SCALING POLYHEDRAL NEURAL NETWORK VERIFICATION ON GPUS

Christoph Müller [*1] **François Serre** [*1] **Gagandeep Singh** [2] **Markus Püschel** [1] **Martin Vechev** [1]

ABSTRACT

Certifying the robustness of neural networks against adversarial attacks is essential to their reliable adoption in safety-critical systems such as autonomous driving and medical diagnosis. Unfortunately, state-of-the-art verifiers either do not scale to bigger networks or are too imprecise to prove robustness, limiting their practical adoption. In this work, we introduce GPUPoly, a scalable verifier that can prove the robustness of significantly larger deep neural networks than previously possible. The key technical 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 inherent sparsity of the underlying verification task. GPUPoly scales to large networks: for example, it can prove the robustness of a 1M neuron, 34-layer deep residual network in ≈ 34.5 ms. We believe GPUPoly is a promising step towards practical verification of real-world neural networks.

1 INTRODUCTION

With the widespread adoption of deep neural networks in several 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 (Szegedy et al., 2013) have shown that deep networks are vulnerable to *adversarial examples*, illustrated in Fig. 1. Here, a neural network classifies an image I^0 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 but the network incorrectly classifies it as a bird.

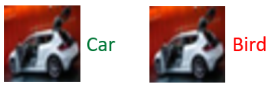


Figure 1. Image I^0 is classified correctly as a car by the neural network, while image I, obtained by increasing the intensity of each pixel in I^0 by $1/255$ is wrongly classified as a bird.

Neural network robustness. Given this susceptibility to adversarial examples, recent years have seen increased interest in automated methods that can certify robustness of neural networks, that is, to prove that adversarial examples cannot occur within a specified adversarial region (Katz et al., 2017; Ehlers, 2017; Wong & Kolter, 2018; Gehr et al., 2018). A typical example of an adversarial region would be the L_∞ ball of radius $\epsilon \in \mathbb{R}^+$ around an image I^0 (Carlini & Wagner, 2017). The goal of certification then is to prove that all images in this region are classified correctly by the network (i.e., to the same label as I^0). Note that the adversarial region usually contains an exponential (in image size) number of images, which makes exhaustive enumeration infeasible. For example, image I^0 in Fig. 1 contains $3,072$ pixels. If we consider a radius of $\epsilon = 1/255$ around I^0, then the number of images in the adversarial set $L_\infty(I^0, \epsilon)$ is 3^{3072} (in our experiments we consider significantly larger ϵ values).

Key challenge: scalable and precise verification. Because concrete enumeration is infeasible, neural network verifiers compute the output for all inputs in the adversarial region symbolically. These verifiers can be broadly classified as either exact or inexact. Exact verifiers typically employ mixed-integer linear programming (MILP) (Tjeng et al., 2019), SMT solvers (Katz et al., 2017; Ehlers, 2017; Bunel et al., 2018; Katz et al., 2019) and Lipschitz optimization (Ruan et al., 2018). They are computationally expensive and do not scale to the network sizes considered in our work. To address this scalability issue, inexact verifiers compute an over-approximation of the network output. Due to this approximation, a verifier may fail to prove the network robust when it actually is. Inexact verifiers are typically based on abstract interpretation (Gehr et al., 2018; Mirman et al., 2018; Singh et al., 2018; 2019b), duality (Dvijotham et al., 2018; Wong & Kolter, 2018), linear approximations (Weng et al., 2018; Zhang et al., 2018; Boopathy et al., 2019; Salman et al., 2019; Zhang et al.,

[*]Equal contribution [1]Department of Computer Science, ETH Zurich, Switzerland [2]VMware Research and Department of Computer Science, UIUC, USA. Correspondence to: Christoph Müller <christoph.mueller@inf.ethz.ch>, Francois Serre <serref@inf.ethz.ch>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021.

2020; Wang et al.; Gowal et al., 2018; Xu et al., 2020; Tran et al., 2020), and semi definite relaxations (Raghunathan et al., 2018; Dathathri et al., 2020). There are also methods (Wang et al., 2018; Singh et al., 2019c;a; Tjandraatmadja et al., 2020) 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 trade off between scalability and the degree of over-approximation of inexact verifiers. More precise, inexact verifiers (Wong & Kolter, 2018; Gehr et al., 2018; Singh et al., 2018; 2019b; Weng et al., 2018; Zhang et al., 2018; Boopathy et al., 2019; Salman et al., 2019; Raghunathan et al., 2018; Wang et al., 2018; Singh et al., 2019c; Tran et al., 2020) scale to medium-sized networks ($\approx$ 100K neurons) or verify weaker robustness properties (e.g. brightness (Pei et al., 2017)) but cannot handle the networks and properties (e.g. L_∞-norm) that our work can ($\approx$ 1M neurons). On the other hand, more approximate verifiers (Mirman et al., 2018; Wang et al.; Gowal et al., 2018; Zhang et al., 2020; Xu et al., 2020) scale to bigger networks but lose too much precision and fail to prove robustness, which limits their applicability. Thus, a key challenge is to design neural network verifiers that scale to large networks yet maintain the precision necessary to certify meaningful robustness guarantees.

Scalable, precise and sound verification on a GPU. In this work, we present GPUPoly, a new neural network verifier that addresses the above challenge via algorithms that leverage the processing power of GPUs. Concretely, GPUPoly: (i) introduces a method that enables the fine-grain data parallelism needed to benefit from GPUs, (ii) is memory efficient and can fit into GPU memory (which is much smaller than that of a CPU), and (iii) is sound for floating point arithmetic, capturing all results possible under different rounding modes and orders of execution of floating point operations, thus handling associativity correctly (important concern, as recent verifiers which are unsound for floating-point have been shown vulnerable to such attacks (Jia & Rinard, 2020; Zombori et al., 2021)).

GPUPoly is based on the state-of-the-art DeepPoly relaxation (Singh et al., 2019b) equipped with new, custom algorithms which exploit the underlying sparsity, and use a novel stopping criteria that can decrease runtime without compromising accuracy.

We note that it is possible to implement DeepPoly on a GPU using off-the-shelf libraries such as PyTorch (Paszke et al., 2017) and Tensorflow (Abadi et al., 2015) as in (Zhang et al., 2020; Xu et al., 2020). Unfortunately, these frameworks cannot exploit the sparsity patterns produced by DeepPoly, resulting in implementations that lack the performance and memory efficiency needed for handling the large networks considered in our work.

Main contributions. Our main contributions are:

- New algorithms to efficiently parallelize the state-of-the-art DeepPoly relaxation on GPUs, enabling fast and precise verification of networks with up to $\approx$ 1M neurons.
- A complete floating-point-sound CUDA implementation in a verifier called GPUPoly that handles fully-connected, convolutional, and residual networks. Our code is available as part of the ERAN framework at `https://github.com/eth-sri/eran`.
- An experimental evaluation of GPUPoly demonstrating its effectiveness in proving the robustness of neural networks beyond the reach of prior work.

We note that while we use GPUPoly for proving robustness against intensity perturbations in this work, GPUPoly is more general and can be used to certify other properties including safety (Katz et al., 2017), fairness (Ruoss et al., 2020), and robustness against geometric (Balunovic et al., 2019; Ruoss et al., 2021), contextual (Paterson et al., 2021), and generative (Mirman et al., 2020) perturbations.

2 Background and Notation

We now introduce the necessary background on both neural network robustness and the DeepPoly relaxation.

Classification network. For simplicity, the networks we consider here are built from a composition of two kinds of layers: the *affine* and the *ReLU* layer. We use the word *neuron* for the abstract node in such a layer, and we denote with x_i^ℓ the i^{th} neuron in the ℓ^{th} layer x^ℓ. The affine layers such as fully-connected, convolutional, and residual layers perform an affine mapping $x^\ell = A \cdot x^{\ell-1} + b$, where $A = (a_{i,j})$ is a matrix and $b = (b_i)$ is a vector. The ReLU layer trims negative values element wise: $x_i^\ell = \max(x_i^{\ell-1}, 0)$. A given input image I is assigned to the input layer x^0, and evaluated successively through the different layers. The neuron index in the final layer with the highest value yields the inferred category.

L_∞-norm based robustness properties. Given an image I^0 correctly classified by the network, and a number $\epsilon > 0$, the *adversarial region* $L_\infty(I^0, \epsilon)$ is the set of images I for which each pixel i differs by at most ϵ from the corresponding one in I_0: $||I_i - I_i^0||_\infty \leq \epsilon$. The objective of a verifier is to prove that all images in this region classify correctly.

DeepPoly analysis. The DeepPoly (Singh et al., 2019b) relaxation associates four bounds with every neuron x_i^ℓ: (i) lower and upper polyhedral bounds of the form $\sum_j a_{i,j} \cdot x_j^k + c \leq x_i^\ell$ and $x_i^\ell \leq \sum_j a'_{i,j} \cdot x_j^k + c'$, respectively, where $0 \leq k < \ell$, and (ii) interval bounds $l_i^\ell \leq x_i^\ell \leq u_i^\ell$, where

$a_{i,j}, a'_{i,j}, c, c', l_i^\ell, u_i^\ell \in \mathbb{R}$. We refer to $\sum_j a_{i,j} \cdot x_j^k + c$ and $\sum_j a'_{i,j} \cdot x_j^k + c'$ as the lower and upper polyhedral expressions, respectively. The polyhedral bounds for each x_i^ℓ are obtained by modeling the effects of affine transformation and ReLU as follows:

- The affine transformation $x_i^\ell = \sum_j a_{i,j}^{\ell-1} \cdot x_j^{\ell-1} + b_i$ adds the bounds $\sum_j a_{i,j}^{\ell-1} \cdot x_i^{\ell-1} + b_i \leq x_i^\ell \leq \sum_j a_{i,j}^{\ell-1} \cdot x_i^{\ell-1} + b_i$. Thus DeepPoly handles affine layers exactly.
- $x_i^\ell = \text{ReLU}(x_i^{\ell-1})$ adds the bounds $\alpha_i^\ell \cdot x_i^{\ell-1} + \beta_i^\ell \leq x_i^\ell \leq \gamma_i^\ell \cdot x_i^{\ell-1} + \delta_i^\ell$ where the constants $\alpha_i^\ell, \beta_i^\ell, \gamma_i^\ell, \delta_i^\ell$ are determined by the concrete bounds $l_i^{\ell-1} \leq x_i^{\ell-1} \leq u_i^{\ell-1}$. If $l_i^{\ell-1} > 0$ or $u_i^{\ell-1} \leq 0$, then the DeepPoly analysis is exact, otherwise it over-approximates.

The tightness of the concrete bounds of the neurons from the affine layers that are input to ReLU layers affects the precision of the DeepPoly analysis as the bounds determine whether the ReLU is handled exactly and otherwise affect the level of imprecision in case its effect is over approximated. DeepPoly computes tight concrete bounds for each x_i^ℓ in an affine layer by maximizing and minimizing its value with respect to the set of polyhedra and interval bounds over the neurons in all previous layers already computed by DeepPoly. DeepPoly solves both these linear programs approximately (for scalability reasons) using a greedy algorithm called *backsubstitution* (not to be confused with back-propagation) which is the main bottleneck of the analysis and therefore the focus of our work.

Bottleneck Backsubstitution. The backsubstitution algorithm for computing an upper bound u_i^ℓ (the lower bound is computed analogously) for neuron x_i^ℓ in an affine layer starts with the upper polyhedral bound $x_i^\ell \leq \sum_j a_{i,j}^{\ell-1} \cdot x^{\ell-1} + b_i$ added by the affine transformation. It then substitutes the concrete upper or lower bounds for each $x^{\ell-1}$ depending on the sign of the coefficient $a_{i,j}$ obtaining a candidate upper bound. Next, it substitutes for each $x^{\ell-1}$ the corresponding upper or lower polyhedral bound (again depending on the sign of $a_{i,j}$) defined over the neurons in layer $\ell - 2$. This yields a new polyhedral bound for x_i^ℓ now defined over the neurons in layer $\ell - 2$. It again uses concrete bounds for the neurons in $x^{\ell-2}$ to compute another candidate bound.

The algorithm repeats this step until it reaches the input layer. The result is the smallest candidate among the bounds computed at each step. Since backsubstitution only involves reading data from the previous layers, it can be executed in parallel for all neurons in a given layer ℓ, which is ideal for GPU parallelization. For simplicity, we will focus on the most expensive step of backsubstitution for the remainder of this paper: computing new polyhedral bounds for x_i^ℓ when the polyhedral bounds for the neurons x_j^k with $k < \ell$ to be substituted are the result of an affine transformation in fully-connected, convolutional, or residual layers. Without loss of generality, we therefore ignore the ReLU layers and consider neural networks as just a sequence of affine layers.

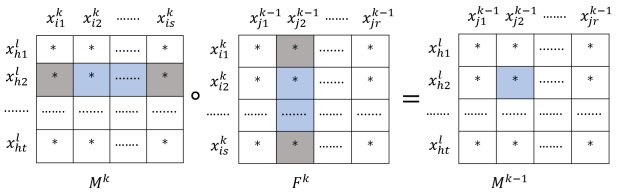


Figure 2. Backsubstitution in polyhedra bounds for ℓ-th layer neurons. We omit the constant term in the bounds for simplicity.

Backsubstitution as matrix multiplication. The left matrix M^k in Fig. 2 encodes bounds for ℓ-th layer neurons with polyhedral expressions defined over the neurons in layer $1 \leq k < \ell$. 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 subset of their $h2$-th row and $j2$-th column respectively (also shown in blue). The entry $(h2, j2)$ encodes the coefficient for neuron $j2$ of layer $k - 1$ in the constraint for x_{h2}^ℓ. The substitution (as defined above) computes $(h2, j2)$ by multiplying blue each entry $(h2, i)$ in M^k with the blue 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}^ℓ. The results are then summed which causes cancellation. This computation can be seen as multiplying the $h2$-th row of M^k with the $j2$-th column of F^k and the overall computation thus is a matrix multiplication $M^{k-1} = M^k \cdot F^k$.

We note that for fully-connected layers, all entries in the $h2$-th row and $j2$-th column are needed for computing $(h2, j2)$, while the convolutional and residual layers require a smaller subset. Identifying this subset is key to achieving the compute and memory efficiency required for obtaining a precise and scalable analysis. We design custom algorithms 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 (Zhang et al., 2020; Xu et al., 2020) are not memory and compute efficient for our task and run out of memory on medium-sized benchmarks ($\approx 170K$ neurons). Further, to ensure floating point soundness we perform all computations in interval arithmetic, which prevents the use of existing libraries.

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 $\ell \leq n$ perform a matrix multiplication in $\mathcal{O}(N^3)$ for all preceding affine layers (the ReLU layers have quadratic cost) resulting in an overall cost of $\mathcal{O}(\ell \cdot N^3)$. Because backsubstitution is performed for every layer of the network, the DeepPoly algorithm requires $\mathcal{O}(n^2 \cdot N^3)$ operations.

3 Robustness Verification on GPUs: Concepts and Algorithms

In this section, we introduce two key concepts that we exploit to design and implement an efficient DeepPoly-based GPU algorithm for verifying deep neural networks. The notion of *dependence set* allows us to harness the sparsity of convolutional and residual layers to speedup backsubstitutions, while an *early termination criterion* allows us to skip computations that would not improve results.

3.1 Dependence set

The core concept for exploiting sparsity in convolutional layers in our algorithms is the dependence set. Before defining it formally, we illustrate it on an example of backsubstitution through two convolutional layers (backsubstitution through fully-connected layers can be implemented as a dense matrix-matrix multiplication as explained in Section 2). We denote the neuron i in a convolutional layer ℓ as $x_i^\ell = x_{w,h,d}^\ell$, where w, h, d are its indices in the width, height, and depth dimensions, respectively. The rows of the matrix M^k ($1 \leq k < \ell$) depicted in Fig. 2 are the intermediate results of ht many independent backsubstitutions, one for each neuron in layer ℓ. We show one such single-neuron backsubstitution in Fig. 3 for neuron $x_{1,3,1}^\ell$. In our example, layer ℓ has size $3 \times 3 \times 2$, whereas the previous layer $\ell - 1$ has size $5 \times 5 \times 2$. The convolution operation with filters having w and h dimension 3×3, creates constraints for the neuron in layer ℓ with a subset of the neurons in layer $\ell - 1$ that are part of a $3 \times 3 \times 2$ block, as shown in layer $\ell - 1$ in Fig. 3.

We call this set of neurons in layer $\ell - 1$ the first dependence set of $x_{(1,3,1)}^\ell$. Note that the first dependence set of $x_{(1,3,1)}^\ell$ and $x_{(1,3,2)}^\ell$ is the same. The second dependence set of $x_{(1,3,1)}^\ell$, also shown in Fig. 3, has size $4 \times 4 \times 2$ (filters between layer $\ell - 2$ and $\ell - 1$ have w- and h-dimension 2×2). The second dependence set of $x_{(1,3,1)}^\ell$ is obtained by taking the neurons in the output of the first dependence set and then for each neuron in this output, adding its corresponding first dependence set to the final output.

The dependence sets identify the dense submatrices (blue entries in M^k and F^k in Fig. 2) needed for computing backsubstitution through convolutional and residual layers. This enables a compute and memory-efficient GPU implementation that leverages dense matrix-matrix operations for high performance gains. Next we define dependence sets formally and then present our algorithms.

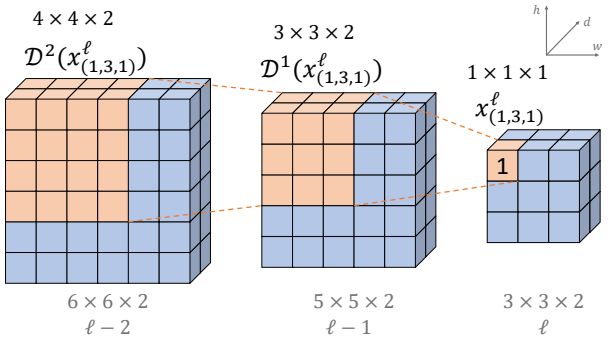


Figure 3. Backsubstitution from a single neuron in layer ℓ to layers $\ell - 1$ and $\ell - 2$. The number of neurons in layers ℓ, $\ell - 1$, $\ell - 2$ are $3 \times 3 \times 2$, $5 \times 5 \times 2$, and $6 \times 6 \times 2$ respectively.

Network DAG. We first define the *network DAG* associated with a neural network. In a network DAG $(\mathcal{V}, \mathcal{E})$, $\mathcal{V}$ is the set of all neurons. Two neurons are connected by a directed edge $(x_j^k, x_i^\ell) \in \mathcal{E}$ if x_j^k is directly needed to compute x_i^ℓ. More formally, $(x_j^k, x_i^\ell) \in \mathcal{E}$ if layer k is an immediate predecessor (contains inputs) of layer ℓ and

- ℓ is a convolutional layer and x_j^k is in the window for computing x_i^ℓ, or
- ℓ is a ReLU or a residual layer and $j = i$, or
- layer ℓ is fully-connected.

Note that for fully-connected and convolutional architectures, we have $k = \ell - 1$, while for a residual network, layer ℓ can have multiple immediate predecessors $k < \ell$.

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

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

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

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

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

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

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

During DeepPoly analysis, all neurons appearing in the polyhedral bounds obtained when backsubstituting iteratively on the polyhedral constraints for x_i^ℓ are available in the different sets $\mathcal{D}^{\ell-k}(x_i^\ell)$ with $k = \ell - 1, \ldots, 0$. The expression in the initial bound contains neurons from $\mathcal{D}^1(x_i^\ell)$ and we call it step 1 of backsubstitution. Step 2 substitutes for each neuron in $\mathcal{D}^1(x_i^\ell)$, the polyhedral bound defined over the neurons in $\mathcal{D}^2(x_i^\ell)$ resulting in a new bound for x_i^ℓ defined over the neurons in $\mathcal{D}^2(x_i^\ell)$. Continuing analogously, we see that $\mathcal{D}^{\ell-k}(x_i^\ell)$ contains the neurons appearing in the bounds after $\ell - k$ steps. In Section 4, we exploit the structure of the convolutional layers to derive recursive expressions for computing $\mathcal{D}^{\ell-k}(x_i^\ell)$ that enable fast computation with negligible overhead. Next, we discuss the backsubstitution for the different network types in greater detail.

Efficient backsubstitution for convolutional networks. Naively using a dense matrix-matrix multiplication for a backsubstitution starting at a convolutional layer ℓ is very memory and compute inefficient. First, the majority of computations are not needed since the filters in the convolutional layers are sparse and thus the filter matrix F^k of Fig. 2 consists of mostly zeroes. Additionally, it is not memory efficient since many coefficients in matrices M^k and M^{k-1} of Fig. 2 will be zero during backsubstitution.

Key idea. The neurons in the polyhedral expression for x_i^ℓ after $\ell - k$ backsubstitution steps ($0 \leq k < l$) are in the dependence set $\mathcal{D}^{\ell-k}(x_i^\ell)$. For an efficient implementation on GPUs, utilizing the dependence set, we can flatten the needed coefficients into a dense matrix to perform the backsubstitution again efficiently as matrix-matrix multiplication. This will be detailed in Section 4.

Dependence set and 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. An example of such an architecture is in Fig. 4 which shows a residual block consisting of one convolutional layer in each branch with all ReLU layers removed for simplicity.

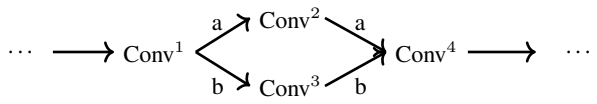


Figure 4. Simplified residual architecture without ReLU layers.

For simplicity, we assume that the two branches of a residual block have the same length and call them a and b. In Fig. 4 branches a and b contain the Conv^2 and Conv^3 layer, respectively. Naturally, the layer at the head of the residual block (Conv^1 in Fig. 4) has two successors while the one at exit (Conv^4 in Fig. 4) has two predecessors.

The first dependence set of a neuron x_i^ℓ in a layer at the exit of a residual block (e.g., Conv^4 in Fig. 4) contains neurons from both branches (subsets of layers Conv^2 and Conv^3 in Fig. 4). The resulting dependence set can be written as:

$$\mathcal{D}^1(x_i^\ell) = \mathcal{D}^{(1,a)}(x_i^\ell) \cup \mathcal{D}^{(1,b)}(x_i^\ell), \tag{4}$$

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

In our algorithm, we leverage the above partition of the first dependence set to backsubstitute through both branches independently and then join the independent backsubstitutions at the head of the residual block (in our case Conv^1) by adding the coefficients of the expressions neuron by neuron. For this, the two resulting dependence sets coming from the two residual branches, which do not necessarily have the same size, need to be overlapped correctly. We omit these details due to lack of space.

3.2 Early termination

The DeepPoly backsubstitution algorithm can be terminated early if the following criterion is met.

Termination criterion. The polyhedral approximation of a ReLU layer is exact if 0 is not strictly included within its bounds. In this case, no additional precision can be gained for this neuron by backsubstituting further.

An efficient implementation of DeepPoly should therefore filter those neurons out of the backsubstitution that satisfy this termination criterion. With the formalism introduced in Section 2, this amounts to removing a selection of rows out of the matrix M^k. We will propose a method to perform this operation efficiently with a shared memory machine model in Section 4.

4 GPUPoly

We now explain our GPUPoly algorithm. We first explain how to maintain floating point soundness using interval arithmetic. Next, we discuss the implementation of the early termination criterion. Then we discuss how to compute the size and elements of the dependence set $\mathcal{D}^{\ell-k}(x_i^\ell)$ of x_i^ℓ in layer $k < \ell$ for convolutional layers. We use this set in our parallel algorithm for the backsubstitution.

4.1 Floating point soundness

An essential property and major challenge is to ensure that our certification guarantees are valid under floating-point arithmetic (Jia & Rinard, 2020; Zombori et al., 2021) where round off errors are frequent and common mathematical

properties such as associativity do not hold. To be floating point sound, our analysis output should contain all results possible under different rounding modes and execution orders of operations. To achieve this, we replace the scalar coefficients of our polyhedra bounds with intervals. Therefore, our bounds actually describe a set of polyhedra instead of a single polyhedron.

To ensure soundness under all rounding modes, all floating point operations on intervals are performed such that the lower bound is always rounded towards $-\infty$ and the upper bound towards $+\infty$. This particularly prevents the use of standard BLAS libraries. Our matrix-matrix multiplication procedure is built around a custom multiply-add operation, and uses the cutlass template library for tiling. In addition, GPUPoly takes into account the error that may occur during inference, because of the lack of associativity for floating point operations. This is done by systematically taking the next representable floating value for the terms of all summations, in the direction that will over-approximate the error as described in detail in (Miné, 2004). Overall, ensuring floating point soundness doubles the memory requirement and more than doubles the number of floating point operations needed.

4.2 Implementing early termination

The original DeepPoly algorithm performs a complete backsubstitution for all the input neurons of ReLU layers. In this part, we describe the changes we introduced in GPUPoly to fully exploit the early termination criterion described in Section 3.

In order to have a first approximation of the interval bounds of hidden neurons, a forward interval analysis is performed as a preliminary step. Then, a regular DeepPoly analysis is performed, with the particularity that when a ReLU layer is encountered, its inputs (rows in the matrix M^k) are not directly backsubstituted. Instead, an intermediate matrix M'^k containing only the neurons that do not match the termination criterion is created, along with an array containing the indices of the corresponding rows in the original matrix M^k (we will explain this step later). Then, a backsubstitution is performed on the matrix M'^k, and the resulting interval bounds are assigned to their corresponding neurons, using the array. Finally, a forward interval analysis updates the approximations of the following layers, before regular DeepPoly analysis resumes.

By construction, GPUPoly visits ReLU layers in a topological order with respect to the network DAG. This ensures that all backsubstitutions only use the best possible polyhedral approximation of their ancestors, thus guaranteeing the same result as the original algorithm. In addition, during these backsubstitutions, concrete bounds are re-evaluated regularly, and the rows of the neurons that match the termination criterion are removed from M'^k.

In the worst case, GPUPoly computes all backsubstitutions completely, and has similar performance as if this optimization was not implemented, as the additional steps have a negligible runtime with respect to backsubstitutions. However, in many practical cases (as in Section 5), significantly fewer backsubstitutions are actually computed, yielding a significant speedup.

Removing rows from a matrix in a shared memory context. To create M'^k and the corresponding index array, each thread of the GPU is associated with one row of M^k, and checks whether the termination criterion is met for that row. A parallel prefix sum is then performed between all threads, with the value 0 if the termination is met, and 1 otherwise. This way, each thread associated with a non-terminated neuron receives a unique integer i ranging between 0 and the number of non-terminated neurons. Finally, each thread associated with a non-terminated neuron copies its corresponding row at the i^{th} row of M'^k, and writes its index at the i^{th} place of the array.

Memory management. For larger networks, the matrix M^k may not entirely fit in GPU memory. In these situations, the intermediate matrix M'^k can be used to sequentially backsubstitute chunks of M^k that are small enough to fit in memory.

4.3 Dependence sets for convolutional networks

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 $s_w^k = s_h^k = s^k$ and the padding $p_w^k = p_h^k = 0$.

In Fig. 3 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 offset of the elements of $(\ell-k)$-th dependence set $\mathcal{D}^{\ell-k}(x_i^\ell)$ as a subset of the neurons in a convolutional layer $0 \leq k < \ell$. $\mathcal{D}^{\ell-k}(x_i^\ell)$ is a cuboid and we compute the size of the set $\mathcal{D}^{\ell-k}(x_i^\ell)$ along the height, width and depth direction separately. We note that the k-th dependence set, given $k > 0$, is always dense in the depth direction for convolutional layers, so the size of $\mathcal{D}^{\ell-k}(x_i^\ell)$ in the depth dimension is equal to the number of channels of layer k. Because of our symmetry assumption for the w and h directions of convolutional parameters, the width and the height of $\mathcal{D}^{\ell-k}(x_i^\ell)$ are equal, and we denote it with $W^{\ell-k}$. The following recurrence computes $W^{\ell-k+1}$ given $W^{\ell-k}$:

$$\begin{aligned} W^0 &= 1, \\ W^{\ell-k+1} &= (W^{\ell-k} - 1) \cdot s^k + f^k, k = \ell \dots 1. \end{aligned} \quad (5)$$

For example, in Fig. 3 we obtain $W^1 = (W^0 - 1) \cdot 1 + 3 = 3$

for the first dependence set and $W^2 = (W^1 - 1) \cdot 1 + 2 = 4$ for the second dependence set. The overall size of $\mathcal{D}^{\ell-k}$ is:

$$|\mathcal{D}^{\ell-k}(x_i^\ell)| = W^{\ell-k} \cdot W^{\ell-k} \cdot C^k, \quad k = \ell - 1 \ldots 0. \quad (6)$$

where C^k is the number of channels of layer k. We compute the neuron indices next.

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

$$w^{\ell-k} = S^{\ell-k} \cdot w^\ell, \quad (7)$$

$$h^{\ell-k} = S^{\ell-k} \cdot h^\ell, \quad k = \ell - 1 \ldots 0. \quad (8)$$

where we introduced the quantity $S^{\ell-k}$, which we call *accumulated stride* computed via the following recurrence:

$$S^0 = 1, \quad (9)$$

$$S^{\ell-k+1} = s^k \cdot S^{\ell-k}, \quad k = \ell \ldots 1. \quad (10)$$

The extension to other padding modes is similar. Using the the size and the position of $\mathcal{D}^{\ell-k}(x_i^\ell)$ in layer k, we can now recursively compute, for $k = \ell - 1 \ldots 0$ the associated coefficients of the neurons in $\mathcal{D}^{\ell-k}(x_i^\ell)$ occurring in the backsubstituted expression. We store these in a dense matrix called $M^k(x_i^\ell)$. In each step these get modified by the backsubstitution:

$$M^{\ell-1}(x_i^\ell) = (a_1, a_2, \ldots, a_{|\mathcal{D}^1(x_i^\ell)|}),$$
$$M^{k-1}(x_i^\ell) = \texttt{GBC}(M^k(x_i^\ell), \mathcal{D}^{\ell-k}(x_i^\ell), \mathcal{D}^{\ell-k+1}(x_i^\ell), F^k), \quad 1 \le k \le \ell - 1.$$

$M^{\ell-1}(x_i^\ell)$ contains the coefficients corresponding to the neurons in the first dependence set in the initial polyhedra bound. We ignore the constant in the bound 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 bound matrix between the neurons in layer k and $k-1$ generated during DeepPoly analysis (Fig. 2). As in Section 2, F^k corresponds to the filter for convolutional layers. We next explain GBC in greater detail.

4.4 Our algorithm for convolutional networks

To be memory and compute efficient on GPU, our algorithm should compute the backsubstitution of the bound matrix M^k through one convolutional layer k obtaining M^{k-1} (as in Fig. 2), but for each row of the M^ks, only iterate over the respective dependence sets $\mathcal{D}_{hi}^{\ell-k}$ and $\mathcal{D}_{hi}^{\ell-k+1}$.

Algorithm 1 GBC($M^{\ell-k}, \forall hi : (\mathcal{D}_{hi}^{\ell-k}, \mathcal{D}_{hi}^{\ell-k+1}), F^k$)

1: $M^k, M^{k-1} \leftarrow$ coefficient matrices for layers $k, k-1$
2: $\mathcal{D}_{hi}^{\ell-k} \leftarrow (\ell-k)$-th dependence set of x_{hi}^ℓ
3: $(W^{\ell-k}, W^{\ell-k}, C^k) \leftarrow$ dimensions of $\mathcal{D}^{\ell-k}$
4: $\mathcal{D}_{hi}^{\ell-k+1} \leftarrow (\ell-k+1)$-th dependence set of x_{hi}^ℓ
5: $(W^{\ell-k+1}, W^{\ell-k+1}, C^{k-1}) \leftarrow$ dimensions of $\mathcal{D}_{hi}^{\ell-k+1}$
6: $(f^k, f^k) \leftarrow$ filter size in w and h directions of layer k
7: $(s^k, s^k) \leftarrow$ strides in w and h directions for layer k
8: $F^k \leftarrow$ 4-D filter weight tensor of layer k
9: $(f^k, f^k, C^k, C^{k-1}) \leftarrow$ dimensions of F^k
10: **for** $hi \in (h1 : ht)$ **do**
11: **for** $(w, h) \in (0 : W^{\ell-k}, 0 : W^{\ell-k})$ **do**
12: **for** $(f, g) \in (0 : f^k, 0 : f^k)$ **do**
13: $a = w \cdot s^k + f$
14: $b = h \cdot s^k + g$
15: **for** $c \in (0 : C^{k-1})$ **do**
16: $M_{hi}^{\ell-k+1}[a][b][c] = 0$
17: **for** $d \in (0 : C^k)$ **do**
18: $M_{hi}^{k-1}[a][b][c]$ += $M_{hi}^k[w][h][d] \cdot F^k[f][g][d][c]$

Ideally it should be possible to implement the algorithm via a matrix-matrix multiplication. Algorithm 1 satisfies these requirements.

The outermost loop $hi \in (h1 : ht)$ in line 10 iterates over the rows of M^k. In lines 11 and 17 the algorithm loops over the dimensions $(W^{\ell-k}, W^{\ell-k}, C^k)$ of $\mathcal{D}_{hi}^{\ell-k}$. Instead of iterating on the full range $(W^{\ell-k+1}, W^{\ell-k+1}, C^{k-1})$ of $\mathcal{D}_{hi}^{\ell-k+1}$ the sparsity of the convolution allows us to only loop over the dimensions (f^k, f^k, C^{k-1}) in lines 12 and 15 . This requires additional index computations in lines 13 and 14 for expressing addresses in layer $k-1$ in terms of those in layer k. Finally in line 18 all non-zero entries of M^{k-1} are computed given M^k and the filter F^k.

Next we discuss the parallelization strategy. A matrix-matrix multiplication always has one dimension which is collapsed ($(i1 : is)$ dimension in Fig. 2), and two dimensions which can be parallelized ($(h1 : ht)$ and $(j1 : jr)$ in Fig. 2). One of these two parallel dimensions is consecutive in memory ($(j1 : jr)$) while the other is not ($(h1 : ht)$). We follow the same strategy for the convolutional case, where the dimension we collapse is the loop in line 17 . The dimensions to be parallelized are the loops in lines 10 and 15 , where the loop in line 15 is consecutive in memory, as can be seen in line 18 where c is the inner dimension for the matrices M^{k-1} and F^k. All other loops are left serial.

Note that this algorithm can be understood as performing a separate transpose convolution for every h_i. This transpose convolution is a map from $\mathcal{D}_{hi}^{\ell-k}$ to $\mathcal{D}_{hi}^{\ell-k+1}$. To guarantee floating point soundness for our algorithm as discussed in Section 4.1, we optimize an interval-scalar matrix-matrix multiplication where the coefficients in M^k are intervals and F^k contains the scalar network weights.

Table 1. Neural networks used in our experiments.

Dataset	Model	Type	#Neurons	#Layers	Training
MNIST	6×500	Fully-connected	3,010	6	Normal
	`ConvBig`	Convolutional	48K	6	DiffAI
	`ConvSuper`	Convolutional	88K	6	Normal
	`IBP_large_0.2, IBP_large_0.4`	Convolutional	176K	6	CR-IBP
CIFAR10	6×500	Fully-connected	3,010	6	Normal
	`ConvBig`	Convolutional	62K	6	DiffAI
	`ConvLarge`	Convolutional	230K	6	DiffAI
	`IBP_large_2_255, IBP_large_8_255`	Convolutional	230K	6	CR-IBP
	`ResNetTiny`	Residual	311K	12	PGD
	`ResNet18`	Residual	558K	18	PGD
	`ResNetTiny`	Residual	311K	12	DiffAI
	`ResNet18`	Residual	558K	18	DiffAI
	`SkipNet18`	Residual	558K	18	DiffAI
	`ResNet34`	Residual	967K	34	DiffAI

Algorithm for residual blocks. Algorithm 1 can be used to backsubstitute through both branches of a residual block separately, as discussed in Section 3.1. The resulting two coefficient matrices then need to be added element-wise. Again we omit the details for space reasons.

Comparison to the parallel CPU implementation. Since the available parallelism of a modern CPU is at least an order of magnitude smaller compared to a GPU, the parallelized CPU implementation (Singh et al., 2019b) of DeepPoly processes fewer rows of M^k than GPUPoly in parallel. 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 indices and the corresponding coefficient. This representation does not exploit the structure of the convolutional layers and is not suitable for SIMD parallelization. In contrast, we exploit structured sparsity in convolutional layers via dependence sets which allows us to create smaller dense submatrices that are suitable for SIMD parallelization.

Comparison to standard backpropagation. Backpropagation (Grund, 1982) is fundamentally different from DeepPoly backsubstitution because it computes a scalar loss function and propagates it back to update the network weights while backsubstitution propagates constraints backwards. Further, backpropagation is usually only performed starting from the last layer which typically contains fewer neurons than the intermediate layers. In contrast, DeepPoly's 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 the backsubstitution on GPUs more challenging (Section 3). Overall, based on the above factors, the DeepPoly backsubstitution is mathematically different, computationally more expensive, and more memory-demanding than backpropagation.

5 Experimental Evaluation

We now demonstrate the effectiveness of GPUPoly for the verification of big neural networks in terms of both precision (number of instances verified) and performance in terms of runtime. GPUPoly is implemented in C++, supports 64-bit double and 32-bit single precision, and uses the CUDA library for GPU support and Cutlass for the template metaprogramming of matrix operations in CUDA. We compare the effectiveness of GPUPoly against two state-of-the-art verifiers: the CPU parallelized version of DeepPoly (Singh et al., 2019b) and the GPU based CROWN-IBP (CR-IBP) from (Zhang et al., 2020; Xu et al., 2020). We note that DeepPoly has the same precision as GPUPoly, however GPUPoly is at least 190x faster, for some networks even 68'000x, than DeepPoly. CR-IBP is implemented for GPUs and is more precise than interval bound propagation (Mirman et al., 2018; Gowal et al., 2018) and more scalable than CROWN-FULL (Zhang et al., 2018). We note that CROWN-FULL also has the same precision as DeepPoly (Salman et al., 2019), however its GPU implementation from (Zhang et al., 2020; Xu et al., 2020) runs out of memory on most of our networks, therefore we do not consider it. Thus CR-IBP is the most precise existing verifier that can scale to the big neural networks used in our experiments. We note that unlike GPUPoly and DeepPoly, CR-IBP is not floating point sound thus its verification results can be incorrect due to floating point errors (Jia & Rinard, 2020; Zombori et al., 2021).

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

Table 2. Experimental results for 10,000 images on fully-connected and convolutional neural networks: CR-IBP vs. GPUPoly.

Dataset	Model	#Neurons	ϵ	#Candidates	#Verified		Median runtime	
					CR-IBP	GPUPoly	CR-IBP	GPUPoly
MNIST	6×500	3,010	8/255	9,844	0	7,291	130 µs	9.06 ms
	`ConvBig`	48K	3/10	9,703	5,312	8,809	220 µs	537 µs
	`ConvSuper`	88K	8/255	9,901	0	8,885	300 µs	266 ms
	`IBP_large_0.2`	176K	0.258	9,895	4,071	7,122	190 µs	9.04 ms
	`IBP_large_0.4`	176K	3/10	9,820	9,332	9,338	190 µs	2.92 ms
CIFAR10	6×500	3,010	1/500	5,607	0	4,519	200 µs	8.04 ms
	`ConvBig`	62K	8/255	4,599	1,654	2,650	320 µs	730 µs
	`ConvLarge`	230K	8/255	4,615	1,672	2,838	900 µs	4.54 ms
	`IBP_large_2_255`	230K	2/255	7,082	5,450	5,588	820 µs	12.3 ms
	`IBP_large_8_255`	230K	8/255	4,540	3,289	3,298	270 µs	3.83 ms

Table 3. Experimental results for 500 images on fully-connected and convolutional networks: DeepPoly vs. GPUPoly.

Model	#Cand.	#Verif.	Median runtime	
			DeepPoly	GPUPoly
6×500	493	334	8.3 s	9.06 ms
`ConvBig`	487	441	12 s	537 µs
`ConvSuper`	495	428	271 s	266 ms
6×500	282	219	15 s	8.04 ms
`ConvBig`	226	127	38 s	730 µs
`ConvLarge`	232	138	309 s	4.54 ms

can also benefit state-of-the-art robust training methods (Balunovic & Vechev, 2020; Zhang et al., 2020) in the future as they depend on approximate verifiers for training.

Neural networks. We used 16 deep neural networks in our experiments as shown in Table 1. Out of these, 5 are MNIST-based (Lecun et al., 1998) and 11 are CIFAR10-based (Krizhevsky, 2009). Table 1 specifies the network architecture, the number of neurons, the number of layers and the training method for each network. There are 2 fully-connected, 8 convolutional and 6 residual architectures in Table 1. The largest network in the table is `ResNet34` with 34 layers and ≈1M neurons.

Regarding training, (i) 7 of our networks were trained using DiffAI (Mirman et al., 2018; 2019) and 4 with CR-IBP (Zhang et al., 2020), both of which perform *provably robust adversarial training*, (ii) 2 of our CIFAR10 networks were trained using Projected Gradient Descent (PGD) (Madry et al., 2018; Dong et al., 2018), which amounts to *empirically 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 which results in a loss of standard accuracy.

In the following we will refer to the non-residual networks as medium networks and to the residual networks as big networks.

Experimental setup. All our experiments for CR-IBP and GPUPoly were performed on a 2.2 GHZ 10 core Intel Xeon Silver 4114 CPU with 512GB of main memory. The GPU on this machine was an Nvidia Tesla V100 GPU with 16GB of memory. The PyTorch version used for running CR-IBP was 1.3.0 and the CUDA version for GPUPoly was 11.0. The experiments for the (prior) CPU version of DeepPoly were performed on a faster 2.6 GHz 14 core Intel Xeon CPU E5-2690 with 512GB of memory.

Benchmarks. For fully-connected and convolutional networks, we consider the full MNIST and CIFAR10 test sets. For the bigger residual networks, we selected the first $1,000$ images from the respective test set. We filtered out the images that were not classified correctly. We call the correctly classified images from the test set *candidate* images. The number of candidates for each network are shown in Table 2 and Table 4.

For each candidate image I_0, we define 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 verifiers in the literature. The ϵ values used for defining $L_\infty(I_0, \epsilon)$ for each neural network are shown in Table 2 and 4. We used larger values of ϵ for testing DiffAI and CR-IBP trained networks since these networks are more robust than the PGD and normally trained networks. However, DiffAI and CR-IBP trained networks suffer from a substantial drop in test accuracy (see #Candidates in Table 2 and Table 4). Furthermore, these networks are also easier to verify and thus even imprecise verifiers like CR-IBP verify large number of properties on these networks while GPUPoly takes advantage of the ease of verification by terminating the backsubstitution early, as explained in Section 4.2. This leads to the runtimes of GPUPoly being orders of magnitude smaller on DiffAI and CR-IBP trained networks compared to normally trained or PGD trained networks. We also note that the ϵ values for MNIST based networks are larger than for CIFAR10 based networks for the same reason.

Table 4. Experimental results for 1,000 images on big CIFAR10 residual networks: our implementation of CR-IBP vs. GPUPoly.

Model	#Neurons	Training	ϵ	#Candidates	#Verified		Median runtime	
					CR-IBP	GPUPoly	CR-IBP	GPUPoly
`ResNetTiny`	311K	PGD	1/500	768	0	651	5.3 s	11.7 s
`ResNet18`	558K	PGD	1/500	823	0	648	60 s	397 s
`ResNetTiny`	311K	DiffAI	8/255	371	203	244	5 s	4.03 ms
`SkipNet18`	558K	DiffAI	8/255	321	114	260	40 s	16.5 ms
`ResNet18`	558K	DiffAI	8/255	372	138	268	26 s	16.9 ms
`ResNet34`	967K	DiffAI	8/255	356	126	229	59 s	34.5 ms

5.1 Results on medium networks

Comparison with CR-IBP. Table 2 compares the precision and the median runtime of CR-IBP and GPUPoly on the medium fully-connected and convolutional networks for 10,000 images. We use the implementation of CR-IBP publicly available from (Zhang et al., 2020). On normally trained networks, CR-IBP does not prove any properties, while GPUPoly proves $20,695$ overall. On DiffAI and CR-IBP trained networks GPUPoly proves an additional $8,863$ properties overall compared to CR-IBP. CR-IBP is more precise on these networks than on normally-trained networks because 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. The number of such neurons for networks trained to be provable robust is relatively low. As can be seen, GPUPoly improves upon the state-of-the-art results. CR-IBP is up to 45x faster than GPUPoly on provably robust networks, and over 2,000x faster on normally trained networks. The speed of CR-IBP comes at the cost of imprecision and the lack of floating point soundness guarantees.

Distribution of runtimes. The runtimes of GPUPoly for normally and PGD trained networks are roughly normally distributed. On the other hand, the cumulative distribution function (CFD) of runtimes for DiffAI and CR-IBP trained networks has a big tail of values which are orders of magnitudes larger than the median. This is because the early termination succeeds in the majority of cases for robustly trained networks yielding very small runtimes. In the small number of instances when it fails, the runtime is quite high. The CFD plots for all networks are in the appendix A.

Comparison with DeepPoly. Table 3 compares the precision and runtime of DeepPoly and GPUPoly on six of our medium networks for the adversarial regions created on the first 500 test images on three MNIST and three CIFAR10 networks. The ϵ values are the same as in Table 2. While both have the same precision, GPUPoly is up to 250x faster than DeepPoly on normally trained networks and up to 68,000x faster than DeepPoly on DiffAI trained networks.

5.2 Results on big residual networks

In Table 4 we compare the precision and runtime of GPUPoly and CR-IBP on our big residual networks, the largest being a `ResNet34` with almost 1M neurons. Since CR-IBP does not support residual networks, we used our own implementation of CR-IBP, which does not employ many of CR-IBP's optimizations, such as batching, making it slower than the original, but equally precise. GPUPoly proves 1,299 samples for PGD trained networks overall while CR-IBP cannot prove any. Furthermore GPUPoly proves 420 additional properties compared to CR-IBP on DiffAI trained networks. GPUPoly only takes 34.5ms to verify our largest `ResNet34`.

6 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 an early termination mechanism. 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 precise polyhedral analysis to even larger models.

7 Acknowledgements

We would like to thank Simon Schirm, Cheng Lai Low, and Marco Foco for their advice which helped shape the initial CUDA implementation of GPUPoly.

REFERENCES

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

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

Balunovic, M., Baader, M., Singh, G., Gehr, T., and Vechev, M. T. Certifying geometric robustness of neural networks. In *Proc. Neural Information Processing Systems (NeurIPS)*, pp. 15287–15297, 2019.

Boopathy, A., Weng, T.-W., Chen, P.-Y., Liu, S., and Daniel, L. Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks. In *Proc. AAAI Conference on Artificial Intelligence (AAAI)*, Jan 2019.

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

Carlini, N. and Wagner, D. A. Towards evaluating the robustness of neural networks. In *Proc. IEEE Symposium on Security and Privacy (SP)*, pp. 39–57, 2017.

Dathathri, S., Dvijotham, K., Kurakin, A., Raghunathan, A., Uesato, J., Bunel, R., Shankar, S., Steinhardt, J., Goodfellow, I. J., Liang, P., and Kohli, P. Enabling certification of verification-agnostic networks via memory-efficient semidefinite programming. In *Proc. Neural Information Processing Systems (NeurIPS)*, 2020.

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

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

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

Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., and Vechev, M. AI2: Safety and robustness certification of neural networks with abstract interpretation. In *Proc. IEEE Symposium on Security and Privacy (SP)*, volume 00, pp. 948–963, 2018.

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

Grund, F. Rall, louis b., automatic differentiation: Techniques and applications. lecture notes in computer science 120. *ZAMM - Journal of Applied Mathematics and Mechanics*, 62(7), 1982.

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

Katz, G., Barrett, C. W., Dill, D. L., Julian, K., and Kochenderfer, M. J. 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.

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljić, A., Dill, D. L., Kochenderfer, M. J., and Barrett, C. The marabou framework for verification and analysis of deep neural networks. In *Proc. Computer Aided Verification (CAV)*, pp. 443–452, 2019.

Krizhevsky, A. Learning multiple layers of features from tiny images. Technical report, 2009.

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

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

Miné, A. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. EuropeanSymposium on Programming (ESOP)*, 2004.

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

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

Mirman, M., Gehr, T., and Vechev, M. Robustness certification of generative models, 2020.

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

Paterson, C., Wu, H., Grese, J., Calinescu, R., Pasareanu, C. S., and Barrett, C. Deepcert: Verification of contextually relevant robustness for neural network image classifiers, 2021.

Pei, K., Cao, Y., Yang, J., and Jana, S. Deepxplore: Automated whitebox testing of deep learning systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 1–18, 2017.

Raghunathan, A., Steinhardt, J., and Liang, P. S. Semidefinite relaxations for certifying robustness to adversarial examples. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 10877–10887. 2018.

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

Ruoss, A., Balunovic, M., Fischer, M., and Vechev, M. T. Learning certified individually fair representations. In *Proc. Neural Information Processing Systems (NeurIPS)*, 2020.

Ruoss, A., Baader, M., Balunovic, M., and Vechev, M. T. Efficient certification of spatial robustness. 2021.

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

Singh, G., Gehr, T., Mirman, M., Püschel, M., and Vechev, M. Fast and effective robustness certification. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pp. 10825–10836. 2018.

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

Singh, G., Gehr, T., Püschel, M., and Vechev, M. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL):41:1–41:30, 2019b. ISSN 2475-1421.

Singh, G., Gehr, T., Püschel, M., and Vechev, M. Boosting robustness certification of neural networks. In *International Conference on Learning Representations (ICLR)*, 2019c.

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

Tjandraatmadja, C., Anderson, R., Huchette, J., Ma, W., Patel, K., and Vielma, J. P. The convex relaxation barrier, revisited: Tightened single-neuron relaxations for neural network verification. In *Proc. Neural Information Processing Systems (NeurIPS)*, 2020.

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

Tran, H., Bak, S., Xiang, W., and Johnson, T. T. Verification of deep convolutional neural networks using imagestars. In Lahiri, S. K. and Wang, C. (eds.), *Proc. Computer Aided Verification (CAV)*, volume 12224 of *Lecture Notes in Computer Science*, pp. 18–42, 2020.

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

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

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

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

Xu, K., Shi, Z., Zhang, H., Wang, Y., Chang, K., Huang, M., Kailkhura, B., Lin, X., and Hsieh, C. Automatic perturbation analysis for scalable certified robustness and beyond. 2020.

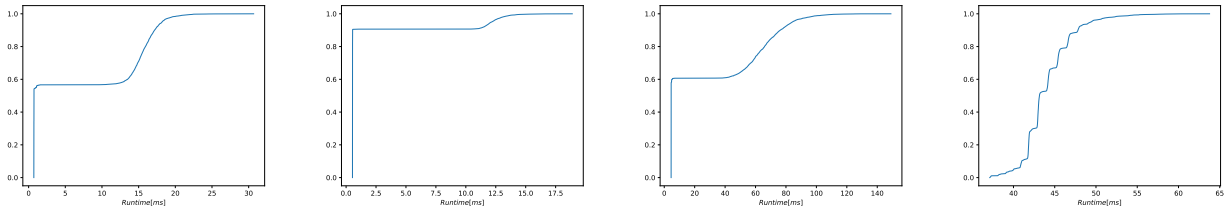
Zhang, H., Weng, T.-W., Chen, P.-Y., Hsieh, C.-J., and Daniel, L. Efficient neural network robustness certification with general activation functions. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*. 2018.

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

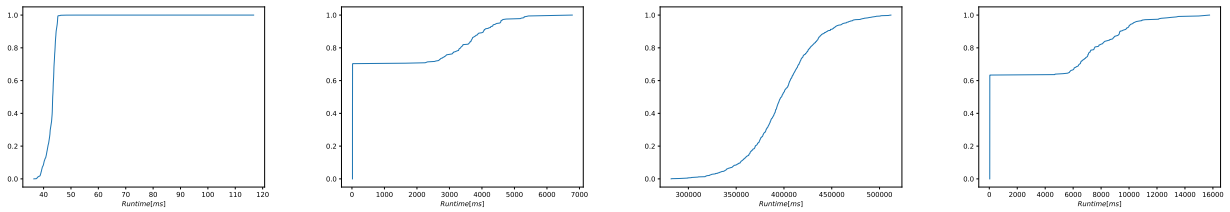
Zombori, D., Bánhelyi, B., Csendes, T., Megyeri, I., and Jelasity, M. Fooling a complete neural network verifier. In *Proc. International Conference on Learning Representations (ICLR)*, 2021.

A CUMULATIVE DISTRIBUTION FUNCTIONS OF RUNTIMES

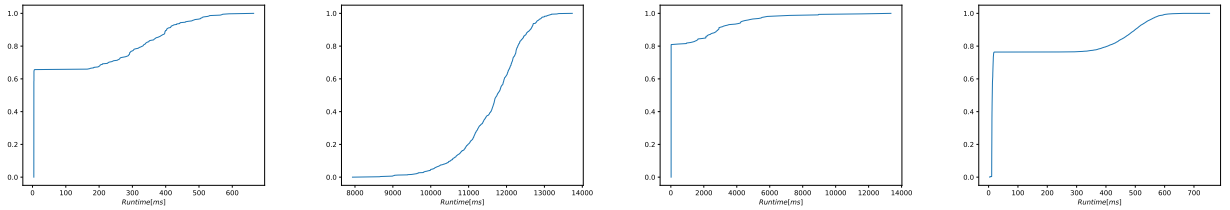
Fig. A shows the CDFs of the runtime of GPUPoly for the different networks. While the runtimes are roughly normally distributed for normally and PGD trained networks, the CDFs of the runtimes for DiffAI and CR-IBP trained networks have large tails on the right side. The reason for this is that most of the time early termination will result in a very low runtime for robustly trained networks, but sometimes the runtime can be orders of magnitudes higher.



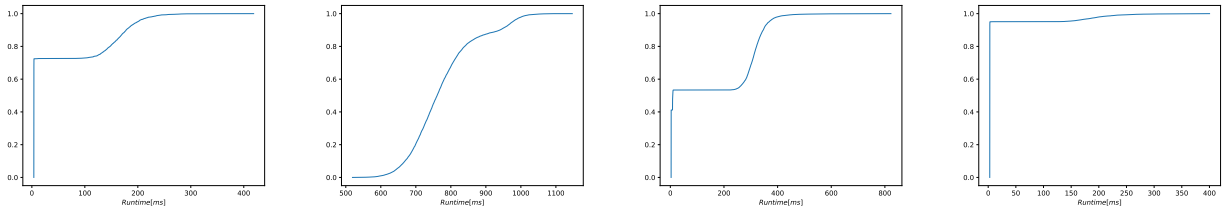
(a) ConvBig_cifar10 (b) ConvBig_mnist (c) ConvLargeIBP (d) FFNN_cifar10



(e) FFNN_mnist (f) ResNet18_DiffAI (g) ResNet18_PGD (h) ResNet34_DiffAI



(i) ResNetTiny_DiffAI (j) ResNetTiny_PGD (k) SkipNet18_DiffAI (l) cifar_crown_large_2_255



(m) cifar_crown_large_8_255 (n) convSuper_mnist (o) mnist_crown_large_0.2 (p) mnist_crown_large_0.4

Figure 5. CDF plot of the runtime of GPUPoly on the networks shown in Table 1.