autonomous decision-making in smart cities, and even autonomous landing in avionic applications. Thus, there is an increasing need to verify the safety and correctness [16, 31, 32] of NNs when they are used to control physical systems.

The problem of NN Verification has been well studied in literature [25]. Most NN verifiers rely mainly on either using linear relaxation and optimization [10, 20, 24, 35, 37, 38] to falsify a given property or prove its satisfaction, or reachability analysis to compute an over-approximation of the output set. The latter is specifically important for control applications where the property of interest is defined over a time horizon. Both techniques rely on overapproximation, hence, having tight output bounds is at the core of NN verification as it allows reasoning about NN properties in an efficient manner. For example, model checking the robustness of NNs against adversarial perturbations can be done by simply comparing the tight bounds of the outputs of the network. Moreover, networks used in control applications often involve multi-step reachability, and hence computing tight bounds is crucial to harness the accumulation of the error and hence be able to efficiently reason about the safety of the system.

Due to the non-convexity and non-linearity of NNs, the problem of finding the exact bounds of NN outputs is NP-hard[23]. Different tools have been proposed to find tight overapproximations of NN outputs. MILP-based methods [1, 3–6, 8, 15, 33] encode the non-linear activations as linear and integer constraints. Reachability methods [14, 19, 22, 34, 36, 39, 40] use layer-by-layer reachability analysis (exact or overapproximation) of the network. Most of these methods either rely on convex *linear relaxation* of the non-linear activation functions to overapproximate the output of the NN, or try to find the exact bounds which are often intractable.

In this work, we explore using polynomials to approximate non-linear activations (e.g. ReLU). More specifically, we approximate non-linear activations using Bernstein polynomials which are constructed as a linear combination of the Bernstein basis polynomials [12]. The use of Bernstein polynomials is motivated by two reasons. First, based on the Stone-Weierstrass approximation theorem [7], Bernstein polynomials can uniformly approximate continuous activation functions. Second and most importantly, bounding

a Bernstein polynomial is computationally cheap based on the interesting properties of Bernstein polynomials discussed in section 3. The goal of using higher-order polynomials versus linear relaxation is to get tight bounds on NNs which is crucial for verifying a large class of formal properties. This idea of using polynomials has inspired other researchers [9, 11, 21], however, the proposed tools suffer from scalability issues.

Our main contributions can be summarized as follows:

- We propose a tool that uses Bernstein polynomials to approximate ReLU activations and hence compute tighter NN bounds than state-of-the-art.
- The tool is designed with scalability in mind; hence, the entire operations can be accelerated using GPUs.
- We show that by using the proposed approximation, we are able to compute tighter output sets than alpha-Crown (winner of VNN22' competition[2] for Formal Verification of NNs) and other state-of-the-art bounding methods. For instance, BERN-NN approximations are twice reduced compared to alpha-Crown for actual NN's controllers. Moreover, Numerical results showed that Bern-NN can process neural networks with more than 1000 neurons in less than 2 minutes

2 PROBLEM FORMULATION

2.1 Notation:

General notation: We use the symbols $\mathbb{N}$ and $\mathbb{R}$ to denote the set of natural and real numbers, respectively. We denote by $x = (x_1, x_2, \cdots, x_n) \in \mathbb{R}^n$ 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 $\underline{d} = (\underline{d}_1, \cdots, \underline{d}_n)$ and $\overline{d} = (\overline{d}_1, \cdots, \overline{d}_n)$ are the lower and upper bounds of the hyperrectangle, respectively. We denote by x^T and A^T the transpose operation of the vector x and the matrix A. We denote by 0_n a vector that contains n zero values and by $0_{n\times m}$ the matrix of shape $n \times m$ that contains zeros. Finally, $A * B$ stands for the element-wise product between the multi-dimensional tensors A and B, and $A \otimes B$ stands for the Kronecker product between the matrices A and B.

Notation pertaining to multivariate polynomials: For a real-valued vector $x = (x_1, x_2, \cdots, x_n) \in \mathbb{R}^n$ and an index-vector $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$, 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 $K = (k_1, \cdots, k_n) \in \mathbb{N}^n$ and $L = (l_1, \cdots, l_n) \in \mathbb{N}^n$, we use the following notation throughout this paper:

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

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 $i = 1, \ldots, n$.

Notation pertaining to neural networks: In this paper, we consider H-layer, feed-forward, ReLU-based neural networks $\mathcal{NN} : \mathbb{R}^n \to \mathbb{R}^o$ defined as:

$$\mathcal{NN}(x) = W^{(H)} z^{(H-1)} + b^{(H)}$$
$$z^{(H-1)} = \sigma\left(W^{(H-1)} z^{(H-2)} + b^{(H-1)}\right)$$
$$\vdots$$
$$z^{(1)} = \sigma\left(W^{(1)} x + b^{(1)}\right)$$

where σ is the ReLU activation function (i.e., $\sigma(z) = \max(0, z)$) that operates element-wise, $W^{(i)} \in \mathbb{R}^{h_i \times h_{i-1}}$ and $b^{(i)} \in \mathbb{R}^{h_i}$ with $i \in \{1, \cdots, H\}$ are the weights and the biases of the network. For simplicity of notation, we use $\hat{z}_j^{(i)}$ and $z_j^{(i)}$ to denote the pre-activation (input) and the post-activation (output) of the j-th neuron in the i-th layer.

2.2 Main Problem:

In this paper, we seek to find polynomials that upper and lower approximate the NN's outputs $\mathcal{NN}(x)$ whenever the NN's input x is confined within a pre-defined hypercube, i.e. $x \in I_n(\underline{d}, \overline{d})$.

Problem 1. *Given a neural network $\mathcal{NN} : \mathbb{R}^n \to \mathbb{R}^o$ and an input domain hypercube $I_n(\underline{d}, \overline{d}) \subset \mathbb{R}^n$. Find lower and upper approximate polynomials $\left(\underline{p}_{\mathcal{NN},1}(x), \overline{p}_{\mathcal{NN},1}(x)\right), \ldots \left(\underline{p}_{\mathcal{NN},o}(x), \overline{p}_{\mathcal{NN},o}(x)\right)$, such that:*

$$\underline{p}_{\mathcal{NN},1}(x) \le \mathcal{NN}_1(x) \le \overline{p}_{\mathcal{NN},1}(x)$$
$$\vdots$$
$$\underline{p}_{\mathcal{NN},o}(x) \le \mathcal{NN}_o(x) \le \overline{p}_{\mathcal{NN},o}(x),$$

where with some abuse of notation, we use $\mathcal{NN}_i(x)$ to denote the ith output of the neural network $\mathcal{NN}$.

Note that the lower/upper bound polynomials $\left(\underline{p}_{\mathcal{NN},1}(x), \overline{p}_{\mathcal{NN},1}(x)\right), \ldots \left(\underline{p}_{\mathcal{NN},o}(x), \overline{p}_{\mathcal{NN},o}(x)\right)$ depend on the input domain I_n. That is, for each value of I_n, we need to find different lower/upper bound polynomials. However, for the sake of simplicity of notation, we drop the dependency on I_n.

3 TIGHT BOUNDS OF RELU FUNCTIONS USING BERNSTEIN POLYNOMIALS

To solve Problem 1, we rely on a class of polynomials called Bernstein polynomials which are defined as follows:

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

$$B_{g,L}(x) = \sum_{K \le L} b^g_{K,L} Ber_{K,L}(x), \tag{1}$$

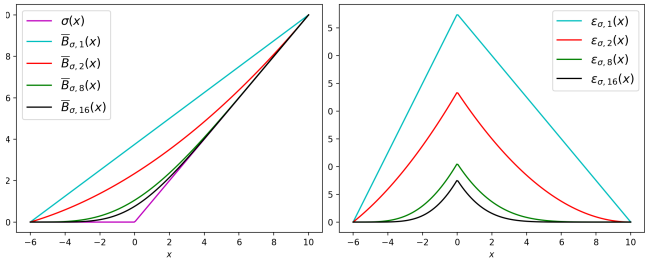

Figure 1: (Left) Bernstein polynomial approximations of ReLU activation for different approximation's order $L \in \{1, 2, 8, 16\}$, in the interval $I_1(-6, 10) = [-6, 10]$. (Right) Bernstein polynomial approximations of ReLU and their associated approximation errors for different approximation's order $L \in \{1, 2, 8, 16\}$ in the interval $I_1(-6, 10) = [-6, 10]$.

$$Ber_{K,L}(x) = \binom{L}{K} \frac{(x - \underline{d})^K \left(\overline{d} - x\right)^{L-K}}{\left(\overline{d} - \underline{d}\right)^L}, \tag{2}$$

$$b^g_{K,L} = g\left(\left(\overline{d}_1 - \underline{d}_1\right)\frac{k_1}{l_1} + \underline{d}_1, \cdots, \left(\overline{d}_n - \underline{d}_n\right)\frac{k_n}{l_n} + \underline{d}_n\right), \tag{3}$$

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

Bernstein polynomials are known to be capable of approximating any continuous function. That is, Bernstein approximation has an advantage compared to Taylor approximation because the latter relies on the function being differentiable. In this case, Taylor model can not approximate ReLU activation functions because they are not differentiable which makes Bernstein polynomials a good option to approximate ReLU functions. Bernstein polynomials have an interesting and useful property called *range enclosing property* which is defined as follows:

Definition 2. *(Range Enclosing Property [29]) Given a multi-dimensional polynomial $p(x)$ of order L that it defined over the region $I_n\left(\underline{d}, \overline{d}\right)$ with its Bernstein polynomial $B_{p,L} = \sum_{K \le L} b^p_{K,L}(x)\, Ber_{K,L}(x)$. The following holds for all $x \in I_n\left(\underline{d}, \overline{d}\right)$:*

$$\min_{K \le L} b^p_{K,L} \le p(x) \le \max_{K \le L} b^p_{K,L}. \tag{4}$$

The range enclosing property states that the minimum (maximum) over all the Bernstein coefficients is a lower (upper) bound for the polynomial p over the region $I_n\left(\underline{d}, \overline{d}\right)$. These bounds provided by the Bernstein coefficients are generally tighter than those given by interval arithmetic and many centered forms [30]. Note that the range enclosing property applies only when the Bernstein polynomial is used to approximate other polynomials p and other continuous functions g. Nevertheless, as we show in Section 4, these bounds will be helpful to provide tight bounds on the polynomials used to over/under approximate the individual neurons and hence obtain tight polynomial bounds on the NN's outputs.

3.1 Over-Approximating ReLU functions using Bernstein Polynomials

We now study how to use Bernstein polynomials to over-approximate the ReLU function $\sigma : \mathbb{R} \to \mathbb{R}$ defined as $\sigma(x) = \max(0, x)$. While Bernstein polynomials can approximate any continuous function g, there is no guarantee that this Bernstein approximation is either over-approximation or under-approximation. The next result establishes an order between the ReLU function σ and its Bernstein approximation.

Proposition 1. *Given an interval $I_1\left(\underline{d}, \overline{d}\right) = \left[\underline{d}, \overline{d}\right]$, where $0 \in \left[\underline{d}, \overline{d}\right]$ and any approximation order $L \ge 1$. The following holds for all $x \in I_1$:*

$$\sigma(x) \le B_{\sigma,L}(x) = \overline{B}_{\sigma,L}(x).$$

Proof. This follows directly by substituting the function σ in the definition of Bernstein polynomials (1)-(3). □

In other words, Proposition 1 states that the Bernstein polynomial of σ is a guaranteed over-approximation of σ. This even holds *for any approximation order L*. Moreover, since the approximation error between a function g and its Bernstein approximation $B_{g,L}$ is known to decrease as L increases [17]. Then another consequence of Proposition 1 is that Bernstein polynomials produce a tighter over-approximation for ReLU functions as L increases.

Figure 1 emphasizes these conclusions pictorially where we show the Bernstein polynomials of σ with orders $L = 1, 2, 8, 16$. As shown in Figure 1 (Left), the Bernstein polynomials $B_{\sigma,L}(x)$ for $L \in \{1, 2, 8, 16\}$ over-approximate the ReLU activation function over the entire input range. Furthermore, the over-approximation gets tighter to the actual ReLU by increasing the approximation order L. We note that using $L = 1$, the resulting Bernstein polynomial produces the well-studied linear convexification of the ReLU function which is used in state-of-the-art algorithms for bounding neural networks including Symbolic Interval Arithmetic (SIA) [35] and alpha-CROWN [41]. In other words, Bernstein polynomials can be seen as a generalization of these techniques.

3.2 Under-approximating ReLU functions using Bernstein polynomials

In addition to the over-approximation of the ReLU function σ, it is essential to establish a Bernstein under-approximation of σ which is captured by the following result.

Proposition 2. *Given an interval $I_1\left(\underline{d}, \overline{d}\right) = \left[\underline{d}, \overline{d}\right]$, where $0 \in \left[\underline{d}, \overline{d}\right]$, then the following holds for all $x \in I_1$:*

$$\underline{B}_{\sigma,L}(x) = \overline{B}_{\sigma,L}(x) - \overline{B}_{\sigma,L}(0) \le \sigma(x).$$

Proof. To prove the result, we define the approximation error $\epsilon_{\sigma,L}$ as:

$$\epsilon_{\sigma,L}(x) = \overline{B}_{\sigma,L}(x) - \sigma(x).$$

We bound the maximum estimation error satisfies as follows:

$$\max_{x \in [\underline{d}, \overline{d}]} \epsilon_{\sigma,L}(x) = \max_{x \in [\underline{d}, \overline{d}]} \left(\overline{B}_{\sigma,L}(x) - \sigma(x)\right) \tag{5}$$

$$\overset{(a)}{=} \max_{x \in [\underline{d}, 0]} \overline{B}_{\sigma,L}(x) \tag{6}$$

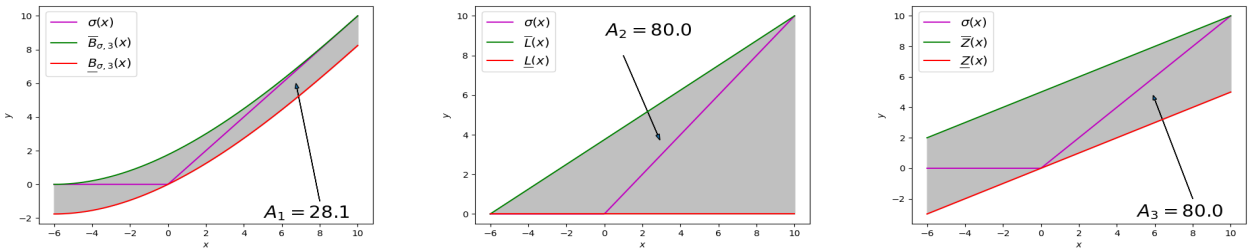

Figure 2: Illustrations of the over-approximation sets (shaded in gray) of the ReLU activation functions in the interval $[-6, 10]$ using different approaches: Bernstein approach (Left), triangulation approach (Center), and zonotope approach (Right). Green (Red)-colored curves represent the over-approximation (under-approximation) curves for every approach, respectively. A_i, $i \in \{1, 2, 3\}$, represents the over-approximation set's area for every approach.

$$\overset{(b)}{=} \overline{B}_{\sigma,L}(0) \tag{7}$$

where (a) follows from the fact that $\sigma(x) = 0$ for $x \in [\underline{d}, 0]$ and $\sigma(x) \geq 0$ for $x \in [0, \overline{d}]$ and hence the maximum of the equation is attained whenever $\sigma(x) = 0$. Equation (b) holds from the monotonicity of $\overline{B}_{\sigma,L}(x)$ when $x \in [\underline{d}, 0]$—the monotonicity follows directly from the definition of $\overline{B}_{\sigma,L}(x)$—and hence the maximum is attained when $x = 0$. It follows from the definition of $\epsilon_{\sigma,L}(x)$ that:

$$\sigma(x) = \overline{B}_{\sigma,L}(x) - \epsilon_{\sigma,L}(x) \geq \overline{B}_{\sigma,L}(x) - \max_{x \in [\underline{d}, \overline{d}]} \epsilon_{\sigma,L}(x)$$
$$= \overline{B}_{\sigma,L}(x) - \overline{B}_{\sigma,L}(0) = \underline{B}_{\sigma,L}$$

which concludes the proof. □

Proposition 2 shows that the maximum error between the Bernstein over-approximation polynomial $\overline{B}_{\sigma,L}$ and the ReLU activation function σ is equal to the value of the Bernstein polynomial at 0, i.e., $\overline{B}_{\sigma,L}(0)$. This result has a direct consequence on the efficiency of our tool. It is enough to propagate over-approximation of the ReLU function and one can get an under-approximation directly by shifting the over-approximation polynomial.

Figure 1 (Right) emphasizes this fact pictorially. As it is shown in the figure, the maximum error $\epsilon_{\sigma,L}(x) = \overline{B}_{\sigma,L} - \sigma(x)$ is reached at $x = 0$ and is equal to $\overline{B}_{\sigma,L}(0)$.

Table 1: The area of the over-approximation set of the ReLU activation functions in the interval $[-6, 10]$ using different Bernstein approach for different approximation order L.

Approx. Method	Triangulation	Zonotope	Bernstein poly		
			$L = 2$	$L = 3$	$L = 8$
error	80.0	80.0	37.5	28.1	16.9

3.3 Comparing Bernstein Approximation Against Widely Used Approximations

The major advantage of using Bernstein polynomials is that they produce a tighter approximation for the response function of ReLU compared to the other state-of-the-art techniques. In particular, existing techniques focus on "convexifying" the response of the ReLU function through linear approximation/triangulation (Figure 2-middle) or zonotopes (Figure 2-right). Unlike these techniques, Bernstein polynomials lead to tighter non-convex approximations of the non-convex ReLU function. While it is direct to obtain a closed-form expression for the difference in the approximation error between Bernstein polynomials and triangulation/zonotope approximations, we, instead support our conclusions with the numerical example shown in Table 1 and highlighted in Figure 2. In this example, we compute the approximation error (highlighted in gray) which captures the quality of the over and under-approximations. As captured by this example, it is direct to see that Bernstein polynomials lead to tighter approximation. Moreover, such approximation gets tighter as the approximation order L increases.

4 ENCODING BASIC BERNSTEIN POLYNOMIAL OPERATIONS USING MULTI-DIMENSIONAL TENSORS

While using Bernstein polynomials to approximate individual ReLU functions provides tighter bounds compared to other techniques, computing Bernstein polynomials via its definition in (1)-(3) is time-consuming. That is why state-of-the-art techniques have focused on linear (or convex) relaxations to obtain tractable computations. Nevertheless, in this section, we show that technological advances in Graphics Processing Units (GPUs) can be used to perform all the required operations to efficiently compute Bernstein polynomial approximations of individual neurons along with propagating these polynomials from one layer of the neural network to the next layer. Our main contribution of this section is to encode all necessary operations over Bernstein polynomials into additions and multiplication of multi-dimensional tensors that can be easily performed using GPUs.

4.1 Multi-dimensional tensor representation of Bernstein polynomials

We represent the Bernstein polynomial:

$$B_{g,L}(x) = \sum_{K \leq L} b^{g}_{K,L} Ber_{K,L}(x)$$

of function g and order L as a multi-dimensional tensor $\text{Ten}(B_{g,L})$ of n dimensions, and of a shape of $L = (l_1 + 1, \cdots, l_n + 1)$, where the $K = (k_1, \cdots, k_n)$ component of $\text{Ten}(B_{g,L})$ is equal to the Bernstein

coefficient $b^g_{K,L}$. The multi-dimensional tensor $\text{Ten}(B_{g,L})$ represent all the Bernstein coefficients $b^g_{K,L}$ of g, $\forall K \leq L$.

Example 1. *Consider the two-dimensional Bernstein polynomial:*

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

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

$$Ten\left(B_{g,L}\right) = \begin{bmatrix} b^g_{(0,0),L} & b^g_{(0,1),L} & b^g_{(0,2),L} & b^g_{(0,3),L} \\ b^g_{(1,0),L} & b^g_{(1,1),L} & b^g_{(1,2),L} & b^g_{(1,3),L} \\ b^g_{(2,0),L} & b^g_{(2,1),L} & b^g_{(2,2),L} & b^g_{(2,3),L} \end{bmatrix}. \quad (8)$$

In a similar manner, we represent a multi-dimensional polynomial of order L written in the power series form $p(x) = \sum_{K \leq L} a_K x^K$ as a multi-dimensional tensor $\text{Ten}(p)$ of n dimensions, and of a shape of $L = (l_1 + 1, \cdots, l_n + 1)$, where the $K = (k_1, \cdots, k_n)$ component of $\text{Ten}(p)$ is equal to the coefficient a_K.

4.2 Multiplication of two multi-variate Bernstein polynomials

Multiplying two polynomials represented in the power series form on GPUs has been widely studied in the literature. Unlike power series, multiplying two Bernstein polynomials need extra handling [28]. In this subsection, we propose how to encode the multiplication of Bernstein polynomials using GPU implementations that were designed for power-series polynomials.

Given two multivariate polynomials written in a power series form, $p_1 = \sum_{K \leq L_1} a^1_K x^K$ and $p_2 = \sum_{K \leq L_2} a^2_K x^K$, and their tensor representation, $\text{Ten}(p_1)$ and $\text{Ten}(p_2)$, we use an efficient algorithm [26] that performs multivariate polynomial multiplications. We denote by $\textbf{Prod}(\text{Ten}(p_1), \text{Ten}(p_2))$ the tensor resulting from such multiplication, i.e.:

$$\text{Ten}(p_1 p_2) = \textbf{Prod}(\text{Ten}(p_1), \text{Ten}(p_2)).$$

Applying power-series-based algorithms to multiply two Bernstein polynomials produce incorrect results. Different algorithms were proposed for the case when the Bernstein polynomials are functions of one variable x_1 [13] and two variables x_1, x_2 [28]. Below, we generalize the procedure in [28] to account for Bernstein polynomials in n variables.

Proposition 3. *Given two multivariate Bernstein polynomials $B_{g_1,L_1}(x) = \sum_{K \leq L_1} b^{g_1}_{K,L_1} Ber_{K,L_1}(x)$ and $B_{g_2,L_2}(x) = \sum_{K \leq L_2} b^{g_2}_{K,L_2} Ber_{K,L}(x)$. The tensor representation of the Bernstein polynomial $B_{g_1,L_1}(x) B_{g_2,L_2}(x)$ can be computed as follows:*

$$Ten\left(\tilde{B}_{g_1,L_1}\right) = Ten\left(B_{g_1,L_1}\right) * C_{L_1}, \quad (9)$$

$$Ten\left(\tilde{B}_{g_2,L_2}\right) = Ten\left(B_{g_2,L_2}\right) * C_{L_2}, \quad (10)$$

$$Ten\left(B_{g_1,L_1} B_{g_2,L_2}\right) = \frac{1}{C_{L_1+L_2}} * \textbf{Prod}\left(Ten\left(\tilde{B}_{g_1,L_1}\right), Ten\left(\tilde{B}_{g_2,L_2}\right)\right). \quad (11)$$

where C_L is the multi-dimensional binomial tensor where its Kth component is equal to $\binom{L}{K}$, i.e., $(C_L)_K = \binom{L}{K}$. With some abuse of notation, we use $1/C_L$ to denote the multi-dimensional binomial tensor where its Kth component is equal to $\frac{1}{\binom{L}{K}}$.

The proof of Proposition 3 generalizes the argument in [28] to multi-dimensional inputs and is omitted for brevity. The Bernstein polynomials in (9) and (10) are called scaled Bernstein polynomials [28] and enjoy the fact that their multiplication corresponds to the multiplication of power series polynomials. Hence we can use the power series **Prod** in (11) followed by the element-wise multiplication with the $\frac{1}{C_{L_1+L_2}}$ tensor to remove the effect of the scaling. Recall that we use $A * B$ to denote the element-wise multiplication between the tensors A and B, which can also be carried over using GPUs efficiently which renders all the steps in equations (9)-(11) to be efficiently implementable on GPUs. We refer to the equations (9)-(11) as $\textbf{Prod_Bern}(B_{g_1,L_1}, B_{g_2,L_2})$.

Using **Prod_Bern**, one can compute the tensor corresponding to raising the function g to power i, where $i \in \mathbb{N}$ is an integer power, denoted by $\text{Ten}(B_{g^i,L})$ by applying the **Prod_Bern** procedure i times. We refer to this procedure as $\textbf{Pow_Bern}(\text{Ten}(B_{g,L}), i)$.

4.3 Addition between two Bernstein polynomials

The authors in [13] studied how to add two Bernstein polynomials. However, their study is restricted to one-dimensional polynomials which are defined over the unity interval $I_1(x) = [0, 1]$. We extend the argument to the general case with n inputs and any interval $I_n(\underline{d}, \overline{d})$ using the following result.

Proposition 4. *Given two Bernstein polynomials $B_{g_1,L_1}(x)$ and $B_{g_2,L_2}(x)$ with two different orders $L_1 = (l^1_1, \cdots, l^1_n)$ and $L_2 = (l^2_1, \cdots, l^2_n)$. Define $L_{sum} = \max(L_1, L_2)$, where the* max *operator is applied element-wise. The tensor representation of $B_{g_1+g_2,L_{sum}}$ can be computed as:*

$$L_{sum} = (\max(l^1_1, l^2_1), \ldots, \max(l^1_n, l^2_n)) \quad (12)$$

$$Ten\left(B_{g_1,L_{sum}}\right) = \textbf{Prod_Bern}\left(Ten\left(B_{g_1,L_1}\right), 1_{L_{sum}-L_1+1}\right) \quad (13)$$

$$Ten\left(B_{g_2,L_{sum}}\right) = \textbf{Prod_Bern}\left(Ten\left(B_{g_2,L_2}\right), 1_{L_{sum}-L_2+1}\right) \quad (14)$$

$$Ten\left(B_{g_1+g_2,L_{sum}}\right) = Ten\left(B_{g_1,L_{sum}}\right) + Ten\left(B_{g_2,L_{sum}}\right) \quad (15)$$

where 1_{L_e-L+1} is a multi-dimensional tensor of a shape $L_e - L + 1$ that contains just ones.

The proof of Proposition 4 generalizes the argument in [13] and is omitted for brevity. The operation in (13) and (14) is referred to as *degree elevation* in which we change the dimensions of the tensors ... Once both tensors are of the same dimension, we can add them element-wise. We denote by **Sum_Bern** the procedure defined by (12)-(15). Again, we note that all the operations in the **Sum_Bern** entail tensor element-wise multiplication and addition

5 BERN-NN ALGORITHM

In this section, we provide the details of our tool, named BERN-NN. BERN-NN uses the tensor encoding discussed in Section 4 to propagate Bernstein polynomials that over- and under-approximate

the different neurons in the network until over- and under-approximation polynomials for the final output of the network are computed.

5.1 Propagating bounds through single neuron

We first discuss how to propagate over- and under-approximations through neurons. Recall our notation that we use $\hat{z}_j^{(i)}$ and $z_j^{(i)}$ to denote the input and output of the j-th neuron in the i-th layer. For ease of notation, we drop the i and j from the notation in this subsection.

Assume that we already computed the over- and under-approximations for the input of one of the hidden neurons, denoted by $\overline{B}_{\hat{z},L_{\hat{z}}}(x)$ and $\underline{B}_{\hat{z},L_{\hat{z}}}(x)$, respectively. The objective is to compute the over- and under-approximations for the output of such a neuron, denoted by $\overline{B}_{z,L_z}(x)$ and $\underline{B}_{z,L_z}(x)$, respectively. We proceed as follows.

Step 1: Compute input bounds for the neuron. Recall that the Bernstein coefficients depend on the input bounds of the function it aims to approximate. Since our aim is to approximate the scalar ReLU function of a neuron, we start by computing the bounds on the input to that neuron as follows:

$$lo = \min_{x \in I_n(\underline{d},\overline{d})} \underline{B}_{\hat{z},L_{\hat{z}}}(x), \qquad hi = \max_{x \in I_n(\underline{d},\overline{d})} \overline{B}_{\hat{z},L_{\hat{z}}}(x) \tag{16}$$

Thanks to the enclosure property (4), we can solve the optimization problems (16) by finding the minimum and the maximum coefficients of $\underline{B}_{\hat{z},L_{\hat{z}}}$ and $\overline{B}_{\hat{z},L_{\hat{z}}}$.

Step 2: Compute the polynomials $\overline{B}_{\sigma,L}$ and $\underline{B}_{\sigma,L}$ that approximate the ReLU function. Given a user-defined approximation order L, the next step is to compute the Bernstein polynomials that over- and under-approximate the ReLU activation function σ denoted by $\overline{B}_{\sigma,L}$ and $\underline{B}_{\sigma,L}$. These polynomials can be computed using the knowledge of lo and hi along with the definition of the Bernstein polynomial in (3). To facilitate the computations of the next step, we need to convert these polynomials into the corresponding power series form. This can be done by following the procedure in [27] to obtain:

$$p_{\overline{B}_{\sigma,L}}(x) = \sum_{K \le L} a_K^{\overline{B}_{\sigma,L}} x^K, \qquad p_{\underline{B}_{\sigma,L}}(x) = \sum_{K \le L} a_K^{\underline{B}_{\sigma,L}} x^K \tag{17}$$

Step 3: Propagate the bounds through the decomposition of polynomials. First, note that the following holds due to the monotonicity of the ReLU function σ and the fact that $z = \sigma(\hat{z})$:

$$\underline{B}_{\hat{z},L_{\hat{z}}}(x) \le \hat{z}(x) \le \overline{B}_{\hat{z},L_{\hat{z}}}(x) \Rightarrow \tag{18}$$

$$\underline{B}_{z,L_z}(x) \le \sigma\left(\underline{B}_{\hat{z},L_{\hat{z}}}(x)\right) \le \underbrace{\sigma\left(\hat{z}(x)\right)}_{z(x)} \le \sigma\left(\overline{B}_{\hat{z},L_{\hat{z}}}(x)\right) \le \overline{B}_{z,L_z}(x) \tag{19}$$

In other words, the post-bounds of the neuron, denoted by $\overline{B}_{z,L_z}(x)$ and $\underline{B}_{z,L_z}(x)$ can be computed by composing the function σ with the under- and over-approximations of the neuron input $\underline{B}_{\hat{z},L_{\hat{z}}}(x)$ and $\overline{B}_{\hat{z},L_{\hat{z}}}(x)$. Indeed such composition is hard to compute due to the nonlinearity in σ. Instead, we perform such composition with the over- and under-approximations of σ, $p_{\overline{B}_{\sigma,L}}$ and $p_{\underline{B}_{\sigma,L}}$, computed in Step 2, as:

$$\underline{B}_{z,L_z}(x) = \sum_{K \le L} a_K^{\underline{B}_{\sigma,L}} \left(\underline{B}_{\hat{z},L_{\hat{z}}}(x)\right)^K \tag{20}$$

$$\overline{B}_{z,L_z}(x) = \sum_{K \le L} a_K^{\overline{B}_{\sigma,L}} \left(\overline{B}_{\hat{z},L_{\hat{z}}}(x)\right)^K \tag{21}$$

Given the tensor representation $Ten(\underline{B}_{\hat{z},L_{\hat{z}}})$ and $Ten(\overline{B}_{\hat{z},L_{\hat{z}}})$, we can use the **Pow_Bern** and **Sum_Bern** procedures to perform the computations in (20) and (21) to calculate $Ten(\underline{B}_{z,L_z})$ and $Ten(\overline{B}_{z,L_z})$ with $L_z = L_{\hat{z}} * L$.

5.2 Propagating the bounds through one layer

Next, we discuss how to propagate the under- and over-approximation polynomials of the outputs of the $i-1$ layer denoted by $\underline{B}_{z_j^{(i-1)},\,L_z}, \overline{B}_{z_j^{(i-1)},\,L_z}, j \in \{1,\dots,h_{i-1}\}$ to compute under- and over-approximation of the inputs of the neurons in the ith layer $\underline{B}_{\hat{z}_m^{(i)},\,L_{\hat{z}}}, \overline{B}_{z_m^{(i)},\,L_{\hat{z}}}, m \in \{1,\dots,h_i\}$ of the neural network. Such bound propagation entails composing the under- and over-approximation polynomials $\underline{B}_{z_j^{(i-1)},\,L_z}, \overline{B}_{z_j^{(i-1)},\,L_z}$ with the weights of the ith layer of the neural network $W^{(i)}, b^{(i)}$. To that end, we define the set of positive and negative weights as:

$$W_+^{(i)} = \max\left(W^{(i)}, 0_{i\times(i-1)}\right) \qquad W_-^{(i)} = \min\left(W^{(i)}, 0_{i\times(i-1)}\right).$$

Similarly, for the outputs of the $i-1$ layer of the network, we define the vector of over-approximation polynomials and vector of the under-approximation polynomials as:

$$\overline{B}_{z^{(i-1)},\,L_z} = \left[\overline{B}_{z_1^{(i-1)},L_z} \dots, \overline{B}_{z_{h_{i-1}}^{(i-1)},L_z}\right]^T,$$

$$\underline{B}_{z^{(i-1)},\,L_z} = \left[\underline{B}_{z_1^{(i-1)},L_z} \dots, \underline{B}_{z_{h_{i-1}}^{(i-1)},L_z}\right]^T,$$

and for the inputs of the $i$the layer as:

$$\overline{B}_{\hat{z}^{(i)},\,L_{\hat{z}}} = \left[\overline{B}_{\hat{z}_1^{(i)},L_{\hat{z}}} \dots, \overline{B}_{\hat{z}_{h_i}^{(i)},L_{\hat{z}}}\right]^T$$

$$\underline{B}_{\hat{z}^{(i)},\,L_{\hat{z}}} = \left[\underline{B}_{\hat{z}_1^{(i)},L_{\hat{z}}} \dots, \underline{B}_{\hat{z}_{h_i}^{(i)},L_{\hat{z}}}\right]^T$$

Hence, the over- and under-approximations of the inputs of the ith layer can be efficiently computed as:

$$Ten\left(\overline{B}_{\hat{z}^{(i)},L_{\hat{z}}}\right) = W_+^{(i)} \times Ten\left(\overline{B}_{z^{(i-1)},L_z}\right) + W_-^{(i)} \times Ten\left(\underline{B}_{z^{(i-1)},L_z}\right) + b^{(i)} \tag{22}$$

$$Ten\left(\underline{B}_{\hat{z}^{(i)},L_{\hat{z}}}\right) = W_+^{(i)} \times Ten\left(\underline{B}_{z^{(i-1)},L_z}\right) + W_-^{(i)} \times Ten\left(\overline{B}_{z^{(i-1)},L_z}\right) + b^{(i)} \tag{23}$$

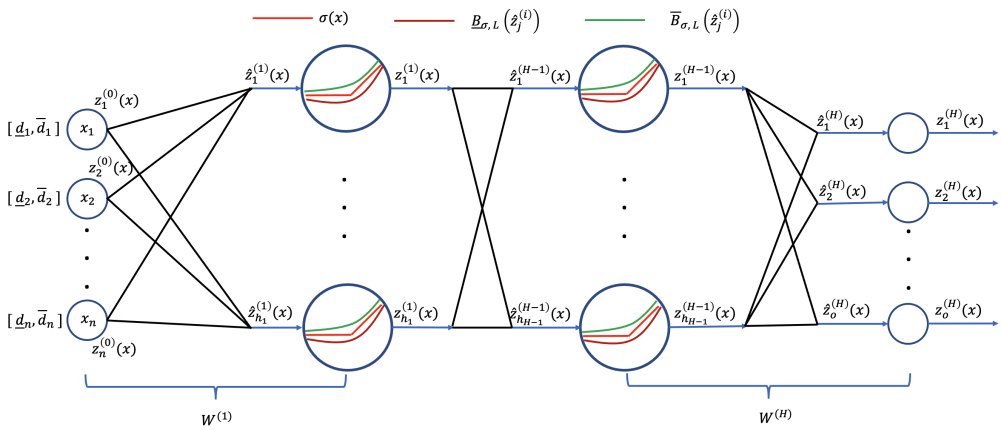

Figure 3: Mechanism of BERN-NN Polynomial Interval Arithmetic.

5.3 Mechanism of BERN-NN Polynomial Interval Arithmetic

We finally describe the proposed BERN-NN Polynomial Interval Arithmetic algorithm, depicted in Figure 3. For a neural network with n inputs $x_1, \ldots, x_n$, we initialize an over- and under-approximation Bernstein polynomials for each of the inputs, i.e.,:

$$\overline{B}_{z_i^{(0)},1} = \underline{B}_{z_i^{(0)},1} = \overline{B}_{z_i^{(0)},1} \qquad i \in \{1, \ldots, n\}.$$

Note that in the equation above, we used $z_i^{(0)}$ as a replacement of x_i to unify the notation with the remainder of the operations (see Figure 3). To compute the Bernstein polynomials $\overline{B}_{z_i^{(0)},1}$ and $\underline{B}_{z_i^{(0)},1}$, we recall that the coefficients of such polynomials depend on the input domain. Hence, given a hypercube $I_n(\underline{d}, \overline{d})$ that bounds the input x of the neural network, we compute the tensor representation of these polynomials as:

$$Ten\left(\overline{B}_{z_1^{(0)},1}\right) = Ten\left(\underline{B}_{z_1^{(0)},1}\right) = \begin{bmatrix} \underline{d}_1 \\ \overline{d}_1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{24}$$

$$Ten\left(\overline{B}_{z_2^{(0)},1}\right) = Ten\left(\underline{B}_{z_2^{(0)},1}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} \underline{d}_2 \\ \overline{d}_2 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{25}$$

$$\vdots$$

$$Ten\left(\overline{B}_{z_n^{(0)},1}\right) = Ten\left(\underline{B}_{z_n^{(0)},1}\right) = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 1 \end{bmatrix} \otimes \ldots \otimes \begin{bmatrix} \underline{d}_n \\ \overline{d}_n \end{bmatrix} \tag{26}$$

Next, we propagate these over- and under-approximation polynomials to the inputs of the first layer in the neural network using (22) and (23). Given a user-defined approximation order L, we propagate the polynomial approximations through the ReLU function using (20) and (21) for each of the neurons in layer 1. The produced over- and under-approximations of the outputs of all neurons are aggregated together in one tensor which is then propagated to the next layer. This process continues until we compute the over- and under-approximation polynomials of the outputs of the neural network, denoted by $\overline{B}_{z_j^{(H)},L^{H-1}}(x), \underline{B}_{z_j^{(H)},L^{H-1}}(x)$ for $j = 1, \ldots, o$. These polynomials are used as the solution of Problem 1.

It is important to note that the final Bernstein polynomials $\overline{B}_{z_j^{(H)},L^{H-1}}(x), \underline{B}_{z_j^{(H)},L^{H-1}}(x)$ have orders of L^{H-1} where L is the user-defined order of approximation of the ReLU function and H is the number of layers. This polynomial order increases exponentially with the number of hidden layers. Similarly, the shape of their multi-dimensional tensor representations is equal to $L^{H-1}+1$ which increases exponentially with the number of hidden layers. To alleviate this problem, we introduce a parameter called Lin. Based on this parameter, we drop the orders of the post-bound over- and under-approximation polynomials to $[1, \cdots, 1]$. In other words, we linearize the approximation polynomials every Lin hidden layers. We use the algorithm in [18] to perform such linearization of the Bernstein polynomial. Luckily, this algorithm, like all the other operations in our BERN-NN involves tensor multiplications and additions and hence can be parallelized over GPUs efficiently.

Finally, note that one can always obtain absolute bounds on the inputs or outputs of any of the neurons (including the outputs of the neural network), thanks to the enclosure property of Bernstein polynomials (4). Such absolute bounds are useful for reachability analysis and model checkers.

5.4 GPU Implementation Details

To get the performance increase of GPUs without the complications of low-level languages, we implemented this tool in PyTorch. As mentioned above, we represent n-dimensional Bernstein polynomials as dense n-dimensional tensors. The tool becomes memory bound very quickly as the number of input nodes increases, making the number of dimensions in the tensors larger. In order to combat this, we use as many in-place operations as possible to avoid repeatedly allocating large chunks of memory during computation. Similarly, the multinomial coefficients used for degree elevation are used multiple times throughout the tool, and we cache each the first time they are generated to avoid spending time re-doing calculations and allocating additional memory.

We parallelized the tool on a node level: at each layer, the outputs of the last layer are passed to each node, which then can run independently of each other on separate GPUs. However, because

the tensors become large very quickly, the gains in computation time only offset the overhead of copying tensors between GPUs when the neural network is particularly large. We collect and stack the outputs of all the nodes in one tensor and pass it to the next layer. When the polynomials are being composed with the ReLU approximation, each term is elevated to the highest degree expected of a composition between these two polynomials. This both ensures that the outputs of all the neurons can be stacked, as they are all the same shape and size, and also allows the multiplication of the stacked outputs of the last layer by the incoming weights to be a simple broadcasting multiplication, which is then easily parallelizable on a GPU.

We achieved additional performance gains by rewriting for-loops as element-wise tensor operations and by batching linear algebra operations like matrix multiplications and calculating the least-square solutions of matrices, both of which allow operations to be easily parallelized on GPUs and reduce the amount of time spent allocating many small patches of memory, instead doing a single large allocation.

6 NUMERICAL RESULTS

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., neural network width, neural network depth, ReLU approximation order) on the performance of our tool. We utilize two metrics:

- **Execution time**: which measures the time (in seconds) needed to compute the final Bernstein polynomials. Indeed, smaller values indicate better performance.
- **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 neural networks with one output $z^{(H)}$ and we compute this metric as:

$$\text{Vol_relative} = \frac{\text{Vol_Output}}{\text{Vol_Input}} \tag{27}$$

$$\text{Vol_Input} = \prod_{i=1}^{n} \left(\overline{d}_i - \underline{d}_i \right) \tag{28}$$

$$\text{Vol_Output} = \int \cdots \int_{I_n} \left(\overline{B}_{z^{(H)}}(x) - \underline{B}_{z^{(H)}}(x) \right) dx_1 \ldots dx_n \tag{29}$$

  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 2022 Verification of Neural Network (VNN) competition [2]—to study the relative performance.

Setup: We implemented our tool in Python3.9 using PyTorch for all tensor arithmetic. We run all our experiments using a single GeForce RTX 2080 Ti GPU and two 24-core Intel(R) Xeon(R). We like to note that the throughput of the tool can be increased by utilizing multiple GPU to process different neurons in parallel in a batch-processing fashion. However, in this section, we focus on using only one GPU and we leave the generalization of our algorithm to utilize multiple GPUs for future work.

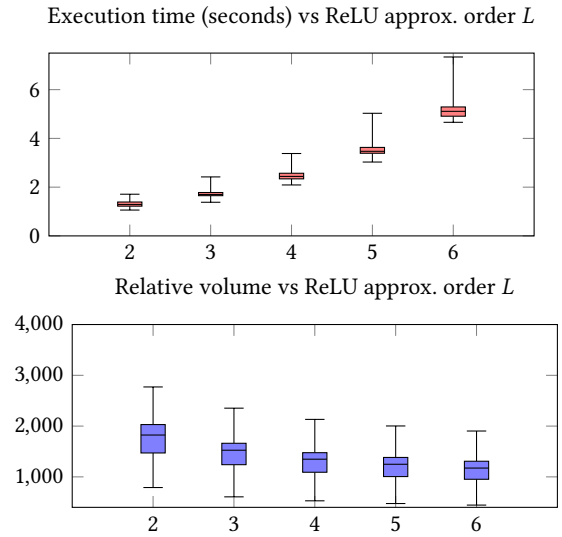

Figure 4: Effect of varying the ReLU's order of approximation L for a NN architecture $[2, 20, 20, 1]$ on the execution time of our tool (top) and the relative volume of the output set (bottom). We set $n = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

6.1 Ablation study

6.1.1 The effect of varying the ReLU's order of approximation: We study the effect of varying the ReLU's order of approximation L for a fixed NN architecture on the execution time and the output's relative volume space of our tool. In Figure 4, we report the statistical results for 50 random networks of a fixed architecture. Figure 4 (top) shows that increasing the approximation order increases the execution time. On the other hand, Figure 4 (bottom) shows that the relative volume of the output set significantly decreases with increasing the order of approximation. The results of both figures highlight the trade-off between the tightness of the output bounds and the execution time as a function of the ReLU approximation order L.

6.1.2 The effect of varying the input's dimension: We study the effect of varying the input's dimension n, for a fixed NN architecture on the execution time of our tool. Figure 5 shows that the execution time for computing the output set grows linearly for smaller values of n but seems to grow more rapidly after $n = 7$. This suggests that the proposed tool can be used efficiently for many control applications.

6.1.3 The effect of increasing the number of neurons per layer: We study the effect of varying the number of neurons per layer N_e, for a fixed NN architecture $[3, N_e, N_e, 1]$ on the execution time of our tool. Figure 6 summarizes the execution times with a varying number of neurons per layer. The results show that increasing the number of neurons per layer highly affects the execution time. This

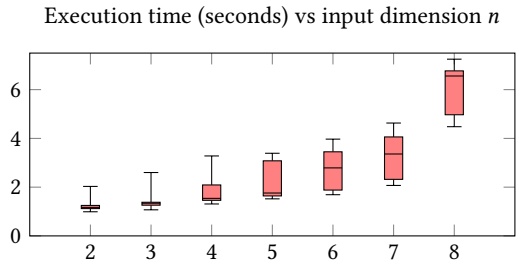

Figure 5: Effect of varying the input's dimension n for a NN architecture $[n, 20, 20, 1]$ on the execution time our tool. We set $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

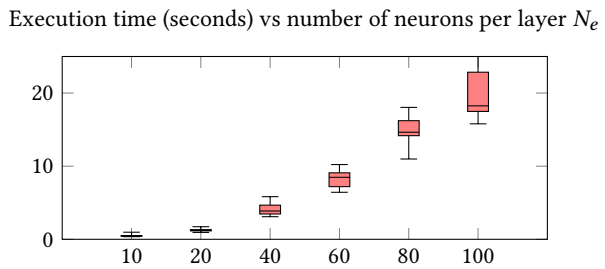

Figure 6: Effect of varying the number of neurons per layer N_e for a NN architecture $[2, N_e, N_e, 1]$ on the execution time of our tool. We set $n = 2$, $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

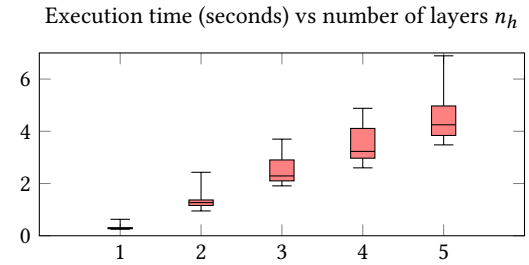

Figure 7: Effect of varying the number of hidden layers n_h, for a NN architecture $[2, 20, .., 20, 1]$ with 20 neurons in every hidden layer on the execution time of our tool. We set $n = 2$, $L = 2$, $I_n = [-1, 1]^n$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. The reported results are generated for 50 experiments.

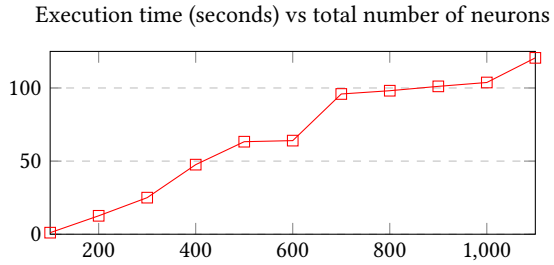

Figure 8: Scalability of the Bern-NN tool as a function of increasing the total number of neurons.

is due to the expensive arithmetic and memory operations for large tensors that represent the Bernstein polynomials. Nevertheless, this increase in execution time can be harnessed by using multiple GPUs to compute bounds for different nodes in parallel along with using the same GPU to process multiple nodes simultaneously.

6.1.4 The effect of increasing the number of hidden layers: We study the effect of varying the number of hidden layers n_h, with 20 neurons in every hidden layer, on the execution time of our tool. Unlike the effect of increasing the number of neurons per layer, the results in Figure 7 show that the execution time almost grows linearly with the number of hidden layers.

6.1.5 Scalability analysis of Bern-NN:. We finally try to study the execution time of Bern-NN for relatively large neural networks. In this study, we add extra layers with 100 neurons each and report the execution time in Figure 8 for random neural networks. As shown in the figure, Bern-NN can process neural networks with more than 1000 neurons in less than 2 minutes.

6.2 Comparison against other tools

In this subsection, 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)[35], alpha-CROWN [41], and reachability analysis tool such as POLAR [21]. We note that alpha-CROWN [41] was the winner of the 2022 VNN competition and 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. We compare Bern-NN against POLAR since it also uses polynomials (Taylor Model with a Bernstein error correction) to compute bounds on the output of neural networks. POLAR [21] outperforms other reachability-based tools and hence is a representative tool for such techniques.

6.2.1 Comparison against SIA and alpha-CROWN for random NN. We compare the performance of our tool to SIA and alpha-CROWN for random neural networks with $[2, 20, 20, 1]$ architecture for different hyperrectangle input spaces (Figure 9). We also compare the performance as the input dimension of the network increases (Figure 10). 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's relative volume is the smallest for all different input spaces thanks to its tight higher-order ReLU approximations. Compared to alpha-CROWN (which also runs on GPUs), Bern-NN is both faster and produces tighter bounds leading to an average of 25% reduction in execution time with an average of 10% reduction in the relative volume metric. This shows the practicality of Bern-NN for control applications.

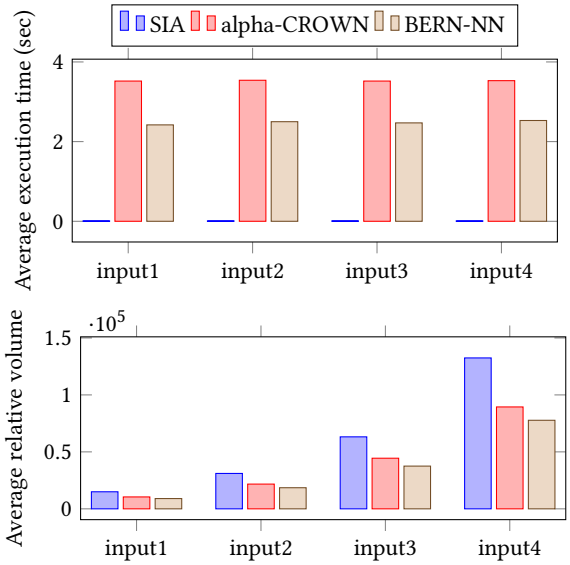

Figure 9: Performance results in terms of average execution times (top) and relative volume (bottom) for BERN-NN, SIA, and alpha-CROWN for different input spaces. The NN's architecture is $[2, 20, 20, 1]$. The ReLU's order of approximation is $L = 4$, and $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. Input1 = $I_n = [-5, 5]^2$, Input2 = $I_n = [-10, 10]^2$, Input3 = $I_n = [-20, 20]^2$, Input4 = $I_n = [-40, 40]^2$.

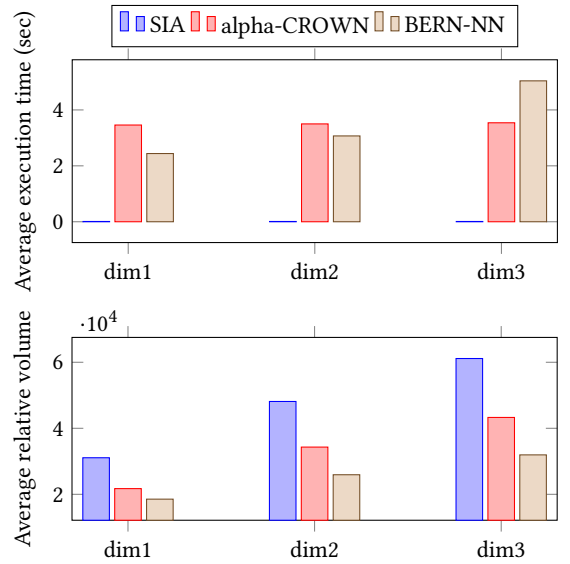

Figure 10: Performance results in terms of average execution times (top) and relative volume (bottom) for BERN-NN, SIA, and alpha-CROWN for input's dimensions n. The NN's architecture is $[n, 20, 20, 1]$. the input's space is $[-10, 10]^n$. The ReLU's order of approximation is $L = 4$, $Lin = 0$. The weights and biases are generated randomly following uniform distribution between -5 and 5. dim1 = $n = 2$, dim2 = $n = 3$, dim3 = $n = 4$.

6.2.2 Case Study for Control Benchmarks. In this experiment, we test different tools on benchmarks of NN controllers (used by POLAR) to evaluate the tightness of their estimated bounds. POLAR and BERN-NN use different Bernstein polynomials to approximate ReLU functions. Proposition 2 shows the maximum error of ReLU's Bernstein overapproximation equals the Bernstein approximation at 0. This error is the smallest for ReLU's Bernstein overapproximation. However, POLAR approximates the ReLU using samples and adds a symmetric error bound estimated using the Lipschitz constant of ReLU functions. Such a method is conservative and leads to additional errors for each ReLU function. Table 2 represents six standard benchmarks used by the authors in [21] to evaluate the POLAR tool. Every benchmark represents a trained NN controller for a closed-loop system. Numbers in the square brackets refer to NN architecture, e.g., [2,20,20,1] means the NN has an input layer of two neurons, two hidden layers of 20 neurons, and an output layer of 1 neuron. Table 2 summarizes the performance of the tools with respect to the average execution time and average relative volume for six control benchmarks. The results show that Bern-NN provides the tightest estimate for the output set for all benchmarks except Benchmark 3. We would like to highlight that the tight approximation provided by Bern-NN is important for control applications because the specification of interest is usually defined over a time horizon and require multi-step reachability, hence, tighter bounds at each step are crucial. Lastly, Bern-NN is faster than alpha-CROWN over all benchmarks except Benchmark 5. However, SIA and POLAR are faster than Bern-NN but provide looser bound estimates. Each benchmark is run with five different hyperrectangles that are all centered around zero and have a radius $r \in \{1, 1.5, 2, 2.5, 3\}$.

Table 2: Performance results in terms of average execution times and volume for BERN-NN, SIA, alpha-CROWN, and POLAR, for 5 different input's spaces $I_n\left(\underline{d}, \overline{d}\right)$ for 6 benchmarks [21]. The ReLU's order of approximation is $L = 2$, $Lin = 0$.

Tool	Benchmark 1		Benchmark 2		Benchmark 3		Benchmark 4		Benchmark 5		Benchmark 6	
	[2,20,20,1]		[2,20,20,1]		[2,20,20,1]		[3,20,20,1]		[3,100,100,1]		[4,20,20,20,1]	
	time	volume	time	volume	time	volume	time	volume	time	volume	time	volume
SIA	**0.01**	2.544	**0.02**	6.05	**0.01**	1.02	**0.01**	9.41	**0.02**	53.38	**0.02**	2.03
CROWN	2.9	3.1	3.49	5.50	3.54	**0.73**	3.13	17.04	3.80	77.72	4.10	2.4
Bern − NN	0.84	**1.62**	1.30	**5.4**	1.09	0.81	1.15	**6.21**	41.7	**35.85**	3.25	**1.38**
POLAR	0.21	25.43	0.284	51.80	0.29	18.81	0.42	33.32	5.52	432.75	0.81	7.00

7 CONCLUSION

In conclusion, we presented Bern-NN, a tool for computing higher-order tight bounds for NNs by approximating non-linear ReLU activations using Bernstein polynomials. We provided GPU-based computational machinery to handle tensor arithmetic for manipulating polynomials as well as bounding them using the properties of Bernstein polynomials. We conducted extensive experiments to evaluate the scalability of our tool as well as compare its estimated bounds with state-of-the-art methods. The results showed that our tool can process neural networks with thousands of neurons in a few minutes. These results also show that our tool outperforms state-of-the-art tools in terms of computing tighter bounds while reducing the execution time compared to other tools.

ACKNOWLEDGMENTS

This work was partially sponsored by the NSF awards #CNS-2002405 and #ECCS-2139781 and the C3.AI Digital Transformation Institute.

REFERENCES

[1] Ross Anderson, Joey Huchette, Will Ma, Christian Tjandraatmadja, and Juan Pablo Vielma. 2020. Strong mixed-integer programming formulations for trained neural networks. *Mathematical Programming* 183, 1 (2020), 3–39. https://doi.org/10.1007/s10107-020-01474-5

[2] Stanley Bak, Changliu Liu, and Taylor T. Johnson. 2021. The Second International Verification of Neural Networks Competition (VNN-COMP 2021): Summary and Results. *CoRR* abs/2109.00498 (2021), 1–15.

[3] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. 2016. Measuring Neural Net Robustness with Constraints. In *Advances in Neural Information Processing Systems*, Vol. 29. Association for Computing Machinery, Barcelona, Spain, 2613–2621.

[4] Elena Botoeva, Panagiotis Kouvaros, Jan Kronqvist, Alessio Lomuscio, and Ruth Misener. 2020. Efficient verification of relu-based neural networks via dependency analysis. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 4 (2020), 3291–3299.

[5] Rudy Bunel, Jingyue Lu, Ilker Turkaslan, P Kohli, P Torr, and P Mudigonda. 2020. Branch and bound for piecewise linear neural network verification. *Journal of Machine Learning Research* 21, 42 (2020), 1–39.

[6] Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. 2017. Maximum Resilience of Artificial Neural Networks. In *Automated Technology for Verification and Analysis*, Deepak D'Souza and K. Narayan Kumar (Eds.). Springer, 251–268. https://doi.org/10.1007/978-3-319-68167-2_18

[7] Louis De Branges. 1959. The stone-weierstrass theorem. *Proc. Amer. Math. Soc.* 10, 5 (1959), 822–824.

[8] Souradeep Dutta, Xin Chen, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. 2019. Sherlock-a tool for verification of neural network feedback systems: demo abstract. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 262–263.

[9] Souradeep Dutta, Xin Chen, and Sriram Sankaranarayanan. 2019. Reachability analysis for neural feedback systems using regressive polynomial rule inference. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 157–168.

[10] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A Mann, and Pushmeet Kohli. 2018. A Dual Approach to Scalable Verification of Deep Networks.. In *Uncertainty in Artificial Intelligence*, Amir Globerson and Ricardo Silva (Eds.), Vol. 1. 550–559.

[11] Jiameng Fan, Chao Huang, Xin Chen, Wenchao Li, and Qi Zhu. 2020. Reachnn*: A tool for reachability analysis of neural-network controlled systems. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 537–542.

[12] Rida T Farouki. 2012. The Bernstein polynomial basis: A centennial retrospective. *Computer Aided Geometric Design* 29, 6 (2012), 379–419.

[13] Rida T Farouki and VT Rajan. 1988. Algorithms for polynomials in Bernstein form. *Computer Aided Geometric Design* 5, 1 (1988), 1–26.

[14] Mahyar Fazlyab, Alexander Robey, Hamed Hassani, Manfred Morari, and George Pappas. 2019. Efficient and accurate estimation of lipschitz constants for deep neural networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., 11423–11434.

[15] Matteo Fischetti and Jason Jo. 2018. Deep neural networks and mixed integer linear optimization. *Constraints* 23, 3 (2018), 296–309. https://doi.org/10.1007/s10601-018-9285-6

[16] Daniel J Fremont, Johnathan Chiu, Dragos D Margineantu, Denis Osipychev, and Sanjit A Seshia. 2020. Formal analysis and redesign of a neural network-based aircraft taxiing system with VerifAI. In *International Conference on Computer Aided Verification*. Springer, 122–134.

[17] Jürgen Garloff. 1985. Convergent bounds for the range of multivariate polynomials. In *International Symposium on Interval Mathematics*. Springer, 37–56.

[18] Jürgen Garloff and Andrew P Smith. 2007. Guaranteed affine lower bound functions for multivariate polynomials. In *PAMM: Proceedings in Applied Mathematics and Mechanics*, Vol. 7. Wiley Online Library, 1022905–1022906.

[19] 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 *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3–18. https://doi.org/10.1109/SP.2018.00058

[20] Patrick Henriksen and Alessio Lomuscio. 2021. DEEPSPLIT: An efficient splitting method for neural network verification via indirect effect analysis. In *IJCAI*. 2549–2555.

[21] Chao Huang, Jiameng Fan, Xin Chen, Wenchao Li, and Qi Zhu. 2022. Polar: A polynomial arithmetic framework for verifying neural-network controlled systems. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 414–430.

[22] Radoslav Ivanov, James Weimer, Rajeev Alur, George J Pappas, and Insup Lee. 2019. Verisig: verifying safety properties of hybrid systems with neural network controllers. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control (HSCC '19)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/3302504.3311806

[23] 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 *Computer Aided Verification* (Cham, 2017) *(Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, 97–117. https://doi.org/10.1007/978-3-319-63387-9_5

[24] Haitham Khedr, James Ferlez, and Yasser Shoukry. 2021. PEREGRiNN: Penalized-Relaxation Greedy Neural Network Verifier. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 287–300.

[25] Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, Mykel J Kochenderfer, et al. 2021. Algorithms for verifying deep neural networks. *Foundations and Trends® in Optimization* 4, 3-4 (2021), 244–404.

[26] Diana Andreea Popescu and Rogelio Tomas Garcia. 2016. Multivariate polynomial multiplication on GPU. *Procedia Computer Science* 80 (2016), 154–165.

[27] Shashwati Ray and PSV Nataraj. 2012. A Matrix Method for Efficient Computation of Bernstein Coefficients. *Reliab. Comput.* 17, 1 (2012), 40–71.

[28] Javier Sánchez-Reyes. 2003. Algebraic manipulation in the Bernstein form made simple via convolutions. *Computer-Aided Design* 35, 10 (2003), 959–967.

[29] Andrew Paul Smith. 2009. Fast construction of constant bound functions for sparse polynomials. *Journal of Global Optimization* 43, 2 (2009), 445–458.

[30] Volker Stahl. 1995. *Interval methods for bounding the range of polynomials and solving systems of nonlinear equations*. Ph. D. Dissertation. Johannes Kepler University Linz.

[31] Xiaowu Sun, Haitham Khedr, and Yasser Shoukry. 2019. Formal verification of neural network controlled autonomous systems. In *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*. 147–156.

[32] Xiaowu Sun and Yasser Shoukry. 2022. Neurosymbolic motion and task planning for linear temporal logic tasks. *arXiv preprint arXiv:2210.05180* (2022).

[33] Vincent Tjeng, Kai Y. Xiao, and Russ Tedrake. 2019. Evaluating Robustness of Neural Networks with Mixed Integer Programming. In *International Conference on Learning Representations*.

[34] Hoang-Dung Tran, Xiaodong Yang, Diego Manzanas Lopez, Patrick Musau, Luan Viet Nguyen, Weiming Xiang, Stanley Bak, and Taylor T. Johnson. 2020. NNV: The Neural Network Verification Tool for Deep Neural Networks and Learning-Enabled Cyber-Physical Systems. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, 3–17. https://doi.org/10.1007/978-3-030-53288-8_1

[35] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Efficient formal safety analysis of neural networks. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. 6367–6377.

[36] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. 2018. Formal security analysis of neural networks using symbolic intervals. In *Proceedings of the 27th USENIX Conference on Security Symposium (SEC'18)*. USENIX Association, 1599–1614. https://doi.org/10.5555/3277203.3277323

[37] 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. In *Advances in Neural Information Processing Systems*.

[38] Eric Wong and Zico Kolter. 2018. Provable defenses against adversarial examples via the convex outer adversarial polytope. In *International conference on machine learning*. 5286–5295.

[39] Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. 2018. Output reachable set estimation and verification for multilayer neural networks. *IEEE transactions on neural networks and learning systems* 29, 11 (2018), 5777–5783. https://doi.org/10.1109/TNNLS.2018.2808470

[40] Weiming Xiang, Hoang-Dung Tran, Joel A. Rosenfeld, and Taylor T. Johnson. 2018. Reachable Set Estimation and Safety Verification for Piecewise Linear Systems with Neural Network Controllers. In *2018 Annual American Control Conference (ACC)*. 1574–1579.

[41] Kaidi Xu, Huan Zhang, Shiqi Wang, Yihan Wang, Suman Jana, Xue Lin, and Cho-Jui Hsieh. 2021. Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers. In *ICLR*.