Scaling-up the Analysis of Neural Networks by Affine Forms: a Block-Wise Noising Approach

Asma Soualah, Matthieu Martel and Stéphane Abide

September 8, 2023

Scaling-up the Analysis of Neural Networks by Affine Forms: A Block-Wise Noising Approach[1]

Asma Soualah
LAMPS - University of Perpignan
52 Avenue Paul Alduy
Perpignan, France
asma.soualah@univ-perp.fr

Matthieu Martel
LAMPS - University of Perpignan and Numalis
52 Avenue Paul Alduy
Perpignan, France
matthieu.martel@univ-perp.fr

Stéphane Abide
LAMPS - University of Perpignan
52 Avenue Paul Alduy
Perpignan, France
stephane.abide@univ-perp.fr

***Abstract*—The effectiveness of neural networks in handling visual perturbations are frequently assessed using abstract transforms, such as affine transformations. However, these transforms may forfeit precision and be computationally expensive (time and memory consuming). We suggest in this article a novel approach called block-wise noising to overcome these limitations. Block-wise noising simulates real-world situations in which particular portions of an image are disrupted by inserting non-zero noise symbols only inside a given section of the image. Using this method, it is possible to assess neural networks resilience to these disturbances while preserving scalability and accuracy. The experimental results demonstrate that the present block-wise noising achieves a 50% speed improvement compared to the usual affine forms on specific trained neural networks. Additionally, it can be especially helpful for applications like computer vision, where real-world images may be susceptible to different forms of disturbance.**

***Index Terms*—Scalability, Optimisation, Interpretation abstract, artificial intelligence.**

I. Introduction

Neural networks [1, 2] are increasingly being employed in mission-critical systems [3, 4] such as self-driving cars, airplanes, air traffic control systems, health monitoring systems and many more. These networks are frequently used to make choice decisions in real time, such as object recognition, trajectory prediction, anomaly detection, etc. The application of neural networks in critical systems, on the other hand, raises substantial safety challenges. Indeed, these networks are frequently built using training data that are biased or incomplete. Furthermore, networks can be vulnerable to adversarial attacks [5]. As a result, neural network validation and safety verification in mission-critical systems are crucial. To guarantee that neural networks perform correctly under all possible scenarios, formal verification approaches such as abstract interpretation [6, 7], Satisfiability Modulo Theories (SMT) [8, 9] and property testing [10] are frequently used.

Complex tasks require deep neural network models with a large number of parameters that have to be trained. For example, Resnet-50 [11] requires 25.5 million parameters, AlexNet [12] 61 million parameters, and VGG-16 [13] 138 parameters for training. Verifying these neural networks using abstract interpretation, such as affine forms, presents a challenge of balancing scalability and precision [14].

Incorporating affine forms in the verification of neural networks becomes computationally intensive process that may take several days or even weeks to complete. This is due to the necessity of updating the model parameters by adding noise symbols for each pixel of the input. For instance, VGG-16 [13] takes an input of size (224×224), so we add (224×224) noise symbols in the first input layers of the model. As the neural networks grow in size and the available datasets increase, the number of noise symbols grows as well, then the running neural network becomes more complex and computationally intensive.

To adress this issue, we propose in this article a new approach called block-wise noising, which consists of adding non-zero noise symbols only within a defined region of an image (see Figure 8). Intuitively, we divided our input into n blocks. Then we add non-zero noise symbols to only one region of each block, while leaving all other regions with zero noise symbols. We repeat this technique for all blocks of our input. When the computation is complete, we notice that the sum of all blocks exactly matches the results produced by the original input with the usual affine forms. The main advantage of this approach is that computations are significantly faster and require less memory compared to the usual affine forms. On average, the execution time is reduced by 50% in the majority of the case (see Section V).

We present some experimental results to assess the efficiency of block-wise noising in practice. These experiments are done by evaluating trained neural networks with affine forms. We used noised grayscale images as inputs. In short, our experiments show that block-wise noising is much faster than usual affine forms (for instance, when using 4 blocks, the execution time is reduced by a factor of 2). Block-wise noising parallel has also been implemented, and the findings demonstrate that it can speed up computation by 3.1–4 times (see Section VI). In other words, our technique strikes a balance between scalability and precision, enabling significant speedups and memory savings without sacrificing precision.

This article is organized as follows. In Section III, we briefly describe neural networks, affine arithmetic and activation functions. We introduce our block-wise noising approach in

[1]This work was supported by a regional funding (Region Occitanie).

Section IV. Section VI deals with parallel block-wise noising. The experimental results are described in Section V and we conclude in Section VII.

II. Overview

In this section, we introduce informally our block-wise noising technique, and we illustrate how to compute with it. These ideas are formalized further in Section IV. In our example, illustrated in Figure 1, we use a 2×2 matrix of affine forms. For instance, $x_{00} = 2 + \varepsilon_1$. We also use a 2×4 matrix of weights W and a vector b for the bias. Our technique is taylored for the analysis of neural networks. So noise is attached to the input x while W and b corresponds to the weights of the NN and contains scalar coefficients.

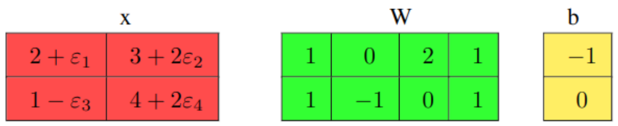

Fig. 1: Inputs given to a fully connected layer.

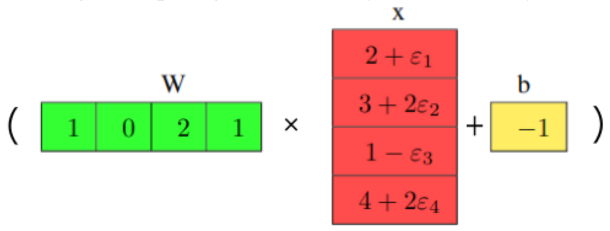

Fig. 2: The first operation of fully connected layer.

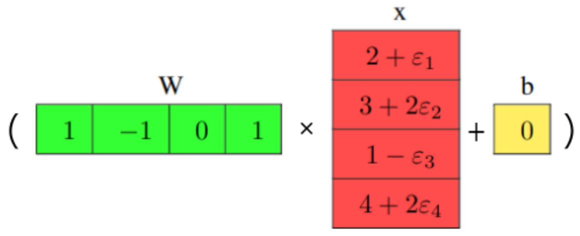

Fig. 3: The second operation of fully connected layer.

As usual in neural networks, a fully connected layer computes $y_j = \sum_{i=1}^{n} W_{ij}.x_i + b_j$. We obtain for our example:

$$y_1 = 7 + \varepsilon_1 - 2\varepsilon_3 + 2\varepsilon_4, \quad y_2 = 3 + \varepsilon_1 - 2\varepsilon_2 + 2\varepsilon_4.$$

For efficiency reasons, instead of x, we want to use block-wise noising, so we split x into two blocks: Block 1 and Block 2 (see Figure 6).

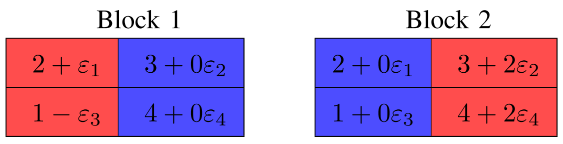

Fig. 6: Splitting of x into two block-wise noising.

For example, in the case of Block 1 we add non-zero noise symbols for the red part and zero noise symbols for the blue part. Conversely, for Block 2 we set the noise symbols for the red parts and a zero noise symbols in the blue parts.

We compute fully connected layers for Block 1 and Block 2.

For Block 1, we have

$$b_{11} = 7 + \varepsilon_1 - 2\varepsilon_3, \quad b_{12} = 3 + \varepsilon_1.$$

And for Block 2, we have

$$b_{21} = 7 + 2\varepsilon_4, \quad b_{22} = 3 - 2\varepsilon_2 + 2\varepsilon_4.$$

We can observe that

$$b_{11} + b_{21} = 14 + \varepsilon_1 - 2\varepsilon_3 + 2\varepsilon_4 \quad = y_1 - 7.$$

Finally,

$$b_{12} + b_{22} = 6 + \varepsilon_1 - 2\varepsilon_2 + 2\varepsilon_4 \quad = y_2 - 3.$$

The constants 7 and 3 subtracted to $y1$ and $y2$ are due to the fact that centers are added twice in the formula and must be removed. See Section IV for details.

In this way, we reduce the execution time and the memory needed to calculate a fully connected layers without precision loss. We will see that this approach also holds for other kinds of layers, convolutional layers in particular.

III. Background Material

In this section, we introduce some basic concepts of neural network analysis by affine forms. Section III-B covers neural network zonotope, Section III-A covers affine forms, while Section III-C discusses abstract transformers for activation functions.

A. Affine Forms

The affine forms [14, 15] and operations between them are introduced in this section. An affine form $\hat{x}$ is defined by:

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \quad . \tag{1}$$

Where, x_0, x_i, $i > 0$ and ε_i represent the center value, partial deviations and noise symbols, respectively. The values of these noises are unknown but lie in the interval [-1, 1].

The elementary operations among affine forms $+$, $-$ and multiplication by constant are defined as follows.

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i)\varepsilon_i \quad , \tag{2}$$

$$\hat{x} \pm a = (x_0 \pm a) + \sum_{i=1}^{n} (x_i)\varepsilon_i \quad , \tag{3}$$

$$\hat{x} \times a = (a x_0) + \sum_{i=1}^{n} (a x_i)\varepsilon_i \quad . \tag{4}$$

The multiplication is one of the non-univariate operations, the product of two affine forms is defined by:

$$\begin{aligned} \hat{x} \times \hat{y} &= (x_0 + \sum_{i=1}^{n} x_i \varepsilon_i) \times (y_0 + \sum_{i=1}^{n} y_i \varepsilon_i) \\ &= x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i) + \left(\sum_{i=1}^{n} |x_i| \times \sum_{i=1}^{n} |y_i| \right) \varepsilon_{n+1} \quad . \end{aligned}$$

B. Deep Neural Networks

In this paper, we consider several kinds of neural networks layers, enumerated hereafter.

Fully connected layers [16]: also known as dense layers, involve every neuron in one layer being connected to every neuron in the next layer, they are commonly used in the output of neural networks. The entries of the output y for a given input x are given by

$$y_j = \sum_{i=1}^{n} x_i.W_{ij}w + b_j \ \ , \tag{5}$$

where $W \in \mathbb{R}^{n\times m}$ assigns a weight to each edge of the network, b $\in \mathbb{R}^n$ assigns a bias to each node and x assigns an affine input as follows: $x = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i$.

Convolutional layers [16]: are used to extract features from input data, such as images, by applying a set of learnable filters to local regions of the input. Each convolutional operation includes stride, filter size, and zero padding, a positive integer that specifies sliding steps for each operation. This allows the network to learn spatial hierarchies of patterns and features that can be used for classification, segmentation, and other image processing tasks. The entries of the output y for a given input x are given by

$$y_{i,j} = \sum_{i'=1}^{p}\sum_{j'=1}^{q} W_{i'j'}.x(i+i'-1)(j+j'-1) + b_i \ \ . \tag{6}$$

Pooling layers [17]: are typically applied after convolutional layers to reduce the dimensionality of the input. Max Pooling, the most common pooling method using the Max function inside the pooling filter (2×2) as output.

In DNN, there are two kinds of operations: addition and multiplication by a constant, whose definition in the zonotope domain are given in equations (2) and (4).

C. Activation Functions

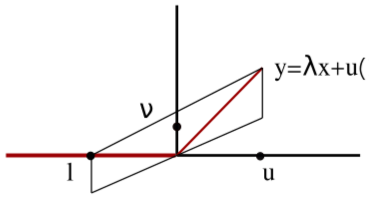

Fig. 7: The zonotope approximation of the Relu function.

Affine forms is a successful approach for verifying neural networks, as they are fast and exact for affine transformations [14]. However, when it comes to model nonlinear activation functions such as $Relu$, the zonotope abstraction [18] is not exact. Therefore, an approximation technique [7] which creates a tradeoff between computational cost and precision is necessary. Following the approach developed in [19] for intervals, we present hereafter our approximation method that strikes a balance between computational cost and precision.

Let x be an affine form $x = x_0+\sum_{i=1}^{n} x_i\varepsilon_i$ given to a $Relu$ function (y = $Relu(x)$). Let $l_x = x_0 + (\sum_{i=1}^{n}|x_i|)\times -1$ and $u_x = x_0 + (\sum_{i=1}^{n}|x_i|)\times 1$ denote the lower and upper bounds respectively, for the input x. The abstract transformer of the $Relu$ activation function is given by:

$$Relu(x) = \begin{cases} x & for \ \ l_x \geq 0 \\ 0 & for \ \ u_x \leq 0 \\ \lambda x + \nu + \nu\varepsilon & otherwise \end{cases}$$

where λ = $\frac{u_x}{u_x - l_x}$ and ν = $\frac{u_x(1-\lambda)}{2}$ represents the minimum of the area of the parallelogram in the xy-plane and the center of the zonotope in the vertical axis respectively (see Figure 7).

Let us consider an example to illustrate this concept. Suppose we have an affine input x represented as $x = 1+2\varepsilon_1+3\varepsilon_2$ and we want to compute y = $Relu(x)$. Here, $l_x = -4$, $u_x = 6$, $\lambda = 0.6$ and $\nu = 1.2$. So the result of $Relu$ is

$$1.8 + 1.2\varepsilon_1 + 1.8\varepsilon_2 + 1.2\varepsilon_3 \ \ .$$

We end this section by introducing the soundness of the abstract $Relu$. First of all, let us define the concretization of an affine form $\hat{x}$ defined in equation 1 into an interval.

$$\gamma_I(\hat{x}) = x_0 + \left(\sum_{i=1}^{n}|x_i|\right) \times [-1,1].$$

Then, we define the soundness of the abstract transfer activation functions $Relu$, Let $x \in \mathbb{R}$ and $x^\sharp \in$ Aff, such that $x \subseteq \gamma(x^\sharp)$. Then $Relu(x) \subseteq \gamma(Relu(x^\sharp))$.

IV. Block-wise Noising

We propose in this section a new approach called block-wise noising. This approach strikes a balance between scalability and precision, enabling a faster approximation without sacrificing precision. It can be particularly useful for applications such as computer vision, where real-world images are often subject to various types of perturbations.

Let us see this example in order to clarify this approach: we split our input into, n regions, as shown in Figure 8 (in our example we take $n = 4$). Then, we add non-zero noise symbols only to one region (red parts) and zero noise symbols for all other regions (gray parts). We repeat this process for

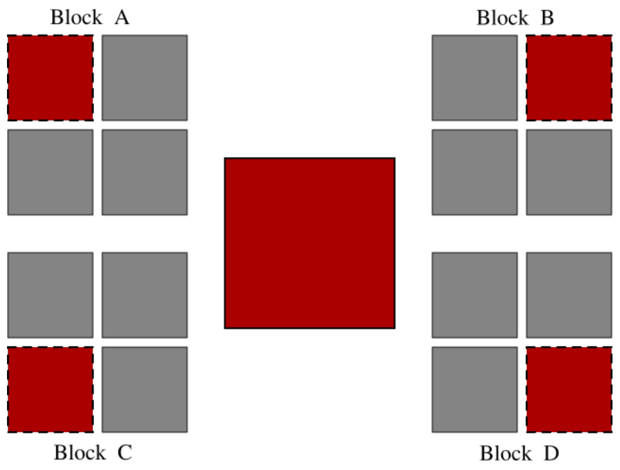

Fig. 8: Splitting input into block-wise noising.

all regions of our input. At the end of the computation, for linear layers (fully connected or convolutional without *Relu*), we observe that the sum of all regions is exactly the same as the results given by the original input with affine forms (one block). To prove this, we have the following two theorems:

Theorem IV.1. *Let x and y two affine forms defined by :*

$$x = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \ , \quad and \ \ y = y_0 + \sum_{i=1}^{n} y_i \varepsilon_i \ .$$

If $n = 2^u$, then x and y can be split into k pieces of size 2^v with $v \leq u$

$$x = \sum_{i=1}^{k} a_i \ , \quad with \ \ a_i = x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \ .$$

And,

$$y = \sum_{i=1}^{k} b_i \ , \quad with \ \ b_i = y_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} y_j \varepsilon_j \ .$$

Then,

$$x + y \quad = \quad x + y - (k-1)(x_0 + y_0) \ .$$

Proof. Base Case: Let us consider the case when $n = 2^0$. In this case, we can split x and y into $k = 1$ pieces of size 2^0 with $v \leq u \leq 0$. The theorem states that:

$$x = a_1 = x_0 + x_1\varepsilon_1 \ , \quad and \ \ y = b_1 = y_0 + y_1\varepsilon_1 \ .$$

Now let us calculate $x + y$

$$\begin{aligned} x + y &= a_1 + b_1 \ , \\ &= (x_0 + y_0) + (x_1 + y_1)\varepsilon_1 \ , \\ &= x + y - 0(x_0 + y_0) \ . \end{aligned}$$

We can see that the theorem holds for the base case.

Inductive Step: Now, let us assume that the theorem holds for $n = 2^u$. We will prove that it also holds for $n = 2^{u+1}$. In other words, we assume that for $n = 2^u$, we can split x and y into k pieces of size 2^v with $v \leq u$, and the theorem holds for this case.
Now, let us consider the case $n = 2^{u+1}$. We can split x and y into k pieces of size 2^{v+1} with $v + 1 \leq u + 1$.

$$x = \sum_{i=1}^{k} a_i \ , \quad with \ \ a_i = x_0 + \sum_{j=(i-1)\times 2^{v+1}+1}^{2^{v+1} \times i} x_j \varepsilon_j \ .$$

And,

$$y = \sum_{i=1}^{k} b_i \ , \quad with \ \ b_i = y_0 + \sum_{j=(i-1)\times 2^v+1}^{2^{v+1} \times i} y_j \varepsilon_j \ .$$

We have

$$\begin{aligned} x + y &= \sum_{i=1}^{k} (a_i + b_i) \ . \\ &= \sum_{i=1}^{k} (x_0 + y_0) + \sum_{j=(i-1)\times 2^{v+1}+1}^{2^{v+1} \times i} (x_j + y_j)\varepsilon_j \ . \end{aligned}$$

Then,

$$x + y \quad = \quad x + y - k(x_0 + y_0) \ .$$

We have shown that if the theorem holds for $n = 2^u$, then it also holds for $n = 2^{u+1}$. By the principle of mathematical induction, the theorem holds for all values of $n = 2^u$. □

Let us consider now the multiplication of an affine form by a constant.

Theorem IV.2. *Let x an affine form defined by :*

$$x = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \ .$$

If $n = 2^u$, then x can be split into k pieces of size 2^v with $v \leq u$

$$x = \sum_{i=1}^{k} a_i \ , \quad with \ \ a_i = x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \ .$$

Then,

$$x \times c = x \times c - (k-1)(x_0 \times c) \ .$$

Proof. Base Case: Let us consider the case when $n = 2^0$. In this case, we can split x into $k = 1$ pieces of size 2^0 with $v \leq u \leq 0$. The theorem states that:

$$x = a_1 = x_0 + x_1\varepsilon_1 \ .$$

Now let us calculate $x \times c$

$$\begin{aligned} x \times c &= a_1 \times c \ , \\ &= (x_0 \times c) + (x_1 \times c)\varepsilon_1 \ , \\ &= x \times c - 0(x_0) \ . \end{aligned}$$

We can see that the theorem holds for the base case.

Inductive Step: Now, let us assume that the theorem holds for $n = 2^u$. We will prove that it also holds for $n = 2^{u+1}$. In other words, we assume that for $n = 2^u$, we can split x into k pieces of size 2^v with $v \leq u$, and the theorem holds for this case.
Now, let us consider the case $n = 2^{u+1}$. We can split x into k pieces of size 2^{v+1} with $v + 1 \leq u + 1$.

$$x = \sum_{i=1}^{k} a_i \ , \quad with \ \ a_i = x_0 + \sum_{j=(i-1)\times 2^{v+1}+1}^{2^{v+1} \times i} x_j \varepsilon_j \ .$$

We have

$$\begin{aligned} x \times c &= \sum_{i=1}^{k} c \times a_i \ , \\ &= \sum_{i=1}^{k} x_0 \times c + \sum_{j=(i-1)\times 2^{v+1}+1}^{2^{v+1} \times i} x_j \varepsilon_j \times c \ . \end{aligned}$$

Then,

$$x \times c = x \times c - k(x_0 \times c) \ .$$

We have shown that if the theorem holds for $n = 2^u$, then it also holds for $n = 2^{u+1}$. By the principle of mathematical induction, the theorem holds for all values of $n = 2^u$. □

V. Efficiency of Block-Wise Noising

In this section, we investigate the efficiency of block-wise noising, in terms of execution time. We present the experimental protocol in Section V-A and report the results in Sections V-B and V-C.

A. Experimental Setting

In this section, we outline the methodology to evaluate block-wise noising in terms of execution time. We employed fully connected feedforward and convolutional neural networks to process grayscale images of size 28×28, with pixel values normalized to the range between -1 and 1 with randomly assigned weights between -1 and 1. The Rectified Linear Unit (Relu) function is used as the activation function.

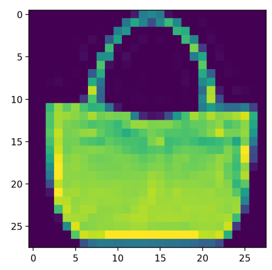

Fig. 9: Image used to test the efficiency of block-wise noising.

Fully Connected Feedforward Networks: we use networks consisted of $L = 2, 4$ and 6 layers. Each layer contains 28×28 neurons. The input given to our neural networks is displayed in Figure 9. The networks input, denoted as x_0, is a vector of 28×28 affine forms, each accompanied by a noise symbol representing the pixels intensity. Let c_{ij} represent the color of the pixel (i, j) and ν denote the noise intensity. The value of the $i \times n + j$ component of the input vector x_0 is given by:

$$x_0[i \times n + j] = c_{ij} + \nu c_{ij} \varepsilon_{i \times n + j} \quad . \tag{7}$$

Note that we associate the same intensity of noise to all the pixels and we consider $\nu = 0.02$.

Convolutional Networks: The networks consisted of $L = 4, 5$ and 6 layers. The input x_0 has a size 28×28 with 28×28 noise symbols (one noise symbol per pixel). Each convolutional networks consist of convolution, nonlinearity, and pooling layers.

Now, let us focus on the case of the block-wise noising. First, let us underline that we need to use the same neural network that we are validating without modification. We consider the cases where our input is split into 4, 8 and 16 blocks.

In our experiments, we measured the time needed to execute the neural network with different numbers of blocks.

All experiments were performed with Python on a Dell Inc. Latitude 5400 laptop, which featured an Intel Core $i5-8365U$ CPU running at 1.60 GHz 8 core with 8.0 GB of RAM.

B. Execution Time

Our experimental results, displayed in figures 11 and 12 which represents the execution time taken to run a fully connected feedforward neural network and convolutional neural network, respectively. With different layers in function of the number of blocks used to split the input (1, 4, 8, and 16) blocks.

Dataset	1 blocks	4 blocks	8 blocks	16 blocks
Sac	$2.10^4 s$	$9,1.10^3 s$	$8.10^3 s$	$7,9.10^3 s$
Sneakers	$1,910^4 s$	$8,7.10^3 s$	$7,1.10^3 s$	$6,9.10^3 s$
Dress	$2,1.10^4 s$	$9,2.10^3 s$	$8,1.10^3 s$	$7,9.10^3 s$
T-shirt	$2,1.10^4 s$	$9,2.10^3 s$	$8,1.10^3 s$	$7,9.10^3 s$

Fig. 10: Execution time for dataset MNIST with fully connected neural networks at 2 layers.

It can be observed that for a fully connected feedforward neural network, the execution times increases as the number of layers increases.

Also, we note that as the number of blocks increases, execution times decrease. For instance, the execution times taken by the neural networks, for $L = 4$ are: $5.8e + 4s$, $2.5e + 4s$, $2.2e + 4s$ and $2.1e + 4s$. For 1, 4, 8, and 16 blocks, respectively. The results show that the block-wise noising divided the execution times by 2 without loss of precision. As seen in Figure 10, these results hold true for all input types.

Additionally, in the case of the convolutional neural networks, the execution times drop as the number of input blocks grows, but they start to rise once the number of input blocks exceeds 4. For example, for $L = 5$ the execution times taken by the neural networks are: $1.0e + 3s$, $6.0e + 2s$, $6.9e + 2s$ and $8.9e + 2s$. For 1, 4, 8 and 16 blocks, respectively. This can be explained by the fact that, as the number of blocks increases, overlaps between blocks become more frequent, leading to redundant calculations. In other words, convolution operations are repeated for pixels that are common to different blocks, resulting in an increase in execution time compared to fully connected neural networks.

Furthermore, we observe that the convolutional layers are faster than fully connected layers, we can explain by the fact that convolutional layers utilize spatial locality by sharing weights and conducting local operations, whereas fully connected layers require all input neurons to be connected to all output neurons, which can result in a larger number of computations.

C. Experimenting with Trained Neural Networks

In this section, we demonstrate some experimental outcomes using five trained neural networks (NNs). All these NNs are classifers. They are described in Figure 13. The first column of the table gives the model and its input, the second column shows the number of layers, the third column gives the number of neurons, and the fourth column gives the number of parameters. A complete description of the network architectures is given in Appendix A.

We use the three popular datasets for our experiments: MNIST, Fashion-MNIST [21] and CIFAR-10 [22]. MNIST and Fashion-MNIST contains 60000 grayscale images of size (28×28) and Cifar-10 contains 60000 grayscale images of

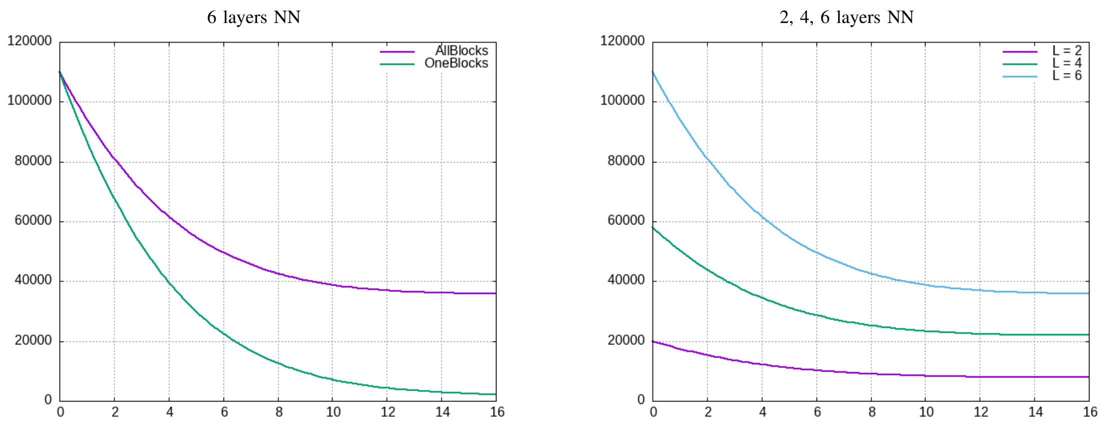

Fig. 11: Execution time in function of the number of blocks for AllBlocks (Block A, Block B, Block C, Block D) and OneBlock (Block A) of figure 8. Input image has a size 28×28 with 2, 4 and 6 layers for the fully connected neural network. We use the image of Figure 9.

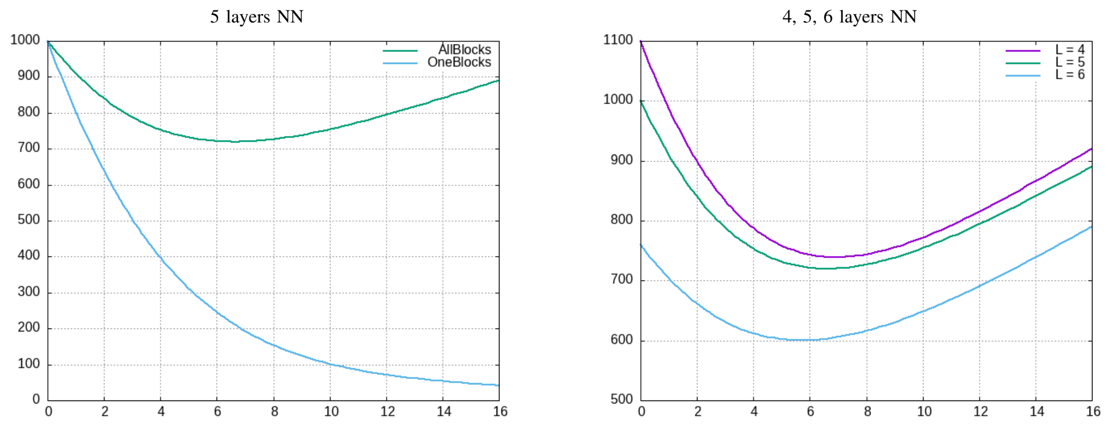

Fig. 12: Execution time in function of the number of blocks for AllBlocks (Block A, Block B, Block C, Block D) and OneBlock (Block A) of figure 8. Input image has a size 28×28 with 4, 5 and 6 layers for the convolutional neural network. We use the image of Figure 9.

Model	Input	Layers	Neurons	Parameters
FashionMNIST				
CNN	(28×28)	7	266	$431,242$
FC	(28×28)	6	970	$575,050$
CIFAR				
CNN	(32×32)	9	522	$443,882$
MNIST				
CNN	(28×28)	9	202	$157,258$
FC	(28×28)	5	234	$111,146$

Fig. 13: Model configurations used in our experiments.

size (32×32). We trained two types of neural networks: fully connected and convolutional neural networks.

We transform the input using affine forms. Then we apply the $ReLU$ activation function which, is previously defined in Section III-C.

Figure 14 displays the results of our measurement of the time required to run each neural network with various amounts of blocks.

We notice that for all our fully connected networks, which use different types of data (MNIST and Fashion-MNIST), the execution time decreases significantly when the number of blocks increases. For example, in the case of a fully connected network that use the Fashion-MNIST dataset, the execution times are: $13000s$, $5700s$ and $5080s$ for 1, 4 and 8 blocks, respectively. It is shown that when we use our block-wise noising method with a fourth block the execution time is roughly divided by 2.

We can observe that in the case of the convolutional networks the execution times drop as the number of input blocks grows, but they start to rise once the number of input blocks exceeds 4. For example, for CNN that use the Fashion-MNIST dataset the execution times are: $5700s$, $3960s$ and 4480s. For 1, 4 and 8 blocks, respectively. This can be explained by the fact that, as the number of blocks increases, overlaps between blocks become more frequent, leading to

Model	Input	1 blocks	4 blocks	8 blocks
FashionMNIST				
CNN	(28×28)	$5700s$	$3960s$	$4480s$
FC	(28×28)	$13000s$	$5700s$	$5080s$
CIFAR				
CNN	(32×32)	$20000s$	$15000s$	$160000s$
MNIST				
CNN	(28×28)	$5600s$	$4300s$	$5300s$
FC	(28×28)	$1100s$	$520s$	$476s$

Fig. 14: Execution time with fully connected and convolutional neural networks at different depths.

redundant calculations. In other words, convolution operations are repeated for pixels that are common to different blocks, resulting in an increase in execution time compared to fully connected layers.

VI. Block-Wise Noising Parallelism

The computational cost of training and inference increases along with the complexity of neural networks. To address this issue, parallelization techniques have emerged as a crucial method for reducing the execution time of neural networks. Data parallelism [23] and model parallelism [24] are some of the commonly used methods for parallelizing neural networks. However, in our study, we aim to parallelize affine forms instead of neural networks.

By incorporating a few synchronization primitives, we use the design and output of block-wise noising (as described in Section V) to generate a simple model for parallel implementation. One strategy to parallelize the block-wise noising is to

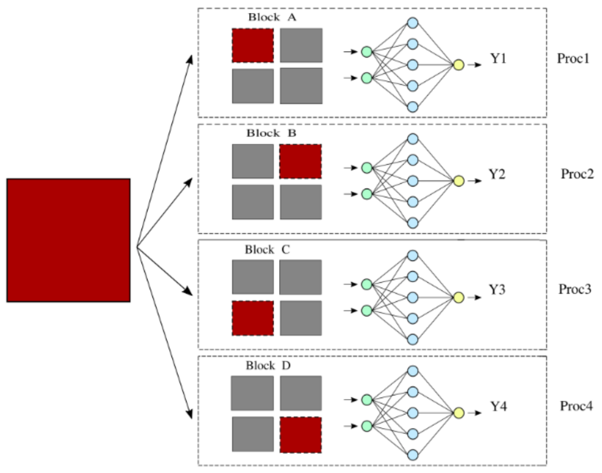

Fig. 15: Block-Wise Noising Parallelism.

distribute the computations of each block (Block A, Block B, Block C and Block D) as shown in Figure 15 between the

different processor and performs those computations simultaneously :

$$Input = [BlockA, BlockB, BlockC, BlockD]$$

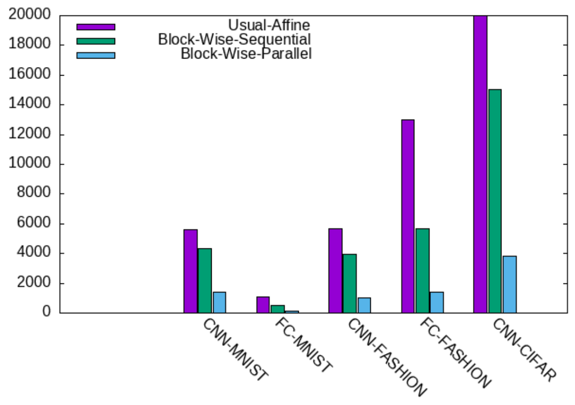

Fig. 16: Execution time of Block-Wise Noising parallelism: 4 blocks and 4 processors.

By parallelizing the computation of affine forms in this manner, we can achieve faster execution times than sequential block-wise noising for complex neural networks.

Note that, we use the experimental setting described in Section V and we measure in our experiments the time needed to execute the neural network.

All experiments were performed with Python on a cluster named MUSE, which consisted of 308 PowerEdge $C6320$ servers based on Intel Xeon $E5 - 2680v4$ chips, and each server had $128GB$ of RAM.

A. Execution time

The execution time of trained neural networks (NNs), described in Appendix A, is shown in figures 16, 17 and 18 of the experimental results presented in this section. The top-right histogram represents NNs with 8 blocks and 4 processors (each processor calculates 2 blocks), the top-left histogram corresponds to NNs with 4 blocks and 4 processors, and the bottom histogram corresponds to NNs with 8 blocks and 8 processors. Each histogram displays a set of three bars for each NNs:

The first bar represents the usual affine form, the second bar block-wise noising ($n = 4$ and $n = 8$) and the last bar block-wise noising parallelism, respectively.

The result shows that the execution time obtained using 4 and 8 processors are more faster than those obtained with the sequantial version. As an example, for the CNN-CIFAR model the execution times obtained with 4 blocks and 4 processors are as follows: $20000s$, $15000s$ and $3400s$. For usual affine, block wise noising sequential and block wise noising parallel, respectively. While the execution time starts to increase when the number of blocks exceeds the number of processors (8 blocks for 4 processors, with each processor calculating 2 blocks). For instance, the CNN-CIFAR model grep execution time is $4400s$.

Model	4 Blocks and 4 Processors			8 Blocks and 8 Processors		
	Sequential	Parallel	Speed Up	Sequential	Parallel	Speed Up
CNN-CIFAR	15000s	3800s	3.9	16000s	2200s	7.2
CNN-MNIST	4300s	1400s	3.1	5300s	950s	5.5
FC-MNIST	520s	140s	3.7	476s	65s	7.3
CNN-FASHION-MNIST	3960s	1000s	3.9	4480s	740s	6
FC-FASHION-MNIST	5700s	1400s	4	5080s	680s	7.4

TABLE I: Speed Up results.

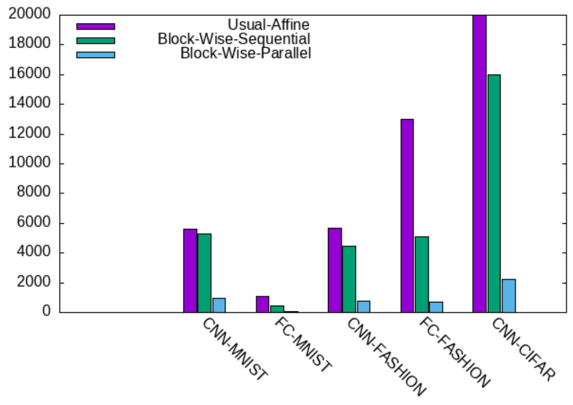

Fig. 17: Execution time of Block-Wise Noising parallelism: 8 blocks and 8 processors.

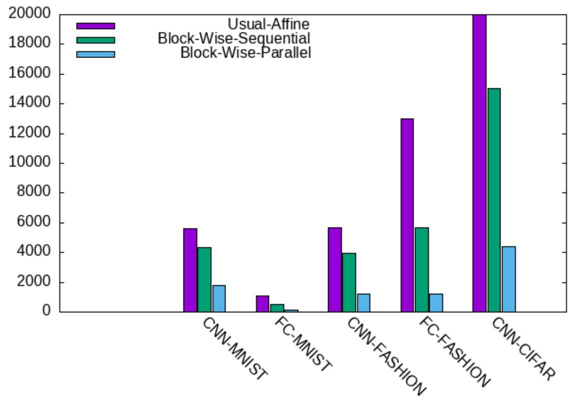

Fig. 18: Execution time of Block-Wise Noising parallelism: 8 blocks and 4 processors.

B. Speedup analysis

In order to evaluate the block-wise noising parallel performance in this section, we compute the speedup [25], which represents a correlation between the block-wise noising serial execution time and the parallel execution time in p processors ($p = 4$ and $p = 8$). Alternatively, we calculate:

$$Speed\ Up = \frac{T_s}{T_p}.$$

The Speed up are given in Table I. We note that Block-wise-noising parallel has been found to achieve a speedup of $3.1-4$ for $p = 4$ and $5.5 - 7.4$ for $p = 8$.

It should be noted that the speedups are significant, and that block-wise noising is of major interest for neural network verification.

VII. Conclusion

In summary, we developed both a sequantial and parallelized version of block-wise noising for the analysis of neural networks by affine forms. We evaluated the effictivness of block wise noising on execution time of various types of NNs. Our results demonstrate that block wise noising is faster compared to usual affine forms, which divide the execution time by two in the majority of the case.

Several perspectives can be considered for this work. First, concerning the experiments, it would be interesting to test block-wise noising with large neural networks that have billions of parameters, such as VGG-16 and AlexNet, to determine the limits of our method. Second, regarding activation functions, it would be intersting to explore the abstract transformations for other types of activation functions, such as sigmoid and tanh. This would allow us to broaden our understanding of the applicability of our approach. Third, it would be valuable to compare the block-wise noising method with our prior work on compressed affine forms [14] to assess the performance and limitations of each approach. This comparative analysis would provide insights into the strengths and weaknesses of both methods. Finally, it would be interesting to investigate if the block-wise noising approach proposed in this paper can be adapted to other techniques for validation and explainability of NN such as the approaches based on automatic differentiation.

References

[1] LAUGHLIN, Simon B. et SEJNOWSKI, Terrence J. ”Communication in neuronal networks. Science”, 2003, vol. 301, no 5641, p. 1870-1874.

[2] ALPAYDIN, Ethem.”Machine learning: the new AI”. MIT press, 2016.

[3] ZHANG, Zhaodi, LIU, Jing, LIU, Guanjun, et al. ”Robustness verification of swish neural networks embedded in autonomous driving systems”. IEEE Transactions on Computational Social Systems, 2022.

[4] BOJARSKI, Mariusz, DEL TESTA, Davide, DWORAKOWSKI, Daniel, et al. ”End to end learning

for self-driving cars". arXiv preprint arXiv:1604.07316, 2016.

[5] NARODYTSKA, Nina et KASIVISWANATHAN, Shiva Prasad. "Simple Black-Box Adversarial Attacks on Deep Neural Networks". In : CVPR Workshops. 2017. p. 2.

[6] MIRMAN, Matthew, GEHR, Timon, et VECHEV, Martin. "Differentiable abstract interpretation for provably robust neural networks". In : International Conference on Machine Learning. PMLR, 2018. p. 3578-3586.

[7] GEHR, Timon, MIRMAN, Matthew, DRACHSLER-COHEN, Dana, et al. "Ai2: Safety and robustness certification of neural networks with abstract interpretation". In : Security and Privacy (SP). IEEE, 2018. p. 3-18.

[8] KATZ, Guy, BARRETT, Clark, DILL, David L., et al. "Reluplex: An efficient SMT solver for verifying deep neural networks". In : Computer Aided Verification. Springer, 2017. p. 97-117.

[9] KATZ, Guy, HUANG, Derek A., IBELING, Duligur, et al. "The marabou framework for verification and analysis of deep neural networks". In : Computer Aided Verification. Springer, 2019. p. 443-452.

[10] TIAN, Yuchi, PEI, Kexin, JANA, Suman, et al. "Deeptest: Automated testing of deep-neural-network-driven autonomous cars". In: software engineering. 2018. p. 303-314.

[11] HE, Kaiming, ZHANG, Xiangyu, REN, Shaoqing, et al. "Deep residual learning for image recognition". In: IEEE computer vision and pattern recognition. 2016. p. 770-778.

[12] ALOM, Md Zahangir, TAHA, Tarek M., YAKOPCIC, Christopher, et al. "The history began from alexnet: A comprehensive survey on deep learning approaches". arXiv preprint arXiv:1803.01164, 2018.

[13] QASSIM, Hussam, VERMA, Abhishek, et FEINZIMER, David. "Compressed residual-VGG16 CNN model for big data places image recognition". In: computing and communication workshop and conference (CCWC). IEEE, 2018. p. 169-175.

[14] SOUALAH, Asma et MARTEL, Matthieu. "Efficient Neural Network Validation with Affine Forms". In : System Reliability and Safety (ICSRS). IEEE, 2022. p. 179-185.

[15] MESSINE, Frédéric et TOUHAMI, Ahmed. "A general reliable quadratic form: An extension of affine arithmetic". Reliable Computing, 2006, vol. 12, no 3, p. 171-192.

[16] O'SHEA, Keiron et NASH, Ryan. "An introduction to convolutional neural networks". arXiv preprint arXiv:1511.08458, 2015.

[17] ALBAWI, Saad, BAYAT, Oguz, AL-AZAWI, Saad, et al. "Social touch gesture recognition using convolutional neural network. Computational Intelligence and Neuroscience", 2017, vol. 2018.

[18] COUSOT, Patrick et COUSOT, Radhia. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In : ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1977. p. 238-252.

[19] SINGH, Gagandeep, GEHR, Timon, MIRMAN, Matthew, et al. "Fast and effective robustness certification". Advances in neural information processing systems, 2018, vol. 31.

[20] DE FIGUEIREDO, Luiz Henrique et STOLFI, Jorge. "Affine arithmetic: concepts and applications". Numerical Algorithms, 2004, vol. 37, p. 147-158.

[21] LeCun, Yann, et al. "Gradient-based learning applied to document recognition". IEEE 86.11 (1998): 2278-2324.

[22] Krizhevsky, Alex and Hinton, Geoffrey and others, "Learning multiple layers of features from tiny images". Toronto, ON, Canada, 2009

[23] LI, Shen, ZHAO, Yanli, VARMA, Rohan, et al. "Pytorch distributed: Experiences on accelerating data parallel training". arXiv:2006.15704, 2020.

[24] SHOEYBI, Mohammad, PATWARY, Mostofa, PURI, Raul, et al. "Megatron-lm: Training multi-billion parameter language models using model parallelism". arXiv preprint arXiv:1909.08053, 2019.

[25] VENKATESH, Eduru Harindra, VIVEK, Yelleti, RAVI, Vadlamani, et al. "Parallel and Streaming Wavelet Neural Networks for Classification and Regression under Apache Spark". arXiv preprint arXiv:2209.03056, 2022.

Appendix

A. Neural Networks Evaluated

We test with five trained networks which are described below:

Fashion-MNIST

- CNN
  $\mathrm{Input}(28 \times 28 \times 1) \rightarrow \mathrm{Conv}(32 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{MaxPool}(2 \times 2) \rightarrow \mathrm{Conv}(64 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{MaxPool}(2 \times 2) \rightarrow \mathrm{Flatten} \rightarrow \mathrm{Fc}(256) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(10)$
- FC
  $\mathrm{Input}(28 \times 28 \times 1) \rightarrow \mathrm{Flatten} \rightarrow \mathrm{Fc}(512) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(256) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(128) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(64) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(10) \rightarrow \mathrm{y}$

MNIST

- CNN
  $\mathrm{Input}(28 \times 28 \times 1) \rightarrow \mathrm{Conv}(64 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{MaxPool}(2 \times 2) \rightarrow \mathrm{Conv}(64 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{MaxPool}(2 \times 2) \rightarrow \mathrm{Conv}(64 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Flatten} \rightarrow \mathrm{Fc}(128) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(64) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(10) \rightarrow \mathrm{y}$
- FC
  $\mathrm{Input}(28 \times 28 \times 1) \rightarrow \mathrm{Flatten} \rightarrow \mathrm{Fc}(128) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(64) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(32) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(10) \rightarrow \mathrm{y}$

Cifar-10

- CNN
  $\mathrm{Input}(32 \times 32 \times 1) \rightarrow \mathrm{Conv}(32 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Conv}(32 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{MaxPool}(2 \times 2) \rightarrow \mathrm{Conv}(32 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Conv}(32 \times 3 \times 3) \rightarrow \mathrm{Relu} \rightarrow \mathrm{MaxPool}(2 \times 2) \rightarrow \mathrm{Flatten} \rightarrow \mathrm{Fc}(512) \rightarrow \mathrm{Relu} \rightarrow \mathrm{Fc}(10) \rightarrow \mathrm{y}$