Neural Network Robustness Verification on GPUs

Christoph Müller, Gagandeep Singh, Markus Püschel, Martin Vechev
Department of Computer Science
ETH Zurich, Switzerland
{christoph.mueller,gsingh,pueschel,martin.vechev}@inf.ethz.com

***Abstract*—Certifying the robustness of neural networks against adversarial attacks is critical to their reliable adoption in real-world systems including autonomous driving and medical diagnosis. Unfortunately, state-of-the-art verifiers either do not scale to larger networks or are too imprecise to prove robustness, which limits their practical adoption.**

In this work, we introduce GPUPoly, a scalable verifier that can prove the robustness of significantly larger deep neural networks than possible with prior work. The key insight behind GPUPoly is the design of custom, sound polyhedra algorithms for neural network verification on a GPU. Our algorithms leverage the available GPU parallelism and the inherent sparsity of the underlying neural network verification task. GPUPoly scales to very large networks: for example, it can prove the robustness of a 1M neuron, 34-layer deep residual network in about 1 minute. We believe GPUPoly is a promising step towards the practical verification of large real-world networks.

I. Introduction

With the widespread adoption of deep neural networks in many real-world applications such as face recognition, autonomous driving, and medical diagnosis, it is critical to ensure that they behave reliably on a wide range of inputs. However, recent studies [1] have shown that trained networks are vulnerable to *adversarial examples*.

In Fig. 1 we illustrate the concept of an adversarial example. Here, a fully connected feedforward neural network (FFN) f has been trained to classify images from the popular CIFAR10 dataset [2]. The network classifies image I_0 from the test set correctly as a car. However, an adversary can increase the intensity of each pixel in I_0 by a small imperceptible amount to produce a new image I that still looks like a car to a human but that the network wrongly classifies as a bird.

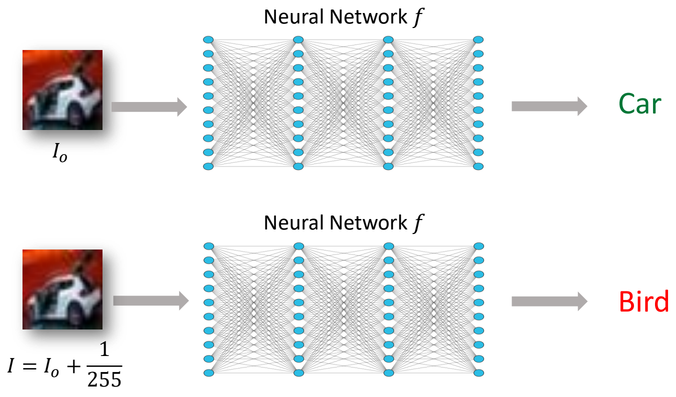


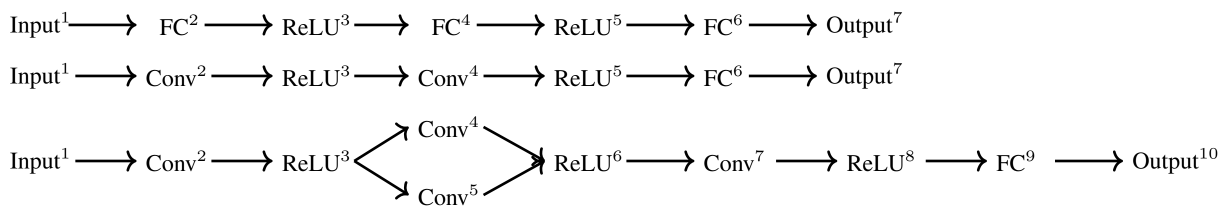
Fig. 1: Image I_0 (top) is classified correctly as a car by the neural network, however image I (bottom), obtained by increasing the intensity of each pixel by $1/255$ is wrongly classified as a bird.

Neural network robustness. Given the susceptibility to adversarial examples, there is growing interest in automated methods for certifying the robustness of neural networks, that is, proving that adversarial examples cannot occur within a specified adversarial region. More formally, robustness certification defines an adversarial region $L_\infty(I_0, \epsilon)$ as an L_∞ ball of radius $\epsilon \in \mathbb{R}$ around a correctly classified image I_0 in the test set. The goal then is to certify that the neural network classifies all images in $L_\infty(I_0, \epsilon)$ correctly. The set $L_\infty(I_0, \epsilon)$ contains an exponential (in the image size) number of images, which makes exhaustive enumeration infeasible. For example, the CIFAR10 image I_0 in Fig. 1 contains $3,072$ pixels. If we consider an L_∞ ball of radius $\epsilon = 1/255$ around I_0, then the number of images in $L_\infty(I_0, \epsilon)$ is 3^{3072}. We note here that the ϵ values used in our experiments (Section V) are significantly larger than $1/255$. Next, we discuss prior work on neural network verification and highlight the key challenges involved in the design of such verifiers.

Key challenge: Scalable and precise verification. Because enumeration is infeasible, neural network verifiers compute the output for all inputs in $L_\infty(I_0, \epsilon)$ symbolically. These verifiers can be broadly classified as either exact or inexact. Exact verifiers typically employ mixed-integer linear programming (MILP) [3], SMT solvers [4]–[6] and Lipschitz optimization [7]. They are computationally expensive and do not scale to networks of the size considered in our work.

To address this scalability issue, inexact verifiers compute an overapproximation of the network output corresponding to the inputs in $L_\infty(I_0, \epsilon)$. Due to overapproximation, there can be false positives, i.e., the verifier may fail to prove the network robust when it actually is. The inexact verifiers are typically based on abstract interpretation [8]–[11], duality [12], [13], linear approximations [14]–[20], and semidefinite relaxations [21]. There are also methods [22]–[24] that combine both exact and inexact approaches aiming to be more scalable than exact methods while improving the precision of inexact methods.

There is a tradeoff between the scalability and the degree of overapproximation of inexact verifiers. More precise inexact verifiers [8], [10], [11], [13]–[18], [21]–[23] scale to medium-sized networks ($\approx$ 100K neurons) but cannot handle the networks considered in our work ($\approx$ 1M neurons). On the other hand, more approximate verifiers [9], [19], [20] scale to larger



Input¹ → FC² → ReLU³ → FC⁴ → ReLU⁵ → FC⁶ → Output⁷
Input¹ → Conv² → ReLU³ → Conv⁴ → ReLU⁵ → FC⁶ → Output⁷
Input¹ → Conv² → ReLU³ → Conv⁴ / Conv⁵ → ReLU⁶ → Conv⁷ → ReLU⁸ → FC⁹ → Output¹⁰

Fig. 2: Sketches of considered neural network architectures.

networks but lose too much precision (that is, may induce too many false positives) which limits their applicability. Thus, a key challenge today is the design of neural network verifiers that scale to larger networks and maintain sufficient precision to provide desired network robustness guarantees.

Scalable and precise network verification on a GPU. In this work, we present our neural network verifier GPUPoly that leverages the parallel processing power of GPUs for precise and scalable verification. Designing such a verifier for GPUs is challenging for three reasons: (i) the verification algorithm has to offer the fine-grain data parallelism needed to benefit from the GPU's processing power, (ii) since GPU memory is much smaller than CPU memory, it has to be memory efficient, and (iii) it has to be sound for floating point arithmetic, i.e., it should capture all results possible under different rounding modes and orders of execution of floating point operations.

To address these challenges, we build our GPU verifier based on the recently introduced DeepPoly polyhedra approximations [11] and design custom algorithms for running it efficiently on a GPU. The DeepPoly approximation is among the more precise inexact methods and its underlying algorithms are highly parallelizable. It is possible to implement this approximation on a GPU using off-the-shelf libraries such as PyTorch [25] and Tensorflow [26] as in [18], but these cannot exploit the sparsity patterns produced by the DeepPoly analysis. Thus, the resulting implementations lack the performance and memory efficiency to scale to large networks considered in our work (Section V). Indeed, the existing scalable GPU-based verifiers [27], [28] using these libraries sacrifice significant precision for scalability.

In this work, we design new algorithms for running DeepPoly approximation fast, precise, and sound on a GPU while also ensuring memory efficiency by exploiting the sparsity inherent in the DeepPoly analysis. Further, we design new algorithms for handling the challenging residual layers which DeepPoly does not support. GPUPoly is sound w.r.t. floating-point arithmetic which is essential for verifier correctness [29].

Main contributions. Our main contributions are:

- Custom algorithms with explicit memory management to efficiently parallelize the state-of-the-art precise DeepPoly approximation on GPUs, enabling fast and precise verification of networks with up to $\approx$ 1M neurons.
- A novel approximation for handling residual layers that balances precision and scalability.
- A complete floating-point-sound CUDA implementation of our approach in a verifier called GPUPoly that handles fully-connected, convolutional, and residual networks.
- An experimental evaluation of GPUPoly demonstrating its effectiveness for handling large neural networks. Our results show that GPUPoly can prove the robustness of neural networks beyond the reach of prior work.

We next provide the necessary background on DeepPoly analysis and GPUs in Section II. We describe our core contributions for adapting DeepPoly analysis on the GPU in Section III and Section IV.

II. Background

We consider networks with fully-connected (FC), convolutional (Conv), and ReLU layers. Fig. 2 shows sketches of the considered neural network architectures. The fully-connected and convolutional networks consist of a sequence of layers with affine transformations and ReLUs. The residual networks additionally have branches containing convolutional layers. At the merging location of two branches the combined result is obtained by adding the branch outputs neuronwise. The networks used in our experiments have significantly more layers than shown in Fig. 2. We number the layers as shown in the superscripts in Fig. 2.

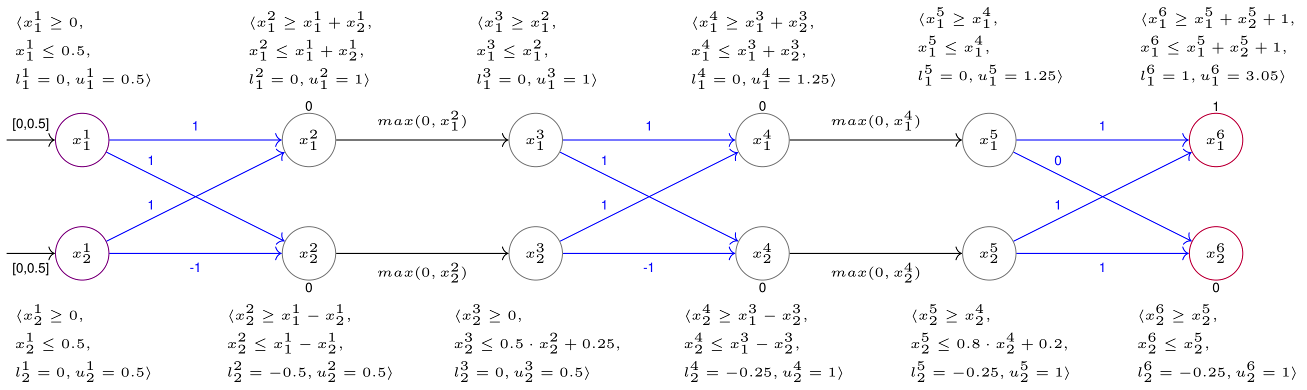
Next, we provide background on the neural network verification problem and the inner workings of the DeepPoly approximation [11]. We conclude with a brief overview of the parallelization mechanism offered by CUDA on GPUs.

A. DeepPoly approximation

We now introduce the commonly used L_∞-norm based robustness property and then show how DeepPoly approximations proves such a property on a toy fully-connected network.

L_∞-norm based robustness properties. Given an image I_0 that is correctly classified by the neural network, the adversarial region defined by $L_\infty(I_0, \epsilon)$, $\epsilon > 0$, includes all images I such that the difference between the intensity of each pixel in I and the corresponding pixel in I_0 is $\leq \epsilon$. The DeepPoly verifier tries to prove that the neural network classifies all images $I \in L_\infty(I_0, \epsilon)$ correctly. If it succeeds then all images are classified correctly. Otherwise, because of approximations, the robustness of the network is unknown.

Verification of a small fully-connected network. We use the word "neuron" interchangeably for the abstract node in a layer of the network and the concrete value of that neuron during an operation (for example 0.24). We use x_i^q to refer to the i-th neuron in layer q. Fig. 3 shows a fully-connected network with 3 fully-connected layers (each an



$\langle x_1^1 \geq 0,$
$x_1^1 \leq 0.5,$
$l_1^1 = 0, u_1^1 = 0.5\rangle$
$\langle x_1^2 \geq x_1^1 + x_2^1,$
$x_1^2 \leq x_1^1 + x_2^1,$
$l_1^2 = 0, u_1^2 = 1\rangle$
$\langle x_1^3 \geq x_1^2,$
$x_1^3 \leq x_1^2,$
$l_1^3 = 0, u_1^3 = 1\rangle$
$\langle x_1^4 \geq x_1^3 + x_2^3,$
$x_1^4 \leq x_1^3 + x_2^3,$
$l_1^4 = 0, u_1^4 = 1.25\rangle$
$\langle x_1^5 \geq x_1^4,$
$x_1^5 \leq x_1^4,$
$l_1^5 = 0, u_1^5 = 1.25\rangle$
$\langle x_1^6 \geq x_1^5 + x_2^5 + 1,$
$x_1^6 \leq x_1^5 + x_2^5 + 1,$
$l_1^6 = 1, u_1^6 = 3.05\rangle$
[0,0.5]
x_1^1
x_2^1
x_1^2
x_2^2
x_1^3
x_2^3
x_1^4
x_2^4
x_1^5
x_2^5
x_1^6
x_2^6
1
-1
0
$max(0, x_1^2)$
$max(0, x_2^2)$
$max(0, x_1^4)$
$max(0, x_2^4)$
$\langle x_2^1 \geq 0,$
$x_2^1 \leq 0.5,$
$l_2^1 = 0, u_2^1 = 0.5\rangle$
$\langle x_2^2 \geq x_1^1 - x_2^1,$
$x_2^2 \leq x_1^1 - x_2^1,$
$l_2^2 = -0.5, u_2^2 = 0.5\rangle$
$\langle x_2^3 \geq 0,$
$x_2^3 \leq 0.5 \cdot x_2^2 + 0.25,$
$l_2^3 = 0, u_2^3 = 0.5\rangle$
$\langle x_2^4 \geq x_1^3 - x_2^3,$
$x_2^4 \leq x_1^3 - x_2^3,$
$l_2^4 = -0.25, u_2^4 = 1\rangle$
$\langle x_2^5 \geq x_2^4,$
$x_2^5 \leq 0.8 \cdot x_2^4 + 0.2,$
$l_2^5 = -0.25, u_2^5 = 1\rangle$
$\langle x_2^6 \geq x_2^5,$
$x_2^6 \leq x_2^5,$
$l_2^6 = -0.25, u_2^6 = 1\rangle$

Fig. 3: Robustness verification of a simple fully connected network with DeepPoly.

affine transformation) and 2 ReLU layers with 2 neurons per layer. The weights for the fully-connected layers are shown on the blue edges and the biases are shown above and below neurons. The original input to the network is the 2-D point $(0.25, 0.25)$ with the corresponding network output $x_1^6 = 2$ and $x_2^6 = 0.5$ satisfying $x_1^6 > x_2^6$. Now, we consider the set of inputs where both input neurons x_1^1 and x_2^1 can take any value in the range $[0, 0.5]$. The verification problem is then to prove that for all such inputs, the network outputs satisfy $x_1^6 > x_2^6$.

The DeepPoly approximation associates four constraints with every neuron x_i^q: (i) lower and upper polyhedral constraints of the form $\sum_j a_j \cdot x_j^k + c \leq x_i^q$ and $x_i^q \leq \sum_j a'_j \cdot x_j^k + c'$ respectively where $1 \leq k < q$, and (ii) concrete lower and upper bounds $l_i^q \leq x_i^q \leq u_i^q$. Here, we have that $a_j, a'_j, c, c', l_i^q, u_i^q \in \mathbb{R}$. We refer to $\sum_j a_j \cdot x_j^k + c$ and $\sum_j a'_j \cdot x_j^k + c'$ as the lower and upper polyhedral expressions respectively. DeepPoly first sets constraints $0 \leq x_1^1, x_2^1 \leq 0.5$, $l_1^1 = l_2^1 = 0$, and $u_1^1 = u_2^1 = 0.5$ for x_1^1, x_2^1. The first fully-connected layer corresponds to the affine transformations $x_1^2 := x_1^1 + x_2^1$ and $x_2^2 := x_1^1 - x_2^1$ which adds:

$$x_1^1 + x_2^1 \leq x_1^2 \leq x_1^1 + x_2^1, \quad l_1^2 = 0, \quad u_1^2 = 1,$$

$$x_1^1 - x_2^1 \leq x_2^2 \leq x_1^1 - x_2^1, \quad l_2^2 = -0.5, \quad u_2^2 = 0.5,$$

Next, the ReLU layer is handled which corresponds to the assignments $x_1^3 := \max(0, x_1^2)$ and $x_2^3 := \max(0, x_2^2)$. The neuron x_1^2 takes only non-negative values as $l_1^2 = 0$ and thus the ReLU activation passes all values of x_1^2 to x_1^3 and hence the verifier adds $x_1^2 \leq x_1^3 \leq x_1^2, l_1^3 = 0, u_1^3 = 1$ for x_1^3. Since $l_2^2 < 0$ and $u_2^2 > 0$, the neuron x_2^2 can take both positive and negative values. As a result, ReLU sets $x_2^3 = 0$ when $x_2^2 < 0$ and $x_2^3 = x_2^2$ when $x_2^2 \geq 0$. Capturing both cases is expensive as one needs to consider both paths ($x_2^2 < 0$ and $x_2^2 \geq 0$). DeepPoly approximates all possible values of x_2^3 by adding:

$$0 \leq x_2^3 \leq 0.5 \cdot x_2^2 + 0.25, \quad l_2^3 = 0, \quad u_2^3 = 0.5.$$

Notice that the polyhedral expressions in the constraints added by DeepPoly for ReLU are sparse as they contain only the input of the ReLU. We note that handling ReLU activations is not a bottleneck for DeepPoly and thus we do not discuss it in further detail. We refer the reader to [11] for detailed discussion about handling ReLU activations with DeepPoly. The next fully-connected layer adds the constraints:

$$x_1^3 + x_2^3 \leq x_1^4 \leq x_1^3 + x_2^3, \quad x_1^3 - x_2^3 \leq x_2^4 \leq x_1^3 - x_2^3,$$

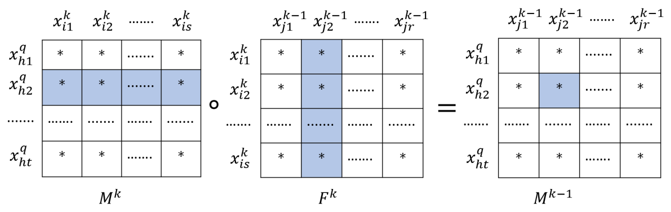
for x_1^4 and x_2^4, respectively.

Bottleneck backsubstitution. We note that x_1^4 and x_2^4 are inputs to the next ReLU layer. The precision of the DeepPoly ReLU approximations depends upon the tightness of the concrete bounds $l_1^4, u_1^4, l_2^4, u_2^4$. A straightforward substitution of the concrete bounds $l_1^3, u_1^3, l_1^3, u_2^3$ into the relational constraints for x_1^4 and x_2^4 results in imprecise values for $l_1^4, u_1^4, l_2^4, u_2^4$. This is because the substitution ignores the relational information from the polyhedral constraints.

DeepPoly employs *backsubstitution* for obtaining more precise concrete bounds for neurons x_i^q input to a ReLU layer. Backsubstitution iteratively computes new polyhedral bounds for each x_i^q by using the relational constraints of neurons in the previous layers. Each iteration starts with the polyhedral constraint for x_i^q expressed in terms of the neurons in layer k with $2 \leq k < q$. The backsubstitution substitutes polyhedral expressions of neurons in layer k defined in terms of neurons in the layer $k-1$ into the polyhedral constraint for x_i^q. The substituted expressions usually have common terms that cancel out, which, in turn, results in more precise bounds.

We next describe the computation of u_1^4 with backsubstitution, other bounds are computed similarly. We substitute the upper polyhedral expressions of the ReLU layer constraints $x_1^3 \leq x_1^2$ and $x_2^3 \leq 0.5 \cdot x_2^2 + 0.25$ for x_1^3, x_2^3 into the upper polyhedra constraint $x_1^4 \leq x_1^3 + x_2^3$ obtaining $x_1^4 \leq x_1^2 + 0.5 \cdot x_2^2 + 0.25$. Next we substitute the polyhedra expressions of the constraints from the affine layer $x_1^2 \leq x_1^1 + x_2^1$ and $x_2^2 \leq x_1^1 - x_2^1$ into $x_1^4 \leq x_1^2 + 0.5 \cdot x_2^2 + 0.25$ obtaining $1.5 \cdot x_1^1 + 0.5 \cdot x_2^1 + 0.25$. Notice that the terms containing x_2^1 when substituting expressions for x_1^2 and x_2^2 cancel out. We finally substitute $x_1^1, x_2^1 \leq 0.5$ into $x_1^4 \leq 1.5 \cdot x_1^1 + 0.5 \cdot x_2^1 + 0.25$ obtaining $u_1^4 = 1.25$.

Backsubstitution as matrix multiplication We consider an iteration of the backsubstitution to compute bounds for the neurons in layer q. The left matrix M^k in Fig. 4 encodes



x_{i1}^k x_{i2}^k .... x_{is}^k
x_{h1}^q x_{h2}^q .... x_{ht}^q
M^k
x_{j1}^{k-1} x_{j2}^{k-1} .... x_{jr}^{k-1}
x_{i1}^k x_{i2}^k .... x_{is}^k
F^k
M^{k-1}

Fig. 4: Backsubstitution in constraints for q-th layer neurons. We omit the constant term in the constraints for simplicity.

constraints for q-th layer neurons with polyhedral expressions defined over the neurons in layer $2 \leq k < q$. The center matrix F^k encodes constraints for k-th layer neurons defined over the neurons in layer $k-1$. We focus on the computation of the entry $(h2, j2)$ (shown in blue) in the result matrix M^{k-1}. The corresponding entries of M^k and F^k used for computing $(h2, j2)$ are also shown in blue. The entry $(h2, j2)$ encodes the coefficient for neuron $j2$ of layer $k-1$ in the constraint for x_{h2}^q. The substitution (as defined above) computes $(h2, j2)$ by multiplying each entry $(h2, i)$ in M^k with the entry $(i, j2)$ of F^k where $1 \leq i \leq s$. Each multiplication result represents a term involving x_{j2}^{k-1} obtained by substituting the expression for x_i^k in the constraint for x_{h2}^q. The results are then summed which causes cancellation. This computation can be seen as multiplying $h2$-th row of M^k with $j2$-th column of F^k and the overall computation of M^{k-1} thus involves multiplying M^k with F^k.

If layer k is a ReLU layer, then the F^k is a square diagonal matrix as the corresponding expressions contain only one neuron. Thus the backsubstitution through ReLU layers can be implemented as cheap matrix-vector multiplication. We note that this step is not a bottleneck. If layer k is an affine layer (fully-connected, convolutional, residual), then the polyhedral expressions in F^k corresponds to the weight matrix for transformation from layer $k-1$ to layer k. The backsubstitution in this case is more expensive and is the focus of our work. We design custom algorithms with memory management tailored to exploit the sparsity patterns observed when handling convolutional and residual layers. We note that while matrix multiplication can be easily parallelized on GPUs, the standard algorithms [18] are not memory and compute efficient for our task and run out of memory on medium-sized benchmarks ($\approx 100K$ neurons). Further, to ensure floating point soundness we perform all computations in interval arithmetic, which prevents the use of existing libraries. We will provide more details on this later.

Proving the robustness property The analysis proceeds and we obtain the constraints for all neurons in the network shown in Fig. 3. The DeepPoly verifier computes the lower bound for $x_1^6 - x_2^6$ using backsubstitution (recall that our goal was to prove that $x_1^6 - x_2^6 > 0$). This final computation results in the bound 1, which is sufficient to prove the property.

Asymptotic cost Consider a neural network with n affine layers and with each layer containing at most N neurons. The backsubstitution tasks for all neurons at an intermediate layer $q \leq n$ perform a matrix multiplication of worst-case asymptotic cost $\mathcal{O}(N^3)$ for all preceding affine layers (we ignore the quadratic cost of the ReLU layers) resulting in an overall cost $\mathcal{O}(q \cdot N^3)$. Because backsubstitution is performed for every layer of the network, the total asymptotic cost of the DeepPoly algorithm is $\mathcal{O}(n^2 \cdot N^3)$.

Since the backsubstitution through ReLU layers is cheap, we will ignore these for the remainder of this paper. Without loss of generality, we will present neural networks as just a sequence of affine layers.

B. *Parallelization in CUDA*

We briefly introduce key features of GPUs that we exploit in GPUPoly when redesigning DeepPoly for GPUs. We will use the CUDA C++ language extension. There are two different concepts of parallelism defined for CUDA: blocks and threads. CUDA blocks are similar to cores of a CPU and are independent compute units. Thus, different machine code can be executed on different blocks. A modern Nvidia graphics card contains hundreds of these blocks.

CUDA threads are a SIMD (single instruction, multiple data) mechanism allowing the execution of the same instruction sequence on multiple values of the same data type in parallel. A block can execute 32 SIMD threads (called a *warp*) in parallel and its is possible to allocate up to 32 warps to each block to enable latency hiding. For optimal performance, the number of threads per block should be divisible by 32. Additionally, one needs to balance between providing the GPU with a sufficient number of warps for context switching while also not overusing block-wide resources, e.g., the number of registers allocated per thread. The ideal number of threads per block is often 128 or 256 in practice.

III. Robustness Verification on GPUs: Concepts and Algorithms

To utilize the highly parallel GPU hardware effectively, an algorithm must provide a sufficient number of compute tasks that can be distributed over all blocks and threads, and its memory requirements must not exceed the comparatively small GPU memory (compared to CPU memory). In this section, we introduce concepts and techniques for designing and implementing an efficient DeepPoly-based GPU algorithm for verifying deep neural networks.

A. *Network DAG and dependence set*

For networks with convolutional layers, each row of the matrices M^k, F^k, M^{k-1} (Fig. 4) usually contains only a few non-zero entries. We exploit this sparsity for making DeepPoly backsubstitution more compute and memory efficient. We now introduce our concept of *dependence set* that we use for computing a sparse representation of the matrices M^k, F^k, M^{k-1}. Let neuron x_i^q be the i-th neuron of layer q. The dependence set of x_i^q contains the neurons in preceding layers $k < q$ affecting the value of x_i^q. Only the columns corresponding to these neurons have non-zero entries in the row for x_i^q and are

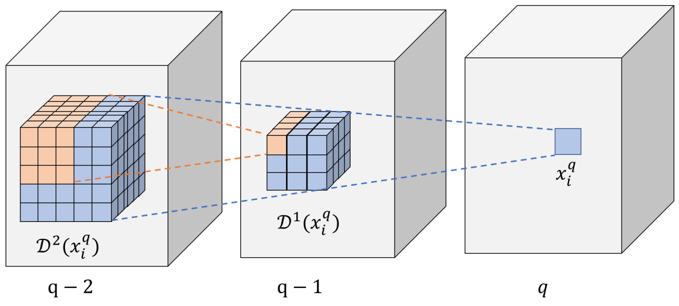


Fig. 5: $\mathcal{D}^1(x_i^q)$ and $\mathcal{D}^2(x_i^q)$ for a neuron x_i^q with two preceding convolutional layers.

therefore sufficient for defining polyhedral expressions in the corresponding matrices. We first provide an example of the dependence sets and then give their formal definition.

Example 1. We consider convolutional layers with filters of size 3×3 and a stride of 1 in both height (h) and width (w) direction. Fig. 5 shows the subset of the neurons in layer $q-1$ influencing the value of neuron x_i^q. Note that the dimensions $3 \times 3 \times 4$ of this subset directly correspond to the filter sizes. The neurons in layer $q-2$ that influences x_i^q contains all neurons that influence the value of neurons in the computed subset for layer $q-1$. The size of this subset is $5 \times 5 \times 6$.

To formally describe dependence sets, we first define the *network DAG* (directed, acyclic graph; an example is Fig. 3) associated with a neural network. As before we denote with x_i^q, the neuron i in layer q. In a network DAG $(\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ is the set of all neurons. Two neurons are connected by a directed edge $(x_j^k, x_i^q) \in \mathcal{E}$ if x_j^k is directly needed to compute x_i^q. More formally, $(x_j^k, x_i^q) \in \mathcal{E}$ if layer k is an immediate predecessor (contains inputs) of layer q and layer q is fully-connected or one of the following applies, depending on the type of layer q (in parenthesis):

- (ReLU) $j = i$,
- (Convolutional) x_j^k is in the window for computing x_i^q,

Note that for fully-connected and convolutional architectures (top two rows in Fig. 2), we have that $k = q-1$ while for a residual architecture (last row in Fig. 2), layer q can have multiple immediate predecessors $k < q$.

The *first dependence set* of a neuron x_i^q collects all its immediate predecessors in the network DAG:

$$\mathcal{D}^1(x_i^q) = \{x_j^k | (x_j^k, x_i^q) \in \mathcal{E}\}, \tag{1}$$

Similarly for a set of neurons $\mathcal{X}^q$ in the same layer q:

$$\mathcal{D}^1(X^q) = \bigcup_{x_i^q \in X^q} \mathcal{D}^1(x_i^q) \tag{2}$$

We extend this concept recursively. The *m-th dependence set*, $m \geq 2$, of x_i^q is the first dependence set of $\mathcal{D}^{m-1}(x_i^q)$:

$$\mathcal{D}^m(x_i^q) = \mathcal{D}^1(\mathcal{D}^{m-1}(x_i^q)) \tag{3}$$

and the definition of $\mathcal{D}^m(\mathcal{X}^q)$ is analogous. We also define the *zeroth dependence set* $\mathcal{D}^0(x_i^q) = \{x_i^q\}$.

Finally, we define the *full dependence set* of a neuron x_i^q to be the union of the dependence sets of all levels:

$$\mathcal{D}^{\text{full}}(x_i^q) = \bigcup_{m \in \{1 \ldots t\}} \mathcal{D}^m(x_i^q) \tag{4}$$

The full dependence set of a neuron x_i^q is the set of all neurons in the preceding layers with a (computational) path to x_i^q.

Example 2. In the previous example of Fig. 3, $\mathcal{D}^{\text{full}}(x_1^6) = \{x_1^1, x_2^1, x_1^2, x_2^2, x_1^3, x_2^3, x_1^4, x_2^4, x_1^5, x_2^5\}$. Neuron x_1^5 is included even though the coefficient of x_1^5 in the corresponding affine transformation is 0. The reason is that our definition of the network DAG does not consider coefficients of the affine transformations for defining edges.

During DeepPoly analysis, all neurons appearing in the polyhedral expressions when backsubstituting recursively on the polyhedral constraints for x_i^q are available in the different subsets $\mathcal{D}^{q-k}(x_i^q)$ with $k = q-1, \ldots, 0$ of $\mathcal{D}^{\text{full}}(x_i^q)$. The expression in the initial constraint contains neurons from $\mathcal{D}^1(x_i^q)$ and we call it step 1 of the backsubstitution. Step 2 substitutes for each neuron in $\mathcal{D}^1(x_i^q)$, the polyhedral expression defined over the neurons in $\mathcal{D}^2(x_i^q)$ resulting in the constraint for x_i^q to be defined over the neurons in $\mathcal{D}^2(x_i^q)$. Continuing similarly, we see that $\mathcal{D}^{q-k}(x_i^q)$ contains neurons appearing in the expressions after $q-k$ steps. In Section IV, we exploit the structure of the convolutional layers to derive recursive expressions for computing $\mathcal{D}^{q-k}(x_i^q)$ that enable fast computation with negligible overhead. Next we discuss the backsubstitution for the different network types in greater detail.

B. Fully-connected networks

In Section II-A we represented the backsubstitution step of the DeepPoly analysis starting at layer q as a sequence of matrix multiplications through all preceding layers $k < q$. For fully-connected networks, the dependence sets $\mathcal{D}^{q-k}(x_i^q)$ for x_i^q are dense and include all neurons in layer k. Thus we use dense matrix multiplication parallelized over the blocks of the GPU and SIMD parallelized over the threads. The coefficients in the polyhedral expressions are accordingly stored as dense matrices. However, as we explain later, the computation is done in interval arithmetic to ensure floating point soundness (Section IV-D). Thus we could not use existing libraries and our implementation is from scratch.

C. Convolutional networks

Naively using the same algorithm as for fully-connected layers for the backsubstitution starting at a convolutional layer q does not utilize the GPU efficiently. First, the majority of the computations are not needed since the filters in the convolutional layers are usually sparse and thus the filter matrix F^k of Fig. 4 consists of mostly zeroes. Additionally, it is not memory efficient since many coefficients in matrices M^k and M^{k-1} of Fig. 4 will be zero during backsubstitution.

Key ideas. The neurons in the polyhedral expression for x_i^q after $k < q$ backsubstitution steps (starting from q) are in the dependence set $\mathcal{D}^{q-k}(x_i^q)$. For an efficient implementation on GPUs, we use these sets in two ways:

- Using the set we can flatten the needed coefficients into a dense matrix to perform the backsubstitution again efficiently as matrix multiplication.
- All the backsubstitution tasks for a given layer (one per neuron) are independent of each other. The sizes of the dependence sets enable us to predict the memory needed per task. This, in turn, enables us to determine the number of tasks that can be allocated in the GPU memory without exceeding its limits.

Size prediction and management As in Fig. 5, the first few dependence sets of x_i^q are usually very small compared to the size of the layers. If the number of layers is small, there is no memory problem. For example, the total memory footprint of the analysis of a 6-layer 230K neuron convolutional network (e.g., `ConvLarge` in Table I later) fits into 16 GB of RAM.

For deeper networks the k-th dependence set will ultimately grow to the size of the full layer. For example, assuming a realistic layer size of 100K neurons, each backsubstitution task then requires at least $16 \cdot 100$KB (16 since one double is read and one written) per neuron after k steps. Doing so for all neurons in layer q then amounts to $16 \cdot 100 \cdot 100$KB = 160GB. The complete algorithm has four times the memory requirement since both, the upper and lower, polyhedral constraints are backsubstituted and the floating point soundness property discussed in Section IV-D requires double the memory yet again. The resulting 640GB exceed the memory of any GPU.

Our solution in this case is to predict the maximal memory footprint μ of one backsubstitution and then allocate t backsubstitutions in the memory and distribute them over the blocks of the GPU, where t is the largest number with $64 \cdot t \cdot \mu \leq \lambda$ where λ is the GPU memory size in bytes.

For each backsubstitution step of a single task, there are two sets of coefficients in memory corresponding to the neurons in $\mathcal{D}^{q-k}(x_i^q)$ of the current layer k which we read (a row of the matrix M^k in Fig. 4) and the ones in $\mathcal{D}^{q-k+1}(x_i^q)$ of the previous layer $k-1$ which we compute (a row of the matrix M^{k-1} in Fig. 4). The largest memory footprint that occurs during a single backsubstitution task is obtained by finding the two neighboring layers k and $k-1$ with the biggest added size of their dependence sets:

$$\mu = \max_{k<q}(|\mathcal{D}^{q-k}(x_i^q)| + |\mathcal{D}^{q-k+1}(x_i^q)|) \tag{5}$$

We compute μ in advance, and use it to determine t, thus efficiently using both, available parallelism and GPU memory. We discuss the details of the actual implementation in Section IV.

D. *Residual networks*

To simplify the exposition of our ideas and without loss of generality, we assume that the width of the residual network is two, i.e., a layer has no more than two immediate predecessors or successors. We further assume that the two branches of a residual block contain one convolutional layer each. An example of such an architecture is in Fig. 6 which shows the residual block of the bottom network of Fig. 2 with all ReLU layers removed for simplicity. The point-wise nature of ReLU layers allows us to ignore them with respect to the dependence set. The two branches of the residual block contain one convolutional layer each.

$$\begin{tikzcd}
 & & \text{Conv}^4 \arrow[dr, "a"] & & \\
\cdots \arrow[r] & \text{Conv}^2 \arrow[ur, "a"] \arrow[dr, "b"'] & & \text{Conv}^7 \arrow[r] & \cdots \\
 & & \text{Conv}^5 \arrow[ur, "b"'] & &
\end{tikzcd}$$

Fig. 6: Simplified residual architecture without ReLU layers.

We call the two branches a and b. In Fig. 6 branch a and b contains $Conv^4$ and $Conv^5$ layer respectively. Naturally, the layer at the head of the residual block ($Conv^2$ in Fig. 6) has two successors while the one at exit ($Conv^7$ in Fig. 6) has two predecessors.

The first dependence set of a neuron x_i^q in a layer at the exit of a residual block (e.g. $Conv^7$ in Fig. 6) contains neurons from both branches (subsets of layers $Conv^4$ and $Conv^5$ in Fig. 6). The resulting dependence set can be written as:

$$\mathcal{D}^1(x_i^q) = \mathcal{D}^{(1,a)}(x_i^q) \cup \mathcal{D}^{(1,b)}(x_i^q), \tag{6}$$

where $\mathcal{D}^{(1,a)}(x_i^q)$ and $\mathcal{D}^{(1,b)}(x_i^q)$ are the first dependence sets of x_i^q with respect to the branches a and b respectively.

The second dependence set of x_i^q is a subset of the layer at the head of the residual block ($Conv^2$ in Fig. 6) and can then be expressed as the union of the first dependence sets of $\mathcal{D}^{(1,a)}(x_i^q)$ and $\mathcal{D}^{(1,b)}(x_i^q)$:

$$\begin{aligned}
\mathcal{D}^2(x_i^q) &= \mathcal{D}^1\left(\mathcal{D}^{(1,a)}(x_i^q) \cup \mathcal{D}^{(1,b)}(x_i^q)\right) && (7)\\
&= \mathcal{D}^1\left(\mathcal{D}^{(1,a)}(x_i^q)\right) \cup \mathcal{D}^1\left(\mathcal{D}^{(1,b)}(x_i^q)\right) && (8)
\end{aligned}$$

For a residual block, we backsubstitute through both branches independently and then join the independent backsubstitutions at the head of the block by adding the coefficients of the expressions neuron by neuron.

For size prediction, we modify (5) for handling residual blocks. The backsubstitution through the two branches is performed in series and then the results are added. This requires keeping a copy of the values of neurons in the layer at the exit of the residual block ($Conv^7$ in Fig. 6) while backsubstituting through the a branch first.

Similarly we need to keep the results of the a branch (subset of $Conv^4$) while performing the backsubstitution through the b branch. Thus compared to (5) we have to keep neuron values of three dependence sets in memory at the same time. The following equation examplifies this for a backsubstitution starting from the exit layer of the residual block ($Conv^7$ in our example), where we retain a copy of the coefficients in the exit layer while performing the backsubstitution through branch a (from $Conv^4$ to $Conv^2$ in our example):

$$\mu_{r,a} = |\mathcal{D}^{(1,a)}(x_i^q)| + |\mathcal{D}^{(2,a)}(x_i^q)| + |\mathcal{D}^{0}(x_i^q)|. \quad (9)$$

In order to derive the μ for the size prediction (similar to equation (5)), the maximum over all posible sizes over all layers then has to be taken, which includes $\mu_{r,a}$ and also includes $\mu_{r,b}$ associated with branch b.

IV. GPUPoly

In this section we explain in greater detail the algorithms and our implementation of the GPUPoly verifier. We first explain how to compute the size and the elements of the dependence set $\mathcal{D}^{q-k}(x_i^q)$ of x_i^q in layer $k < q$. Then we show our parallel algorithm for the backsubstitution in DeepPoly analysis. We then contrast this algorithm to the parallel algorithm that was used for the CPU version of DeepPoly [11] and to the well-known backpropagation algorithm [30] used for training DNNs. Finally, we explain how we maintain floating point soundness using interval arithmetic. As before, we ignore the ReLU layers.

We handle the backsubstitution for the fully-connected networks by parallelizing dense matrix-matrix multiplication on a GPU as mentioned in Section III. We thus focus on the most challenging convolutional and residual networks.

We will use the following notation for convolutional layers. We represent the neurons in a convolutional layer k as a three-dimensional block, shown as gray in Fig. 5. A neuron x_i^k has a unique three-dimensional index $i = (w, h, d)$, where w and h are the horizontal and vertical indices and d is the filter index. To simplify the exposition and without loss of generality, we assume that all parameters of convolutional layers have the same value in horizontal h and vertical w directions, such as filter sizes $f_w^k = f_h^k = f^k$, strides $s_w^k = s_h^k = s^k$ or the padding $p_w^k = p_h^k = p^k$. We further assume the following values for all layers: $f^k = 3$, $s^k = 1$ and $p^k = 0$ (zero padding). The remaining cases can be handled analogously.

A. Computing dependence sets for convolutional networks

In Fig. 5 we have seen examples of the first and second dependence set of a neuron in a convolutional layer. Now we derive the general equations for the size and location (or offset) of the elements of $(q-k)$-th dependence set $\mathcal{D}^{q-k}(x_i^q)$ as a subset of the neurons in a convolutional layer $k < q$.

$\mathcal{D}^{q-k}(x_i^q)$ is a cuboid and we compute the size of the set $\mathcal{D}^{q-k}(x_i^q)$ along the height, width and depth direction separately. We note that a dependence set is always dense in the depth direction for convolutional layers, so the size of $\mathcal{D}^{q-k}(x_i^q)$ in the depth direction is equal to the number of filters of layer k. Because of the symmetry assumption for the w and h direction of convolutional layers, the width and the height of $\mathcal{D}^{q-k}(x_i^q)$ are equal, and we denote it with W^{q-k}. The following recurrence computes W^{q-k+1} given W^{q-k}:

$$\begin{aligned} W^0 &= 1, \\ W^{q-k+1} &= (W^{q-k} - 1) \cdot s^k + f^k, \quad k = q \dots 1. \quad (10) \end{aligned}$$

For example, in Fig. 5 we obtain $W^1 = (W^0 - 1) \cdot 1 + 3 = 3$ for the first dependence set and $W^2 = (W^1 - 1) \cdot 1 + 3 = 5$ for the second dependence set. The overall size of $\mathcal{D}^{q-k}$ is:

$$|\mathcal{D}^{q-k}(x_i^q)| = W^{q-k} \cdot W^{q-k} \cdot C^k, \quad k = q-1 \dots 0. \quad (11)$$

where C^k is the number of channels of layer k. We note that (11) can be plugged into (5) and (9) for size prediction. We compute the neuron indexes next.

The indexes depend on the location of x_i^q in layer q. We only need to derive the position in the width and the height direction as all the corresponding channels of layer k are in $\mathcal{D}^{q-k}(x_i^q)$. Let the position of x_i^q in layer q be $i = (w^q, h^q, d^q)$. Then the w- and h-positions of the neuron with the smallest coordinates in $\mathcal{D}^{q-k}(x_i^q)$ are:

$$w^{q-k} = S^{q-k} \cdot w^q, \quad (12)$$

$$h^{q-k} = S^{q-k} \cdot h^q, \quad k = q-1 \dots 0. \quad (13)$$

where we introduced the quantity S^{q-k}, which we call *accumulated stride*. It is computed using the following recurrence:

$$\begin{aligned} S^0 &= 1, & (14) \\ S^{q-k+1} &= s^k \cdot S^{q-k}, \quad k = q \dots 1. & (15) \end{aligned}$$

The extension to padding other than zero padding is analogous.

Using the the size and the position of $\mathcal{D}^{q-k}(x_i^q)$ in layer k, we can now compute, recursively, for $k = q-1 \dots 0$ the associated coefficients of the neurons in $\mathcal{D}^{q-k}(x_i^q)$ occurring in the backsubstituted expression. We store these in a dense vector called $M^{q-k}(x_i^q)$. In each step these get modified by the backsubstitution:

$$M^1(x_i^q) = (a_1, a_2, \dots, a_{|\mathcal{D}^1(x_i^q)|}),$$
$$M^{q-k+1}(x_i^q) = \text{GBC}(M^{q-k}(x_i^q), \mathcal{D}^{q-k}(x_i^q), \mathcal{D}^{q-k+1}(x_i^q), F^k).$$

$M^1(x_i^q)$ contains the coefficients corresponding to the neurons in the first dependence set in the initial polyhedra expression before the backsubstitution starts bounding x_i^q. We ignore the constant in the constraint for simplifying our exposition. GBC (GPUPoly Backsubstitution for Convolution) is our algorithm for handling a single step of a backsubstitution task in convolutional networks, shown in Algorithm 1 and explained below. F^k is the constraint matrix between the neurons in layer k and $k-1$ generated during DeepPoly analysis (Fig. 4). As in Section II, F^k corresponds to the filter for convolutional layers. We next explain GBC in greater detail.

B. Our algorithm for convolutional networks

In this section we describe our algorithm for verifying convolutional networks on GPUs. We recall that the back-substitution at layer q is done as one task per neuron, and all these tasks are independent. We first use our size prediction algorithm from Section III-C to determine the the number of tasks t to run in parallel. Next, our algorithm distributes these independent tasks over the GPU blocks. Since convolutional layers usually contain tens of thousands of neurons and a modern GPU consists of a few hundred blocks we can be sure that all blocks are populated.

Next, we describe our mapping of a single backsubstitution task for x_i^q on the SIMD thread-parallelized structure of a single block. We focus on the backsubstitution step from layer k to layer $k-1$ (computation for the other steps is similar). This step can be seen as a map from $M^{q-k}(x_i^q)$ to $M^{q-k+1}(x_i^q)$. For convenience, we assume that both vectors are reshaped as per their corresponding 3-D dependent sets.

Algorithm 1 Single backsubstitution step for x_i^q through convolutional layers

```
1: function GBC($M^{q-k}$, $\mathcal{D}^{q-k}$,$\mathcal{D}^{q-k+1}$,$F^k$)
2:     // Compute the polyhedra expression $M^{q-k+1}$ bounding $x_i^q$
3:     $M^{q-k}, M^{q-k+1} \leftarrow$ submatrix for layer $k$ and $k-1$
4:     $\mathcal{D}^{q-k} \leftarrow (q-k)$-th dependence set of $x_i^q$
5:     $(W^{q-k}, W^{q-k}, C^k) \leftarrow$ dimensions of $\mathcal{D}^{q-k}$
6:     $\mathcal{D}^{q-k+1} \leftarrow (q-k+1)$-th dependence set of $x_i^q$
7:     $(W^{q-k+1}, W^{q-k+1}, C^{k-1}) \leftarrow$ dimensions of $\mathcal{D}^{q-k+1}$
8:     $(f^k, f^k) \leftarrow$ filter size in $w$ and $h$ directions of layer $k$
9:     $(s^k, s^k) \leftarrow$ strides in $w$ and $h$ directions for layer $k$
10:    $F^k \leftarrow$ 4-D filter weight tensor of layer $k$
11:    // Thread-parallelization dimension is consecutive in memory
12:    $(C^k, f^k, f^k, C^{k-1}) \leftarrow$ dimensions of $F^k$

13:    // Serial loop
14:    for $(w, h) \in (0 : W^{q-k}, 0 : W^{q-k})$ do
15:        // Zero, one or two of these loops can be thread parallelized
16:        // depending on if $C^{k-1}$ is big enough. If any of these is
17:        // parallelized, then thread-sync is needed after going
18:        // through 0..$C^k$ loop

19:        for $(f, g) \in (0 : f^k, 0 : f^k)$ do
20:            $a = w \cdot s^k + f$
21:            $b = h \cdot s^k + g$

22:            // Thread-parallelized loop
23:            for $c \in (0 : C^{k-1})$ do
24:                $M^{q-k+1}[a][b][c] = 0$
25:                // Serial loop, reduction in every thread

26:                for $d \in (0 : C^k)$ do
27:                    // Coalesced read from $F^k$
28:                    // Coalesced write to $\mathcal{D}^{q-k+1}$
29:                    $M^{q-k+1}[a][b][c] \mathrel{+}= M^{q-k}[w][h][d] \cdot F^k[d][f][g][c]$
```

Algorithm 1 shows our algorithm for a single step ($q-k \rightarrow q-k+1$) of a backsubstitution task through convolutional layers. To avoid notational clutter, we write $\mathcal{D}^{q-k}(x_i^q), M^{q-k}(x_i^q)$ as simply $\mathcal{D}^{q-k}, M^{q-k}$ respectively for the rest of this section. We use Matlab-style notation $(i : j) = \{i, \ldots, j\}$

The two outermost loops of the algorithm are the iteration on the w- and h-dimension of $\mathcal{D}^{q-k}$ at line 14. For a given (w, h), we obtain a $1 \times 1 \times C^k$ subset of $\mathcal{D}^{q-k}$. Since the convolutional layers are fully-connected in the d-direction, each neuron in this subset has the same first dependence set, which is a $f^k \times f^k \times C^{k-1}$ block in layer $k-1$. Fig. 5 shows such an example where the first dependence set of every neuron in the orange $1 \times 1 \times 4$ set in layer $q-1$ is the same $3 \times 3 \times 6$ set of neurons in layer $q-2$.

We target thread-parallelization of the backsubstitution from the $1 \times 1 \times C^k$ subset in $\mathcal{D}^{q-k}$ to the $f^k \times f^k \times C^{k-1}$ subset of $\mathcal{D}^{q-k+1}$. Since the convolutional filter F^k has four dimensions, we have four loops, the two loops over the filter sizes $0 : f^k$ in w- and h-direction and the loops $(0 : C^k), (0 : C^{k-1})$ over the d directions of layer k and $k-1$ respectively (lines 19–26).

We have multiple candidates for arranging the four loops. To maximize the benefits from thread-parallelization, we require coalescent reads and writes. Since we sum over C^k dimension at line 29, this loop is not suited for thread parallelism. The two $(0 : f^k)$ loops at line 19 can be thread-parallelized, however, this requires a synchronization before moving to the next $1 \times 1 \times C^k$ block in layer k, otherwise there are race conditions.

The loop best suited for thread-parallelization is the loop $(0 : C^{k-1})$ at line 23 and thus we parallelize over it. If $C^{k-1} < 128$, then our parallelization of this loop does not fully utilize the full power of thread parallelization. Thus our algorithm additionally thread-parallelizes over either one or both of the $(0 : f^k)$ loops at line 19 so that the total number of threads is at least 128. This enables us to gain more parallelism which improves performance at the cost of synchronization overhead.

Mixed layer backsubstitution. We discussed the backsubstitution algorithm in Section III-B where the starting layer q and all preceding layers are fully-connected. The specialized algorithm where all layers are convolutional was explained above. Running GPUPoly on the convolutional network in Fig. 2 (middle) requires handling the backsubstitution starting from the fully-connected layer 6 with steps through the two convolutional layers 2 and 4.

Let k_0 be the first fully-connected layer following $k_0 - 1$ convolutional layers in a convolutional neural network and let all layers $q \geq k_0$ be fully-connected. For a backsubstitution starting at layer $q \geq k_0$, we apply the algorithm for fully-connected layers introduced in Section III-B for the first $q - k_0$ layers encountered during backsubstitution and apply Algorithm 1 when backsubstituting through the remaining convolutional layers. We note here that for all neurons x_i^q in layer q the dependence set $\mathcal{D}^{q-k}$ corresponding to the convolutional layer k contains all neurons of layer k.

Comparison to the parallel CPU implementation. Since the number of CPU cores on modern machines is at least an order of magnitude less than the number of blocks on a GPU, the parallelized CPU implementation [11] of DeepPoly processes smaller number of backsubstitution tasks than GPUPoly. Thus the CPU implementation does not require explicit memory management as the chunk size is already small (CPU memory is also usually larger than GPU) but this also makes the implementation slow. The CPU implementation exploits sparsity in the polyhedral expressions when performing backsubstitution from the convolutional layers by storing the polyhedral expressions with a sparse representation, storing neuron indexes and the corresponding coefficient. This representation does not exploit the structure of the convolutional layers and is not suitable for thread parallelization. In contrast, we exploit structured sparsity in convolutional layers via dependent sets which allows us to create smaller dense submatrices that are suitable for thread parallelization.

Comparison to standard backpropagation We now compare the backsubstitution used in DeepPoly to the standard backpropagation algorithm [30] used for training neural networks. The latter is fundamentally different from our backsubstitution because it computes a scalar loss function and propagates it back to update the network weights while the DeepPoly backsubstitution propagates constraints backwards. Further, the backpropagation is usually only performed starting from the last layer which typically contains fewer neurons than the intermediate layers. In contrast, the DeepPoly backsubstitution is performed starting from all layers in the network. Thus, we also have to backsubstitute starting from intermediate convolutional or residual layers which typically contain orders of magnitude more neurons than the last layer which makes balancing the compute and the memory efficiency of our backsubstitution on GPUs more challenging (Section III). Overall, based on the above factors, the DeepPoly backsubstitution is mathematically different, computationally more expensive, and more memory-demanding than the backpropagation.

C. Dependence sets and algorithm for residual networks

In Section III-D we discussed that backsubstitution through a residual block involves independent backsubstitutions through the two branches followed by neuronwise addition of the two resulting coefficient vectors. We first note as per (11), for a purely convolutional neural network the dependence sets are cuboidal subsets of the convolutional layers. This is not necessarily the case for residual blocks. We discuss the computation of the dependence sets when backsubstituting through a residual block next.

For the neuron x_i^q at the exit of a residual block (e.g. $Conv^7$ in Fig. 6), the two parts of the first dependence set $\mathcal{D}^1(x_i^q) = \mathcal{D}^{(1,a)}(x_i^q) \cup \mathcal{D}^{(1,b)}(x_i^q)$ in the branches a and b are cuboidal just as in the purely convolutional case. However, the second dependence set $\mathcal{D}^2(x_i^q) = \mathcal{D}^{(2,a)}(x_i^q) \cup \mathcal{D}^{(2,b)}(x_i^q)$ (see also (8)), which is a subset of layer at the head of the block ($Conv^2$ in Fig. 6), is guaranteed to be cuboidal only if one of the two following equations holds:

$$\mathcal{D}^{(2,a)}(x_i^q) \quad \subseteq \mathcal{D}^{(2,b)}(x_i^q) \text{ or,} \tag{16}$$

$$\mathcal{D}^{(2,b)}(x_i^q) \quad \subseteq \mathcal{D}^{(2,a)}(x_i^q), \tag{17}$$

that is if one is a subset of the other or vice versa. We note that the above condition is true for the networks used in our experiments. For the general case, one can compute a cuboid containing $\mathcal{D}^{(2,a)}(x_i^q) \cup \mathcal{D}^{(2,b)}(x_i^q)$ and continue the backsubstitution with the resulting cuboid.

We use Algorithm 1 for the two independent backsubstitutions through the two branches, however, the merging of the two resulting dense vectors $M^{(2,a)}(x_i^q)$ and $M^{(2,b)}(x_i^q)$ requires particular attention. Assuming that $\mathcal{D}^2(x_i^q)$ is cuboid, we use equations (11) and (15) to calculate the relative offset of the vectors with respect to each other: $(w^{(2,a)} - w^{(2,b)}, h^{(2,a)} - h^{(2,b)})$. The two dense coefficient vectors $M^{(2,a)}(x_i^q)$ and $M^{(2,b)}(x_i^q)$ representing the two resulting expressions are then combined by adding the values neuronwise.

TABLE I: Neural networks used in our experiments.

Dataset	Model	Type	#Neurons	#Layers	Training
MNIST	6×500	FC	3,010	6	Normal
	ConvBig	Conv	48K	6	DiffAI
	ConvSuper	Conv	88K	6	Normal
CIFAR10	6×500	FC	3,010	6	Normal
	ConvBig	Conv	62K	6	DiffAI
	ConvLarge	Conv	230K	6	DiffAI
	ResNetTiny	Res	311K	12	PGD
	ResNet18	Res	558K	18	PGD
	ResNetTiny	Res	311K	12	DiffAI
	ResNet18	Res	558K	18	DiffAI
	SkipNet18	Res	558K	18	DiffAI
	ResNet34	Res	967K	34	DiffAI

Since a backsubstitution is performed starting from every convolutional layer in the network, the last case we discuss is a backsubstitution starting from a convolutional layer which is part of a residual block, for example $Conv^4$ of branch a in Fig. 6. In this case we backsubstitute until reaching the head of the block. Since there is no contribution from the b branch, for a neuron x_i^q in this convolutional layer we simply get $\mathcal{D}^{(1,a)}(x_i^q) = \mathcal{D}^1(x_i^q)$ for the first dependence set and $M^{(1,a)}(x_i^q) = M^1(x_i^q)$ for the corresponding dense vector. The backsubstitution continues as in the purely convolutional case based on Algorithm 1.

D. Floating point soundness

An essential property and major challenge is to ensure floating point soundness of our analysis. This means that the results of our analysis will always contain all results possible under different rounding modes and under different orders of computations. To guarantee this property, the coefficients of all polyhedra constraints are stored as intervals and computations such as matrix-matrix multiplication are performed on intervals, as done, e.g., in [31], which also prevents the use of existing CUDA libraries. Interval arithmetic doubles the memory requirement and more than doubles the cost of computations. For example, multiplying two intervals requires four multiplications and determining the minimum and maximum of the results. Further, for soundness, this minimum and maximum have to be rounded differently, namely towards $-\infty$ and $+\infty$, respectively. In CUDA this is done using the appropriate rounding intrinsics. We note that existing libraries such as Tensorflow [26] and Pytorch [25] do not allow setting the rounding mode on a per-operation basis, thus they cannot be used for ensuring floating point soundness.

V. Experimental Evaluation

We now demonstrate the effectiveness of GPUPoly for the verification of large neural networks in terms of both precision and performance. GPUPoly is implemented in C, supports 64-bit double precision, and uses the CUDA library for GPU support. We compare the effectiveness of GPUPoly against two state-of-the-art verifiers: the CPU parallelized version of DeepPoly [11] and HBox [9]. We note that DeepPoly has the same precision as GPUPoly, however GPUPoly is up to 170x

TABLE II: *Experimental results for 500 images on medium size neural networks: HBox [9] and DeepPoly [11] vs. GPUPoly*

Dataset	Model	#Neurons	Training	ϵ	#Candidates	Average runtime [s]			#Verified		
						HBox	DeepPoly	GPUPoly	HBox	DeepPoly	GPUPoly
MNIST	6×500	3,010	Normal	8/255	493	0.001	8.3	0.08	0	334	334
	ConvBig	48K	DiffAI	3/10	487	0.05	12	0.35	439	441	441
	ConvSuper	88K	Normal	8/255	495	0.05	271	1.6	0	428	428
CIFAR10	6×500	3,010	Normal	1/500	282	0.06	15	0.11	1	219	219
	ConvBig	62K	DiffAI	8/255	226	0.09	38	0.5	121	127	127
	ConvLarge	230K	DiffAI	8/255	232	0.27	309	2.5	131	138	138

TABLE III: *Experimental results for 500 images on large neural networks: HBox [9] vs. GPUPoly*

Dataset	Model	#Neurons	Training	ϵ	#Candidates	Average runtime [s]		#Verified	
						HBox	GPUPoly	HBox	GPUPoly
CIFAR10	ResNetTiny	311K	PGD	1/500	391	0.29	20	0	322
	ResNet18	558K	PGD	1/500	419	6.8	1021	0	324
	ResNetTiny	311K	DiffAI	8/255	184	0.29	5.0	118	127
	SkipNet18	558K	DiffAI	8/255	168	6.0	41	130	140
	ResNet18	558K	DiffAI	8/255	193	6.3	26	129	139
	ResNet34	967K	DiffAI	8/255	174	16	59	103	114

faster than DeepPoly on the networks that are small enough for DeepPoly and can scale to larger networks. HBox is implemented for GPUs using PyTorch and uses the Zonotope approximations [32] for neural network verification. HBox is more precise than interval based verifiers [20] and more scalable than other GPU based verifiers [13], [18], [27]. We note that [18] is a GPU implementation of DeepPoly based on Pytorch but it runs out of memory on our smallest convolutional benchmark. As a result we do not compare against it. Thus HBox is the most precise existing verifier that can scale to the large neural networks used in our experiments. We note that unlike GPUPoly and DeepPoly, HBox is not floating point sound thus its verification results can be incorrect due to floating point errors.

Our experimental results show that GPUPoly improves over the state of the art by providing the most precise and scalable verification results on all our benchmarks. We believe that the extra scalability and precision of GPUPoly can also benefit state-of-the-art robust training methods [18], [33] in the future as they depend on approximate verifiers for training.

Neural networks. We used 12 deep neural networks in our experiments. These networks were trained on the popular MNIST [34] and CIFAR10 [2] datasets as shown in Table I. Out of these, 3 are MNIST-based and 9 are CIFAR10-based. Table I specifies the network architecture, the number of neurons, the number of layers and the training method for each network. There are 2 fully connected (FC), 4 convolutional (Conv) and 6 residual (Res) architectures in Table I. The largest neural network used in our experiments is ResNet34: it has 34 layers and contains ≈1M neurons.

Regarding training, (i) 7 of our networks were trained using DiffAI [9], [28], which performs *provably robust adversarial training*, (ii) 2 of our CIFAR10 networks were trained using Projected Gradient Descent (PGD) [35], [36], which amounts to *experimentally robust adversarial training*, and (iii) the remaining 3 networks were trained in a standard manner. Both methods, (i) and (ii), aim to increase the robustness of the resulting neural network.

Experimental setup. All our experiments for HBox and GPUPoly were performed on a 2.2 GHZ 10 core Intel Xeon Silver 4114 CPU with 512GB of main memory. The sizes of the L1, L2, and L3 caches were 1KB, 32KB, and 14MB respectively. The GPU on this machine was an Nvidia Tesla V100 GPU with 16GB of memory. The PyTorch version used for running HBox was 0.4.1 and the CUDA version for GPUPoly was 10.2. The experiments for the (prior) CPU version of DeepPoly were performed on a 2.6 GHz 14 core Intel Xeon CPU E5-2690 with 512GB of memory. The sizes of the L1, L2, and L3 caches were 32KB, 256KB, and 35MB respectively. Note that the CPU for running the CPU version of DeepPoly was faster than the one for HBox and GPUPoly.

Benchmarks. For each MNIST and CIFAR10 neural network, we selected the first 500 images from the respective test set and filtered out the images that were not classified correctly. We call the correctly classified images from the test set *candidate* images. The number of candidate images for each neural network are shown in Table II and Table III.

For each candidate image I_0, we defined the $L_\infty(I_0, \epsilon)$ based adversarial region by selecting challenging values of ϵ that are commonly used for testing the precision and scalability of neural network verifiers. The ϵ values used for defining $L_\infty(I_0, \epsilon)$ for each neural network are shown in Table II and Table III. We used larger values of ϵ for testing DiffAI trained networks since these networks can be proven robust for larger values compared to the PGD and normally trained networks. However, DiffAI-trained networks can suffer from a substantial drop in test accuracy (see #Candidates in Table III). Furthermore, these networks are also easier to verify and thus even imprecise verifiers like HBox are able to verify large number of robustness properties on these networks. We also note that the ϵ values for MNIST based networks are larger than for CIFAR10 based networks for the same reason.

A. Results on medium sized networks

Table II compares the effectiveness of GPUPoly vs. DeepPoly and HBox on the robustness verification of our medium sized fully connected and convolutional networks. All of these networks contain 6 layers. The largest network among these contains 230K neurons. We compare the precision of the three verifiers by comparing the number of $L_\infty(I_0, \epsilon)$ regions where the network is proven to be robust by the verifier. We also compare the performance by reporting the average runtime in seconds of the verifiers per $L_\infty(I_0, \epsilon)$ region.

Precision. Table II shows that both DeepPoly and GPUPoly verify the network to be robust on more images than HBox. The difference in precision is higher on the normally trained networks than on the DiffAI trained ones. In total, on the 3 normally trained networks, both DeepPoly and GPUPoly verify $981/1270$ properties whereas HBox verifies only 1.

The reason why HBox verifies more properties on DiffAI trained networks is because all inexact verifiers only sacrifice precision for scalability on neurons that are input to a ReLU and can take both positive and negative values during analysis (e.g., neurons x_2^2 and x_2^4 in Fig. 3). The number of such neurons for DiffAI trained networks is lower than for normally trained networks. On the 3 DiffAI trained networks, both DeepPoly and GPUPoly verify 13 properties more than HBox.

Performance. HBox is the fastest verifier and has an average runtime of < 0.3 seconds on all benchmarks in Table II. The speed of HBox comes at the cost of imprecision. The CPU implementation of DeepPoly is much more precise than HBox but is orders of magnitude slower than HBox and takes > 5 minutes on average for proving one robustness property on the largest ConvLarge CIFAR10 network with 230K neurons in Table II. GPUPoly in contrast achieves the same precision as DeepPoly but is between 35x and 170x faster on all networks.

B. Results on large networks

Table III compares GPUPoly against HBox for verifying our large CIFAR10 residual networks that are out of range for the CPU version of DeepPoly. For example, DeepPoly takes > 2 hours (when we timed out) per image on the ResNet18 network trained with PGD. One can observe that the PGD trained networks classify more images correctly than DiffAI, however they are less provably robust and we use smaller ϵ values for defining the robustness properties on these.

Precision. For the DiffAI trained networks, we noticed that backsubstitution up to the first layer does not bring significant precision gains over backsubstituting only until the end of the first residual layer encountered during backsubstitution. Thus for DiffAI trained networks, we only backsubstitute until the end of the first encountered residual layer for improving performance. We note that GPUPoly still proves 40 more robustness properties on these networks than HBox with the highest increase in precision (11 more properties verified) over HBox taking place on our largest neural networks ResNet34 which is 34 layers deep and contains $\approx$ 1M neurons.

For the PGD trained networks, we backsubstitute until the input layer otherwise significant precision is lost. HBox does not prove any of the 810 properties on the two PGD trained networks in Table III while GPUPoly proves 646.

Performance. It can be seen in Table III that HBox is faster than GPUPoly due to the much simpler polyhedra abstraction (zonotopes). GPUPoly runs slower when backsubstituting up to the input layer on PGD trained networks. On all DiffAI trained networks, limited backsubstitution allows GPUPoly to verify robustness properties in <60 seconds.

Our work advances the state-of-the-art by precisely verifying significantly larger CIFAR10 networks, with up to 1M neurons, than possible with prior work. Based on our results, we believe that our work is a step in the direction towards scaling to even larger models such as those for ImageNet which cannot be handled currently. Another limitation of our work is that we cannot handle GNNs or GANs.

VI. Conclusion

We presented a scalable neural network verifier, called GPUPoly, for verifying the robustness of various types of deep neural networks on GPUs. GPUPoly leverages GPU parallelization, sparsity in convolutional and residual networks, and uses an adaptation mechanism that matches the memory footprint to the available GPU memory. As a result, GPUPoly scales precise, polyhedra-based verification to large neural networks with $\approx$ 1M neurons. Our experimental results show that GPUPoly verifies robustness properties of deep neural networks beyond the reach of existing state-of-the-art verifiers.

References

[1] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[3] V. Tjeng, K. Y. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," in *International Conference on Learning Representations, (ICLR)*, 2019.

[4] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, 2017.

[5] R. Ehlers, "Formal verification of piece-wise linear feed-forward neural networks," in *Automated Technology for Verification and Analysis (ATVA)*, 2017.

[6] R. Bunel, I. Turkaslan, P. H. Torr, P. Kohli, and M. P. Kumar, "A unified view of piecewise linear neural network verification," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 4795–4804.

[7] W. Ruan, X. Huang, and M. Kwiatkowska, "Reachability analysis of deep neural networks with provable guarantees," in *Proc. International Joint Conference on Artificial Intelligence, (IJCAI)*, 2018.

[8] 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 Symposium on Security and Privacy (SP)*, vol. 00, 2018, pp. 948–963.

[9] M. Mirman, T. Gehr, and M. Vechev, "Differentiable abstract interpretation for provably robust neural networks," in *Proc. International Conference on Machine Learning (ICML)*, 2018, pp. 3575–3583.

[10] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 10 825–10 836.

[11] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 41:1–41:30, 2019.

[12] K. Dvijotham, R. Stanforth, S. Gowal, T. Mann, and P. Kohli, "A dual approach to scalable verification of deep networks," in *Proc. Uncertainty in Artificial Intelligence (UAI)*, 2018, pp. 162–171.

[13] E. Wong and Z. Kolter, "Provable defenses against adversarial examples via the convex outer adversarial polytope," in *Proc. International Conference on Machine Learning (ICML)*, 2018, pp. 5286–5295.

[14] L. Weng, H. Zhang, H. Chen, Z. Song, C.-J. Hsieh, L. Daniel, D. Boning, and I. Dhillon, "Towards fast computation of certified robustness for ReLU networks," in *Proc. International Conference on Machine Learning (ICML)*, 2018, pp. 5276–5285.

[15] H. Zhang, T.-W. Weng, P.-Y. Chen, C.-J. Hsieh, and L. Daniel, "Efficient neural network robustness certification with general activation functions," in *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[16] A. Boopathy, T.-W. Weng, P.-Y. Chen, S. Liu, and L. Daniel, "Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks," in *AAAI*, Jan 2019.

[17] H. Salman, G. Yang, H. Zhang, C. Hsieh, and P. Zhang, "A convex relaxation barrier to tight robustness verification of neural networks," *CoRR*, vol. abs/1902.08722, 2019.

[18] H. Zhang, H. Chen, C. Xiao, S. Gowal, R. Stanforth, B. Li, D. Boning, and C.-J. Hsieh, "Towards stable and efficient training of verifiably robust neural networks," in *International Conference on Learning Representations (ICLR)*, 2020.

[19] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana, "Formal security analysis of neural networks using symbolic intervals," in *Proc. USENIX Conference on Security Symposium*, ser. SEC'18, pp. 1599–1614.

[20] S. Gowal, K. Dvijotham, R. Stanforth, R. Bunel, C. Qin, J. Uesato, R. Arandjelovic, T. A. Mann, and P. Kohli, "On the effectiveness of interval bound propagation for training verifiably robust models," *CoRR*, vol. abs/1810.12715, 2018.

[21] A. Raghunathan, J. Steinhardt, and P. S. Liang, "Semidefinite relaxations for certifying robustness to adversarial examples," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018, pp. 10 877–10 887.

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

[23] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "Boosting robustness certification of neural networks," in *International Conference on Learning Representations (ICLR)*, 2019.

[24] G. Singh, R. Ganvir, M. Püschel, and M. Vechev, "Beyond the single neuron convex barrier for neural network certification," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[25] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[26] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: http://tensorflow.org/

[27] E. Wong, F. R. Schmidt, J. H. Metzen, and J. Z. Kolter, "Scaling provable adversarial defenses," in *Proc. Neural Information Processing Systems (NeurIPS)*, 2018, pp. 8410–8419.

[28] M. Mirman, G. Singh, and M. T. Vechev, "A provable defense for deep residual networks," *CoRR*, vol. abs/1903.12519, 2019.

[29] K. Jia and M. Rinard, "Exploiting verified neural networks via floating point numerical error," 2020.

[30] F. Grund, "Rall, louis b., automatic differentiation: Techniques and applications. lecture notes in computer science 120," *ZAMM - Journal of Applied Mathematics and Mechanics*, vol. 62, no. 7, 1982.

[31] K. Ozaki, T. Ogita, S. M. Rump, and S. Oishi, "Fast algorithms for floating-point interval matrix multiplication," *Journal of Computational and Applied Mathematics*, vol. 236, no. 7, pp. 1795 – 1814, 2012.

[32] K. Ghorbal, E. Goubault, and S. Putot, "The zonotope abstract domain taylor1+," in *Proc. Computer Aided Verification (CAV)*, 2009, pp. 627–633.

[33] M. Balunovic and M. Vechev, "Adversarial training and provable defenses: Bridging the gap," in *International Conference on Learning Representations (ICLR)*, 2020.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proc. of the IEEE*, 1998, pp. 2278–2324.

[35] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *Proc. International Conference on Learning Representations (ICLR)*, 2018.

[36] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," in *Proc. Computer Vision and Pattern Recognition (CVPR)*, 2018.