Scaling up the static analysis of neural networks using affine forms

Asma Soualah

▶ To cite this version:

Asma Soualah. Scaling up the static analysis of neural networks using affine forms. Artificial Intelligence [cs.AI]. Université de Perpignan, 2024. English. NNT : 2024PERP0038 . tel-04959207

HAL Id: tel-04959207
https://theses.hal.science/tel-04959207v1

Submitted on 20 Feb 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE
Pour obtenir le grade de
Docteur

Délivrée par
UNIVERSITE DE PERPIGNAN VIA DOMITIA

Préparée au sein de l'école doctorale **ED305**
Et de l'unité de recherche **LAMPS**

Spécialité : **Informatique**

Présentée par

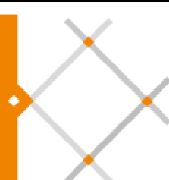
Asma SOUALAH

TITRE DE LA THÈSE

Scaling Up the Static Analysis of Neural Networks Using Affine Forms

Soutenue le 21 Octobre 2024 devant le jury composé de

Mme. **Claire Pagetti,** Directrice de recherche, ONERA Toulouse	Rapportrice
Mr. **Thibault Hilaire,** Maître de conférences, Sorbonne Université	Rapporteur
Mr. **Khalil Ghorbal,** Chargé de recherche, INRIA Rennes	Examinateur
Mr. **Mikael Barboteu,** Professeur, Université de Perpignan	Examinateur
Mr. **Matthieu MARTEL,** Professeur, Université de Perpignan	Directeur
Mr. **Arnault Ioualalen,** CEO et R&D, Numalis	Co-Directeur



Acknowledgements

Je tiens tout d'abord à exprimer ma profonde gratitude à mon directeur de thèse, Matthieu Martel, pour son accompagnement et son soutien tout au long de ces cinq années. Sa patience, ses conseils avisés et son enseignement m'ont permis d'acquérir des compétences essentielles tant sur le plan scientifique que personnel. Il m'a appris énormément et a toujours été disponible pour m'encourager et me guider dans les moments difficiles. Merci, Matthieu, pour votre dévouement et vos efforts constants.

Je remercie également chaleureusement mon co-directeur de thèse, Arnault Ioualalen, dont les conseils ont été précieux. Bien que nos rencontres aient été plus rares en raison de ses nombreuses obligations, chacune de ses interventions a toujours été d'une grande pertinence et a contribué à l'avancement de mon travail.

Je tiens à remercier mes rapporteurs, Claire Pagetti et Thibault Hilaire, pour le temps qu'ils ont consacré à la relecture et à l'évaluation de mon manuscrit. Leurs remarques constructives ont grandement amélioré la qualité de ce travail. Un grand merci également à Mikael Barboteu et Khalil Ghorbal d'avoir accepté de faire partie de mon jury de thèse, j'apprécie sincèrement leur engagement.

Je ne saurais oublier le LAMPS, mon laboratoire, et son directeur, pour m'avoir accueillie durant toutes ces années de thèse. Je tiens à remercier particulièrement Sylvia, Michel et Joëlle, pour leur aide inestimable. Votre soutien a été précieux, et je n'oublierai jamais tout ce que vous avez fait pour moi.

Je tiens aussi à exprimer ma reconnaissance à mes collègues de bureau, Mélanie et Thung . Ces années passées à partager le travail, les joies et parfois les larmes resteront gravées dans ma mémoire. Merci à vous deux pour ces moments inoubliables. Ensemble, nous avons créé une véritable solidarité. Girls power !

Enfin, je remercie ma mère et mon frère Islam, sans qui je ne serais pas là aujourd'hui. Leur soutien indéfectible m'a permis de surmonter bien des épreuves, et je leur dois une grande partie de ma réussite. Merci à eux du fond du cœur.

Un immense merci à Chloé, une amie exceptionnelle, qui a été à mes côtés tout au long de cette aventure. Merci d'avoir été cette oreille attentive, chaque jour, prête à écouter mes doutes, mes craintes, mais aussi mes petites victoires.

Ce travail est dédié à mon père, que je n'ai malheureusement pas eu la chance de connaître, car il est parti bien trop tôt. Même si nous n'avons pas partagé de moments ensemble, je souhaite lui rendre hommage à travers ce travail. J'espère que, là où il est, il serait fier de la personne que je suis devenue.

Contents

Acknowledgements **i**

List of Figures **viii**

List of Table **ix**

1 Introduction **1**
1.1 Context and Motivation 1
1.2 Contributions 4
1.2.1 Contribution 1: Development of the Affine Compressed Approach for Neural Networks Analysis 4
1.2.2 Contribution 2 : Scaling-Up the Analysis of Neural Networks by Affine Forms 5
1.2.3 Contribution 3: Mixed Approach 5
1.2.4 Contribution 4: A Static Analysis Tool for Neural Networks 5
1.3 Structure of the Thesis 6
1.4 List of Published Work 7

I Background and Related Work **9**

2 Overview of Neural Networks **11**
2.1 Architecture of Neural Networks 12
2.1.1 Fully Connected Neural Networks 12
2.1.2 Convolutional Neural Networks 14
2.1.3 Recurrent Neural Networks (RNN) 16
2.1.4 Activation Functions 17
2.2 Formal Method and Tools for the Verification of Neural Networks 19
2.2.1 Parameters Influencing the Design of Formal Methods 20
2.2.2 Complete Formal Method 22
2.2.3 Incomplete Formal Method 25

3 Numerical Abstract Domains **31**
3.1 Abstract Domains 32
3.2 Interval Arithemtic 34
3.3 Affine Forms 34
3.3.1 Notations 34
3.3.2 Elementary Operations Among Affine Forms 36
3.3.3 Extensions of Affine Forms 37
3.4 Abstract Transformer of Activation Functions 40
3.5 Neural Network Verification 42
3.6 AFFapy 44
3.6.1 Overview 44
3.6.2 Elementary Arithmetic Functions with Affapy 44
3.7 Conclusion 46

II Contribution **47**

4 Efficient Neural Network Validation with Affine Forms **49**
4.1 Compressed Affine Forms . . . 50
4.1.1 Principe . . . 50
4.1.2 Abstract Domain of Compressed Affine Forms . . . 52
4.2 Validating Neural Networks with Affine Forms . . . 54
4.2.1 Experimental Setting . . . 54
4.2.2 Influence of Merging Symbols on Accuracy . . . 56
4.2.3 Influence of Merging Symbols on Time . . . 57
4.2.4 Case Study of Convolutional Neural Networks . . . 58
4.3 Local Noise . . . 60
4.3.1 Accuracy in Function of Local Noise . . . 60
4.3.2 Execution Time in Function of Local Noise . . . 61
4.4 Conclusion . . . 62

5 Scaling-up the Analysis of Neural Networks by Affine Forms: A Block-Wise Noising Approach **65**
5.1 Overview . . . 66
5.1.1 Activation Functions . . . 67
5.2 Block-wise Noising . . . 68
5.3 Efficiency of Block-Wise Noising . . . 71
5.3.1 Experimental Setting . . . 71
5.3.2 Execution Time . . . 72
5.3.3 Experimenting with Trained Neural Networks . . . 74
5.4 Block-Wise Noising Parallelism . . . 75
5.4.1 Execution Time . . . 77
5.4.2 Speedup Analysis . . . 78
5.5 Conclusion . . . 78

6 The NNaff Tool **79**
6.1 Mixed Affine Approach . . . 80
6.1.1 Principle of The NNaff Tool . . . 80
6.2 The NNaff Tool . . . 82
6.2.1 Architecture of The Tool . . . 82
6.2.2 Using the Tool . . . 83
6.2.3 Experimental Results . . . 85
6.3 Summary . . . 87

7 Conclusion **89**
7.1 Summary of key Results . . . 89
7.2 Perspectives . . . 91
7.2.1 Enhanced Compressed Affine Forms . . . 91
7.2.2 Scalability: Extending to More Complex Networks and Activation Functions . . . 92
7.2.3 Parallelization and GPU Utilization . . . 92

III Résumé étendu à l'intention du lecteur francophone 95

8 Analyse Statique de réseaux neurones 97
8.1 Introduction . . . 97
8.2 Aperçu des réseaux de neurones . . . 100
8.2.1 L'architecture d'un réseau de neurones . . . 100
8.2.2 Les types des réseaux neurones . . . 101
8.2.3 Méthodes pour vérifier les réseaux de neurones . . . 102
8.3 Formes affines . . . 103
8.3.1 Notation . . . 103
8.3.2 Les opérations élémentaires . . . 103
8.4 Forme affine compressée . . . 105
8.4.1 Principe . . . 105
8.4.2 Résultats expérimentaux . . . 107
8.5 Bruit par bloc . . . 109
8.5.1 Principe de bruit par bloc . . . 110
8.5.2 Fonction d'activation . . . 112
8.5.3 Résultats expérimentaux . . . 112
8.6 L'outil NNAFF . . . 114
8.6.1 L'utilisation de l'outil . . . 115
8.6.2 Résultats expérimentaux . . . 116
8.7 Conclusion . . . 118

Bibliography 132

Abstract 133

Appendix 133
A.1 Neural Networks Evaluated . . . 134
A.2 Loading and saving Images with Pickle library . . . 138

List of Figures

1.1 Increasing Scale of Neural Networks Through the Years (photo credits from [13]). . . . 2
1.2 Architecture of a Neural Networks (photo credits from: https://slideplayer.com/slide/17005812/) . . . 3

2.1 Architecture of a Neural Network. . . . 12
2.2 Fully Connected Neural Network with 5 Layers. . . . 13
2.3 History of the Convolutional Neural Network. . . . 14
2.4 An example of a convolution operation with (padding = same, stride = 1) and (padding = valid, stride = 1). . . . 16
2.5 An example of the architecture of CNN (VGG16). . . . 16
2.6 An example of the architecture of RNN. . . . 17
2.7 Visualizing Common Activation Functions: (Sigmoid, Tanh, ReLU, and Softmax). . . . 18

3.1 Examples of Abstract Domains 33
3.2 The geometrical concretisation $\gamma(B)$. . . . 36
3.3 Interval approximations for the ReLU function (photo credits from [4]). . . . 41
3.4 Convex approximations for the ReLU function. In the figure, $\lambda = \frac{u_i}{u_i - l_i}$ and $\mu = \frac{-l_i u_i}{u_i - l_i}$ (photo credits from [105]). . . . 42
3.5 Examples of Propagation of Intervals in a Neural Network Across Multiple Layers 43
3.6 Examples of Propagation of affine forms in a Neural Network Across Multiple Layers 43

4.1 Images used to test the efficiency of affine forms. . . . 54
4.2 Propagation of the affine forms corresponding to our set of images throughout a fully connected neural network with 4 and 6 layers. Input images have size 12×12 (left graph) and 18×18 (right graph). . . . 56
4.3 Execution time in function of the number of noise symbols. Input images have size 18×18 with 4 layers (leftmost graph) and 6 layers (rightmost graph). We use the six images of Figure 4.1. . . . 58
4.4 The architecture of Model-CNN. . . . 58
4.5 Propagation of the affine forms corresponding to our set of images throughout a convolutional neural network. Widths of the concretizations (top row) and execution times (bottom row). . . . 59
4.6 The upper left and lower right regions of the Sandal image. . . . 61
4.7 Propagation of the affine forms corresponding to our set of images throughout a fully connected neural network with 4 and 6 layers. Input images have size 24×24. . . . 62
4.8 Execution time in function of the number of layers. Input images have size 24×24. We consider the four images of Figure 4.1. . . . 63

5.1 Fully connected layer operations. . . . 66
5.2 Block 1 67

5.3 Block 2 67
5.4 Splitting of x into two block-wise noising. 67
5.5 The zonotope approximation of the Relu function. 67
5.6 Splitting input into block-wise noising. 69
5.7 Image used to test the efficiency of block-wise noising. 71
5.8 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 5.6. 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 5.7. 73
5.9 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 5.6. Input image has a size 28×28 with 4, 5 and 6 layers for the convolutional neural network. We use the image of Figure 5.7. 73
5.10 Block-Wise Noising Parallelism. 76
5.11 Execution time of Block-Wise Noising parallelism. 77

6.1 Mixed affine approach for $n = 4$. 80
6.2 Functional architecture of the NNaff tool. 84

7.1 Neural Network Models Analyzed by Usual Affine Forms and NNaff. . . . 91

8.1 l'architecture d'un réseau neuronal. 100
8.2 Images utilisées pour tester l'efficacité des formes affines. 107
8.3 Propagation des formes affines correspondant à notre ensemble d'images. 108
8.4 Temps d'exécution en fonction du nombre de symboles de bruit. 108
8.5 Diviser l'entrée en 4 blocs. 109
8.6 Opérations de la couche entièrement connectée. 110
8.7 Bloc 1 111
8.8 Bloc 2 111
8.9 La divison de x en 2 blocs. 111
8.10 L'architecture de notre outil NNaff. 114

List of Tables

2.1 Neural Network Verification Tools: Essential Overview 29

4.1 Elementary operations between affine forms and compressed affine forms and their concretizations. 51

5.1 Execution time for dataset MNIST with fully connected neural networks at 2 layers. 72
5.2 configurations of the model used in our experiments. 74
5.3 Execution time with fully connected and convolutional neural networks at different depths. 75
5.4 Speed-Up results. 78

6.1 Neural Network Architectures used for Evaluating the Performance of the NNaff Tool. 85
6.2 Execution Time with Different Models. 86

8.1 Opérations élémentaires entre formes affines standard et formes affines compressées et leurs concrétisations. 106
8.2 Configurations des modèles utilisées dans nos expériences. 113
8.3 Temps d'exécution comparés entre réseaux entièrement connectés et convolutionnels à différentes profondeurs. 113
8.4 Architectures de réseaux de neurones utilisées pour évaluer la performance de NNaff. 117
8.5 Temps d'exécution de différents modèles. 118

Chapter 1

Introduction

1.1 Context and Motivation 1
1.2 Contributions 4
1.2.1 Contribution 1: Development of the Affine Compressed Approach for Neural Networks Analysis 4
1.2.2 Contribution 2 : Scaling-Up the Analysis of Neural Networks by Affine Forms 5
1.2.3 Contribution 3: Mixed Approach 5
1.2.4 Contribution 4: A Static Analysis Tool for Neural Networks 5
1.3 Structure of the Thesis 6
1.4 List of Published Work 7

1.1 Context and Motivation

Neural networks [66, 6] have demonstrated tremendous success in solving complex problems across various domains, including computer vision [122], image classification [61], and natural language processing [81]. Nowadays, neural networks are increasingly being employed in mission-critical systems [19]. For instance, in self-driving cars [34], neural networks are used for tasks such as object detection, lane keeping, and decision-making processes to ensure safe and efficient navigation. Similarly, in health monitoring systems [126], neural networks are applied to analyze medical data, predict patient outcomes, and assist in early diagnosis, thereby improving patient care and treatment outcomes. Additionally, neural networks are employed in financial systems for fraud detection, etc. However, despite their widespread application and success, the deployment of neural networks in these critical systems introduces significant challenges [84, 127]. Neural networks are frequently built using training datasets that are biased or incomplete, which can lead to unreliable or unsafe outcomes. For example, if a neural network in a self-driving car misinterprets a stop sign due to biased training data, it could result in a failure to stop, potentially causing a se-

rious accident. Therefore, it is crucial to ensure the safety of these neural networks before their deployment. Safety, in this context, refers to a general category of correctness properties stipulating that a neural networks should not reach an undesirable state. It denotes to the ability of a neural networks to operate without causing unacceptable risks of harm or damage, even in unexpected situations or when faced with incomplete or biased data. For example, in the case of neural networks designed for an aircraft collision avoidance system [62], one of the safety properties we want to prove is that if an aircraft is approaching from the left, the neural network should instruct the aircraft to turn right.

The key challenge today is the design of verifiers that can scale the large neural networks and maintain sufficient precision to provide the safety of these networks. As illustrated in Figure 1.1, the size of neural networks has exploded over time, making this task increasingly delicate. Several methods have been developed to address this issue and automatically prove that neural networks work correctly in all possible cases [123, 41]. These methods check the safety of the network before deployment. There are two main categories of methods: complete formal techniques [62, 63, 35, 111, 91] and incomplete formal methods [38, 104, 105, 16, 119]. Complete formal techniques target smaller-sized neural networks with high accuracy. These techniques are sound and complete; they tell us exactly whether a given property holds or not for the neural networks. In contrast, incomplete formal methods can verify large-scale networks in a few minutes, compute an overapproximation of the network output corresponding to the input but this overapproximation lead to false positives, where the method incorrectly tags a safe case as unsafe. Additionally, there are methods [103, 116] that integrate both complete and incomplete techniques aiming to be more scalable than complete methods while improving the precision of incomplete methods.

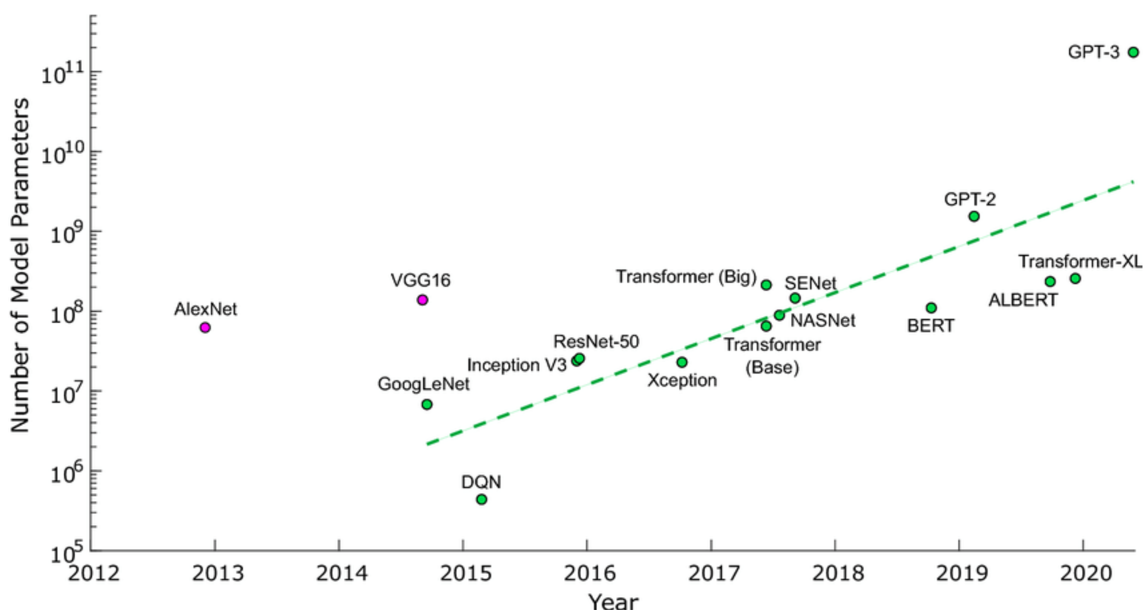


Figure 1.1: Increasing Scale of Neural Networks Through the Years (photo credits from [13]).

In this thesis, we will consider abstract interpretation methods [26], which are a type

of incomplete formal method. The basic idea of abstract methods is to use an abstraction domain, in our case, affine forms [31], to over-approximate computations on inputs. The decision to employ these forms is not accidental, but rather motivated by the fact that they preserve relationships between variables, and since neural networks operate mostly through affine computations like fully connected or convolutional layers, affine forms are precise. However, an important drawback of using affine forms in the context of neural networks is that they are not exact for nonlinear functions, such as some activation functions (ReLU, $tanh, softmax$, etc,.), and they are time and memory-consuming, while at the same time, neural networks become larger and larger as despicted in Figure 1.1. Complex tasks require deep neural network models with a large number of parameters that have to be trained (see Figure 1.2). For example, Resnet-50 [48] requires 25.5 million parameters, AlexNet [64] 61 million parameters, VGG-16 [86] 138 parameters and GPT-3 175 milliard parameters for training. Verifying these neural networks using abstract interpretations, such as affine forms, presents the challenge of balancing scalability and precision. Consequently, incorporating affine forms in the verification of neural networks becomes a computationally intensive process that may take several days or even months 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 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, and then the running neural networks becomes more complex and computationally intensive).

Metrics	LeNet-5	AlexNet	VGG-16	GoogLeNet (v1)	ResNet-50
Top-5 error	n/a	16.4	7.4	6.7	5.3
Input Size	28x28	227x227	224x224	224x224	224x224
# of CONV Layers	**2**	**5**	**16**	**21 (depth)**	**49**
Filter Sizes	5	3, 5,11	3	1, 3 , 5, 7	1, 3, 7
# of Channels	1, 6	3 - 256	3 - 512	3 - 1024	3 - 2048
# of Filters	6, 16	96 - 384	64 - 512	64 - 384	64 - 2048
Stride	1	1, 4	1	1, 2	1, 2
# of Weights	2.6k	2.3M	14.7M	6.0M	23.5M
# of MACs	283k	666M	15.3G	1.43G	3.86G
# of FC layers	**2**	**3**	**3**	**1**	**1**
# of Weights	58k	58.6M	124M	1M	2M
# of MACs	58k	58.6M	124M	1M	2M
Total Weights	**60k**	**61M**	**138M**	**7M**	**25.5M**
Total MACs	**341k**	**724M**	**15.5G**	**1.43G**	**3.9G**

Figure 1.2: Architecture of a Neural Networks (photo credits from: https://slideplayer.com/slide/17005812/)

The aim of this thesis is to develop approaches based on the same principle as affine forms, which allow us to find a compromise between accuracy and execution time compared with standard affine forms. We aim to retain maximum accuracy while significantly reducing computation time. Using these methods, we hope to make neural networks veri-

fication more efficient and practical for large-scale applications while maintaining the robustness and reliability of the models analyzed.

1.2 Contributions

The work carried out during this thesis led to the contributions detailed below in the context of static analysis by abstract interpretation of neural networks.

1.2.1 Contribution 1: Development of the Affine Compressed Approach for Neural Networks Analysis

In this contribution, we introduce a novel approach called Affine Compressed [106] for the static analysis of neural networks using abstract interpretation with more precise affine forms. This approach is based on the same principle as traditional affine forms [31] but incorporates a mechanism to reduce the number of noise symbols, addressing a significant challenge in verifying large neural networks. Traditional affine forms are often employed in the static analysis of neural networks due to their ability to represent linear relationships between variables. However, when applied to large neural networks, these forms generate numerous noise symbols, which can lead to excessive computational time and resource consumption. This inefficiency makes it challenging to verify the properties of large-scale models within a reasonable timeframe. To mitigate this problem, we propose the Affine Compressed approach, which abstracts the usual affine forms by successively compressing k noise symbols into one new symbol. The primary steps and advantages of this approach are outlined as follows:

- **Noise Symbol Compression**: Instead of representing noise with numerous individual symbols, our approach merge several noise symbols into a new single noise symbol. This compression reduces the overall complexity of the analysis without compromising the fundamental relationships captured by the traditional affine forms.
- **Impact on Precision and Execution Time**: By compressing noise symbols, we achieve a balance between precision and computational efficiency. Although this abstraction may slightly affect the precision of the analysis, it significantly reduces the execution time, making it feasible to analyze larger neural networks.
- **Local Noise Study**: To understand the impact of noise symbols on the analysis, we conducted a detailed study of local noise behavior. This study helps to know what happens if some part of the model is perturbed and also which part of the model introduces the overapproximation.

We evaluate the effectiveness of the Affine Compressed approach through a comparative study in terms of both precision and execution time against traditional affine forms and interval analysis.

1.2.2 Contribution 2 : Scaling-Up the Analysis of Neural Networks by Affine Forms

The second contribution focus on developing a novel approach to the analysis of neural networks by affine forms, called Block-Wise Noising [107]. This method addresses the limitations of standard affine forms by simulating real situations where certain parts of the network are perturbed, building on the local noise study from the first contribution. The key advantage of Block-wise Noising compared to the first contribution is that it preserves all relationships inherent in standard affine forms. This means that even though the noise is divided into blocks, the overall precision of the analysis remains intact in linear cases. As part of this contribution, we also give an abstraction for ReLU, one of the most common activation functions for neural networks. We need this approximation for analyzing networks with ReLU activation so that our method applies to a broad class of models. It is relevant to underline that this approach has been motivated by [104]. We base the effectiveness of the Block-Wise Noising procedure on a detailed series of experiments with the sequential and parallel analysis of many trained neural networks.

1.2.3 Contribution 3: Mixed Approach

The third contribution consists of the integration of our prior contributions—Affine Compressed and Block-Wise Noising in the same method called Mixed Approach. This integration is motivated by their complementary advantages. While Affine Compressed significantly reduces computational resources, it may introduce some precision loss. Conversely, block-wise noising maintains precision but requires more computational resources. By combining these methods, we seek to achieve a balance that optimizes both execution time and verification accuracy.

1.2.4 Contribution 4: A Static Analysis Tool for Neural Networks

The automated tool called NNaff, which is short for Neural Networks Affine Forms, has been built to implement the methods described in Contributions 1, 2, and 3. This tool serves as an aid to static analysis of neural networks models. Details of the main hierarchy and workflow are given below:

- **Model Selection**: The user chooses the particular neural networks model that he or she needs to analyze using NNaff. There are various formats for specifying this model that can be used with the tool.
- **Method Selection:** The user chooses from the available methods, those suitable for his or her analysis. Each method requires some parameters set by the user, which can be adjusted to fit their objectives.
- **Analysis Execution:** At this point, NNaff applies the selected method with the parameters defined by the user. During this step, the chosen techniques of static analysis

are applied to a given neural network model.

- **Output Metrics and Performance Evaluation**: In turn, NNaff returns the execution time required by the method to analyze the model.

In order to test its usefulness and reliability on large-scale architectures like VGG16 and ResNet, among others, we utilized NNaff on different kinds of artificial intelligence models, including convolutional ones. These experiments demonstrate how well it performs with complex and large-sized structures, hence showing its relevance in real-life cases.

1.3 Structure of the Thesis

This thesis is mainly divided into two parts, each of which having several chapters. Obviously, these two parts are complemented by the present introduction and the conclusion of chapter 7.

Part 1: State of the Art

This section provides a general understanding of the state of the art regarding various concepts and aspects that are covered in other sections of this thesis. It has two chapters:

Chapter 2 gives a summary of neural networks, including their architectures, the different components that make them up, the methods used to secure these networks when used in critical systems, and the contributions made by each method in the verification of neural networks, which have become crucial today.

Chapter 3 discusses an existing technique for verifying neural networks called abstract interpretation, focused mainly on zonotopes (affine forms). We then define what these forms are and provide their basic operations, as well as some existing extensions of them aimed at dealing with issues related to standard affine forms. Finally, we conclude this chapter with a brief introduction to Affapy library, which implements all the basic operators of those forms in Python and will be highly useful for our work.

Part 2: Contribution

This section brings together the separate work carried out during this thesis within the static analysis by abstract interpretation of neural networks. It is composed of 3 chapters.

Chapter 4 presents the Affine Compressed approach, which is our first contribution in the context of static analysis by abstract interpretation of neural networks. We outline the main principles of this approach and demonstrate how it can be applied. Subsequently, we show the effectiveness of this approach with experimental results, comparing it with the typical affine and interval methods. We conclude this section by studying the impact of local noise symbols. This work has been published in [106].

Chapter 5 presents another method of verifying neural networks using affine forms, called block-wise noising. We emphasize the principles of this new approach and provide an abstract transformer for the ReLU activation function to model this non-linear function. A study of the sequential and parallel aspects of this approach is conducted in this chapter. This work has been published in [107].

Chapter 6 highlights our tool NNaff, detailing its architecture, operation, and performance, along with an introduction to a new mixed affine method that combines the two methods presented in chapters 4 and 5, respectively.

1.4 List of Published Work

The work that has been conducted in this dissertation led to the following publications:

- Soualah Asma, Matthieu Martel, and Stéphane Abide. **Scaling-up the Analysis of Neural Networks by Affine Forms: A Block-Wise Noising Approach**. In the 25th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). IEEE, 2023.
- Soualah Asma, and Matthieu Martel. **Efficient Neural Network Validation with Affine Forms**. In the 6th International Conference on System Reliability and Safety (ICSRS). IEEE, 2022.

Part I

Background and Related Work

Chapter 2

Overview of Neural Networks

2.1	Architecture of Neural Networks	12
2.1.1	Fully Connected Neural Networks	12
2.1.2	Convolutional Neural Networks	14
2.1.3	Recurrent Neural Networks (RNN)	16
2.1.4	Activation Functions	17
2.2	Formal Method and Tools for the Verification of Neural Networks	19
2.2.1	Parameters Influencing the Design of Formal Methods	20
2.2.2	Complete Formal Method	22
2.2.3	Incomplete Formal Method	25

Artificial intelligence (AI) [66, 6] is a thrilling computer science area focused on giving machines human-like thinking abilities. Earlier, scientists developed the concept of pre-neurons and examined interconnected neurons networks to perform logical and numerical functions [73]. However, things changed after some time as technology improved with powerful computing technologies and innovation became more pronounced. Nowadays, AI is being used in different fields such as medicine [126], finance [74, 87], robotics [11, 59], and transport [34, 19]. It is helpful in solving difficult tasks like medical diagnostics, predicting financial trends, automating industrial processes, designing autonomous vehicles etc. AI is changing very quickly, giving rise to new prospects and setting new limits on technological possibilities.

This chapter presents a foundation upon which to understand the ideas we present in our thesis. To begin with, we introduce neural networks in Section 2.1. Afterwards, we move to discussing basic concepts associated with activation functions for different types of artificial neural networks. Next, Section 2.2 examines current developments regarding formal methods employed when verifying neural networks which provide an insight into how they become reliable and accurate.

2.1 Architecture of Neural Networks

Artificial neural networks (ANN) [65] are the fundamental building blocks of deep learning algorithms. Sometimes referred to as simulated neural networks (SNN), these take their name and structure from the human brain, emulating the signals that organic neurons exchange with one another. They rely on the principle of training, which enables them to learn and improve accuracy over time. In this section, we explore their general architecture, how they work, and different types of these neural networks.

2.1.1 Fully Connected Neural Networks

An artificial neural networks [89] is made up of several layers of nodes, such as the input layer, followed by one or more hidden layers, and concluding with an output layer. A single node (refer to Figure 2.1) can be seen as taking as input a vector $X = (x_1 x_2, ... x_n) \in \mathbb{R}^n$ and is multiplying by the weight vector $W = (w_1, w_2, ... w_n)$ added to bias vector $b \in \mathbb{R}^n$ which gives the sum of weights

$$y = \sum_{i=1}^{n} x_i . w_i + b_i \quad . \tag{2.1}$$

Then it applies some function σ, called an activation function, to this sum. This function plays a crucial role in determining the output of a particular node. Depending on the specific activation function employed. For example, If the output value for an individual node exceeds a specific threshold value, then that node will be activated, sending information forward in the network to the next layer; otherwise, no data will be sent. We detail various kinds of activation functions in the next sections.

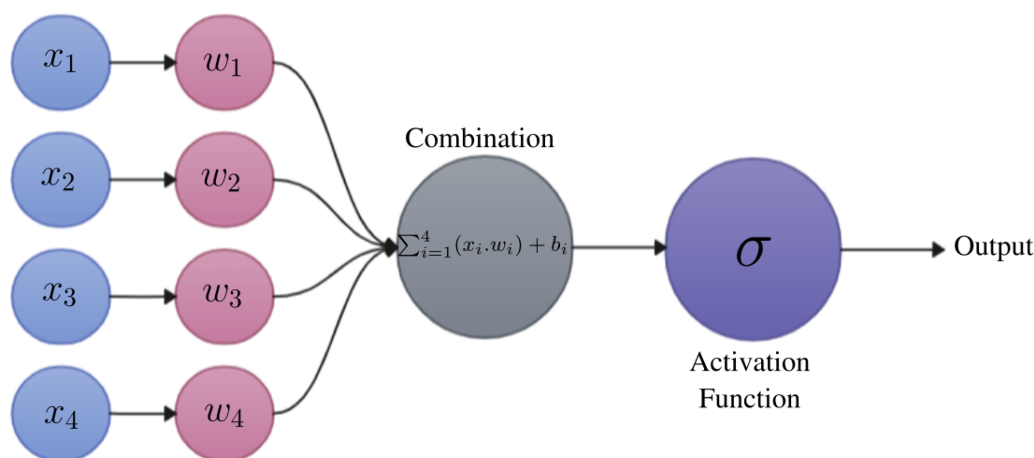


Figure 2.1: Architecture of a Neural Network.

A fully connected neural networks (FCN) [9], is a type of artificial neural network (ANN) with multiple hidden layers. Here, connections between nodes never form cycles, meaning

that information is processed from the input layer to the output layer only in one direction. Each new layer takes its inputs from previous layer outputs. Figure 2.2 shows one example of FNN architecture. The definition below summarizes the notion introduced above.

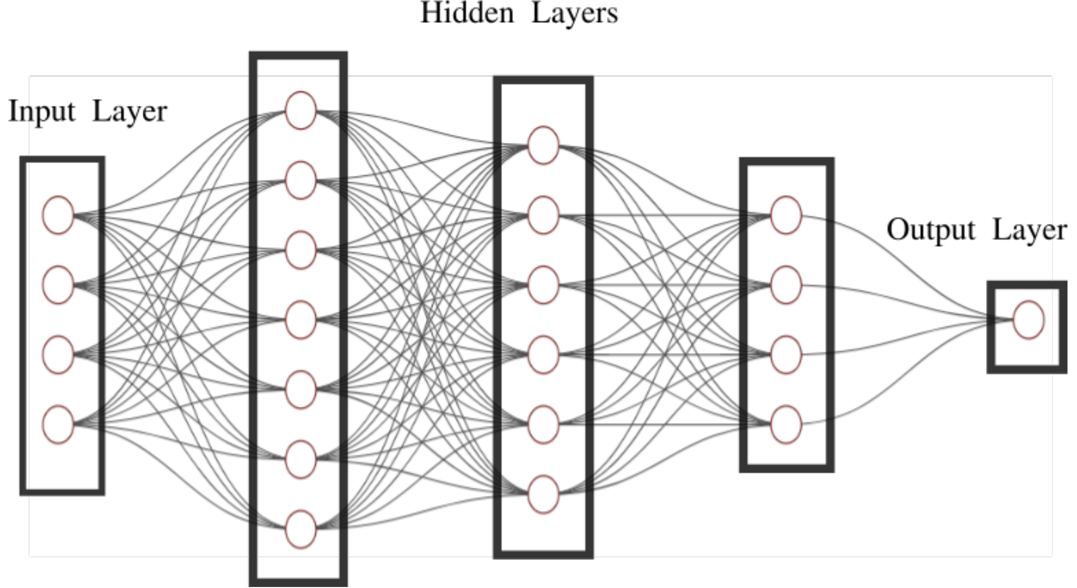


Figure 2.2: Fully Connected Neural Network with 5 Layers.

Definition 1 Let $\hat{x} \in \mathbb{R}^m$ be the input vector to a fully connected neural network. The output vector $\hat{y} \in \mathbb{R}^n$ is computed by applying successive affine transformations and activation functions across multiple layers. The computation for a single layer is given by:

$$\hat{y}_l = \sigma_l(W_l \hat{x}_l + \hat{b}_l) \tag{2.2}$$

For the entire network, the final output $\hat{y}$ is obtained after L layers as follows:

$$\hat{y} = \sigma_L\left(W_L \sigma_{L-1}\left(W_{L-1} \dots \sigma_1\left(W_1 \hat{x} + \hat{b}_1\right) + \hat{b}_{L-1}\right) + \hat{b}_L\right) \tag{2.3}$$

where L is the number of layers, σ_l is the activation function at layer l, W_l and $\hat{b}_l$ are the weights and biases of layer l, and $\hat{x}$ is the input to the network.

We can note that fully connected neural networks have shown good performance on tabular data [15], but when used on images [8] and other more complex deeper data, they failed due to the large number of parameters that needed for training. This complete connectivity, where all neurons in one layer are connected to each neuron in the next layer, leads to high computation time and memory consumption. Due to these constraints, other kinds of neural networks have been developed.

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [71] were developed to address the spatial dependence issue found in fully connected neural networks as described in the former section. They are designed to handle data in the form of multiple arrays. For instance, a color image is made up of three 2D arrays, each representing pixel intensities in one of the three color channels. CNNs are mostly built on the convolution principle, which we will dwell on in detail in the next section and that is used to extract features from data [72, 30, 3]. CNNs and fully connected networks are not the same since CNNs have a local connectivity feature where all neurons in one layer no longer connect with every neuron in the next layer, but only with a few. In addition, the weight sharing property [110] reduces the number of parameters by applying the same weight values to different inputs. These attributes make it one of the major kinds of neural networks popularly adopted in the field of deep learning. The first CNN was presented by Zhang [22] in 1989. Lecun [115], in 1990, became the first person to call his network LeNet "convolutional". It was designed for handwritten zip code recognition. Since then, several architectures have been proposed for CNNs [64, 102, 108, 57, 109, 54, 95, 125, 46]. We summarize some of them in Figure 2.3

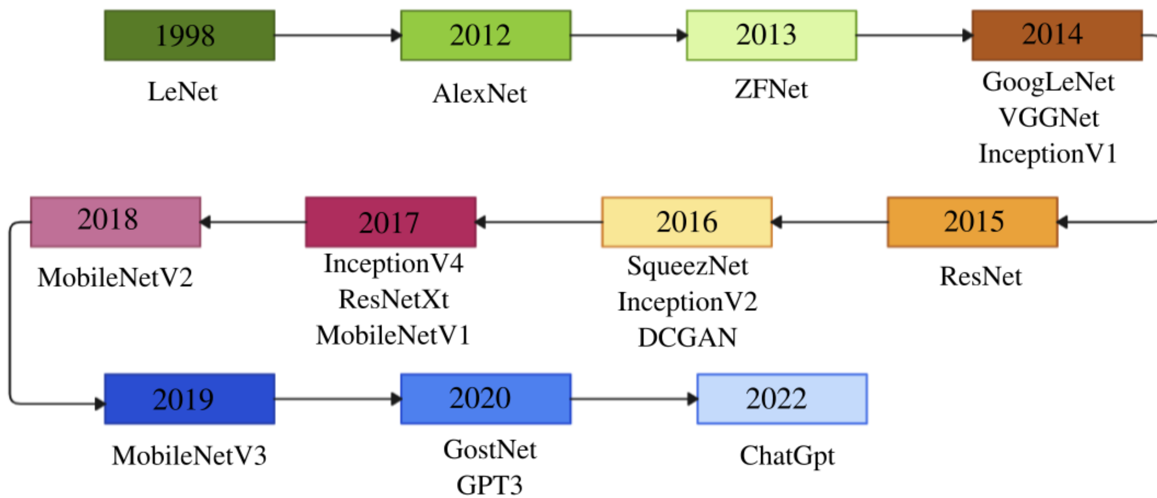


Figure 2.3: History of the Convolutional Neural Network.

A convolutional neural networks (CNN)[5] mostly consists of a convolutional layer, the core of a CNN. This layer is constructed based on the concept of convolution, which requires sets of weights, also called filters (Kernel), to move across input images and calculate the dot product of these values. The mathematical notation for this convolution operation is usually represented as:

$$y_{ij} = \sum_{i'=1}^{p} \sum_{j'=1}^{q} W_{i',j'} . x_{(i+i'-1),(j+j'-1)} + b \quad . \tag{2.4}$$

Here, $W \in \mathbb{R}^{p \times q}$ denotes the weight matrix, $x \in \mathbb{R}^{n \times m}$ stands for the input matrix, and $b \in \mathbb{R}$ is bias.

A convolutional layer is characterized by three hyperparameters:

- **Filter (Kernel)**: represent the weights of the layers that are used to detect specific features or patterns in the input data by sliding over it and performing convolution operations.
- **Stride**: denote the number of pixels by which the filter moves across the input data after each convolution operation. For instance, a stride of 1 means we move the filter one position to the right each time before calculating the output.
- **Padding**: is an operation that allows manipulation of the spatial dimensions of images by adding values (often zeros) around the edges of input images. This helps to maintain or adjust the height and width of images after applying filters, preventing the reduction in the size of feature maps at each convolutional layer and ensuring better utilization of the pixels located at the edges of the image.

Example 1 Let x be an input of size $4 \times 4 \times 1$, and f be a filter of size 2×2 with a stride of 1 and the valid padding ($Padding = 0$). We want to calculate the output shape of convolutional layers using the formula

$$\text{output_shape} = \left(\frac{Input_size - filter_size + 2 \times padding}{stride}\right) + 1 \ .$$

Substituting the given values, we obtain

$$\text{output_shape} = \left(\frac{4 - 2 + 2 \times 0}{1}\right) + 1 = \left(\frac{2}{1}\right) + 1 = 2 + 1 = 3 \ .$$

Therefore, the output size is 3×3.

Typically, following each convolutional layer, we find a pooling layer [97], which applies a pooling operation using a defined window shapes that move across all inputs based on a specific stride. This operation selects either the maximum value (MaxPooling)

$$f(x_1, ..., x_n) = max(x_1, ..., x_n) \ ,$$

or computes the average (AveragePooling)

$$f(x_1, ..., x_n) = \frac{1}{n}\sum_{i=1}^{n} x_i \ .$$

The main reason to employ such layers is to reduce the spatial dimensions of an output in order to improve the speed of computation and enhance feature robustness. It is worth noting that this layer is less complex compared to convolutional layers because it does not have trainable parameters.

After obtaining all features through convolutional layers, there is a fully connected or dense layer aimed at transforming these extracted features into a format suitable for being used in any classification task. As they work, each neuron within this layer forms linear combinations between itself and all neurons from previous layers while applying an activation function (which will be defined in Section 2.1.4). The number of neurons characterizes

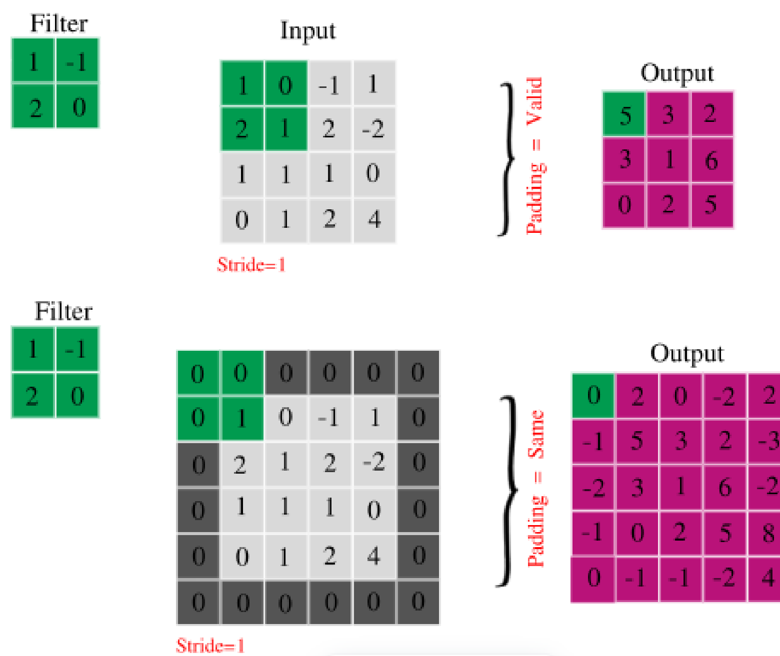


Figure 2.4: An example of a convolution operation with (padding = same, stride = 1) and (padding = valid, stride = 1).

this layer, thereby establishing how complex the model could be. It is worth mentioning that generally, this value may vary depending on how much data you have at your disposal and the complexity of the task. Figure 2.5 shows an example of the architecture of a CNN, specifically VGG16 [86], which is one of the well-known neural networks used in image classification.

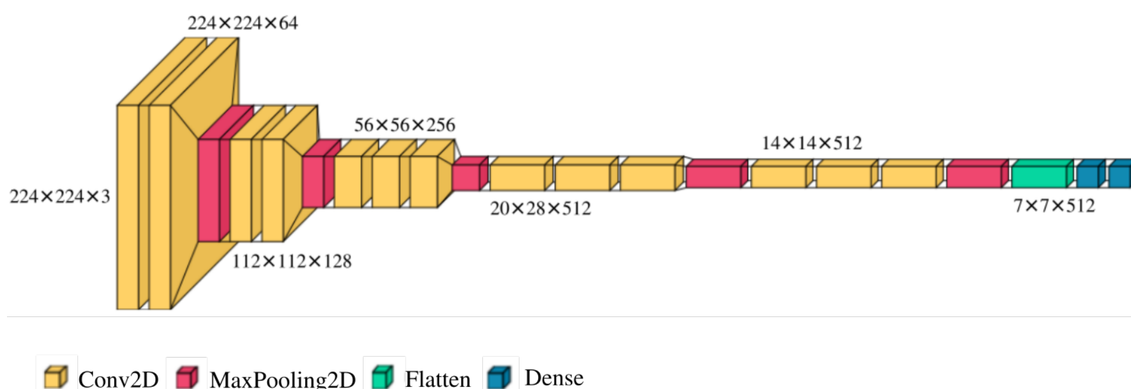


Figure 2.5: An example of the architecture of CNN (VGG16).

2.1.3 Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) [75, 98] are a type of deep neural network specifically designed for processing sequential or temporal data such as audio signals or text. The

architecture of an RNN is characterized by its ability to maintain a memory of previous inputs through time. As illustrated in Figure 2.6, an RNN typically consists of three main layers: Input layer $\mathbf{x}^{(T)} = (x^{(1)}, x^{(2)}, ..., x^{(T)})$, which represents a sequence of vectors at time step, hidden layer $\mathbf{h}^{(T)} = (h^{(1)}, h^{(2)}, ..., h^{(T)})$, which stores necessary information from previous time steps up to the current time step and output layer $y^{(T)} = (y^{(1)}, y^{(2)}, ..., y^{(T)})$ which is calculated using the input at time step t and the hidden state from the previous time step $(t-1)$. Mathematically, the definition of a RNN network can be given as follows:

$$\mathbf{h}^{(t)} = \sigma(\mathbf{W}_x\mathbf{x}^{(t)} + \mathbf{W}_h\mathbf{h}^{(t-1)} + \mathbf{b}_h), \tag{2.5}$$

$$\mathbf{y}^{(t)} = \sigma(\mathbf{W}_y\mathbf{h}^{(t)} + \mathbf{b}_y) \ . \tag{2.6}$$

Where $\mathbf{W}_x$ represents the weight matrix connecting inputs to hidden layers, $\mathbf{W}_h$ represents the weight matrix of recurrent connections, $\mathbf{W}_y$ represents the weight matrix to compute the predictions, and $\mathbf{b}_h$, $\mathbf{b}_y$ represent the bias vectors.

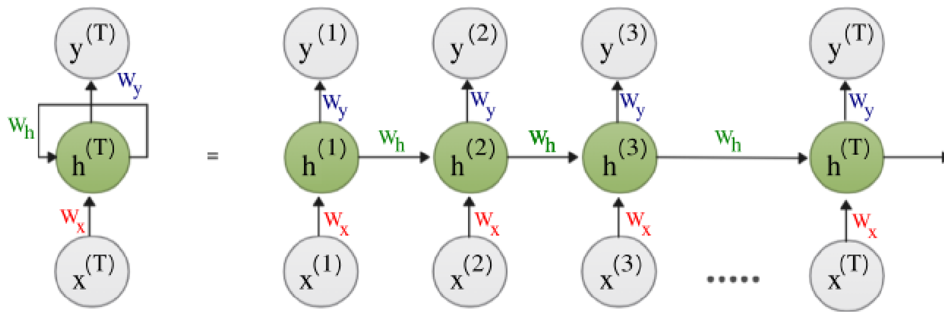


Figure 2.6: An example of the architecture of RNN.

However, there has been problems with these RNNs when considering very long sequences. The vanishing gradient problem has proven to be one of the most significant challenges. The vanishing gradient problem occurs when the gradients decreases exponentially as they backpropagate through time steps, leading to ineffective learning of long-range dependencies. To overcome this problem, other types of networks, such as Long Short-Term Memory (LSTM) networks [52], have been invented. LSTMs have demonstrated their ability to learn long-range connections in sequences effectively [24].

2.1.4 Activation Functions

As we have seen in the previous sections, behind each layer of a type of NN, we generally find an activation function [51, 68, 82]. This function can be seen as the activation signal found in biological neurons. Its main role is to determine whether an artificial neuron is activated or not, depending on the stimulation threshold. Most activation functions are nonlinear, which is crucial for neural networks because this nonlinearity enables them to capture complex patterns and improve prediction accuracy. There are several activation functions, each with its own behavior. In this section, we will define the most commonly used activation functions in NNs.

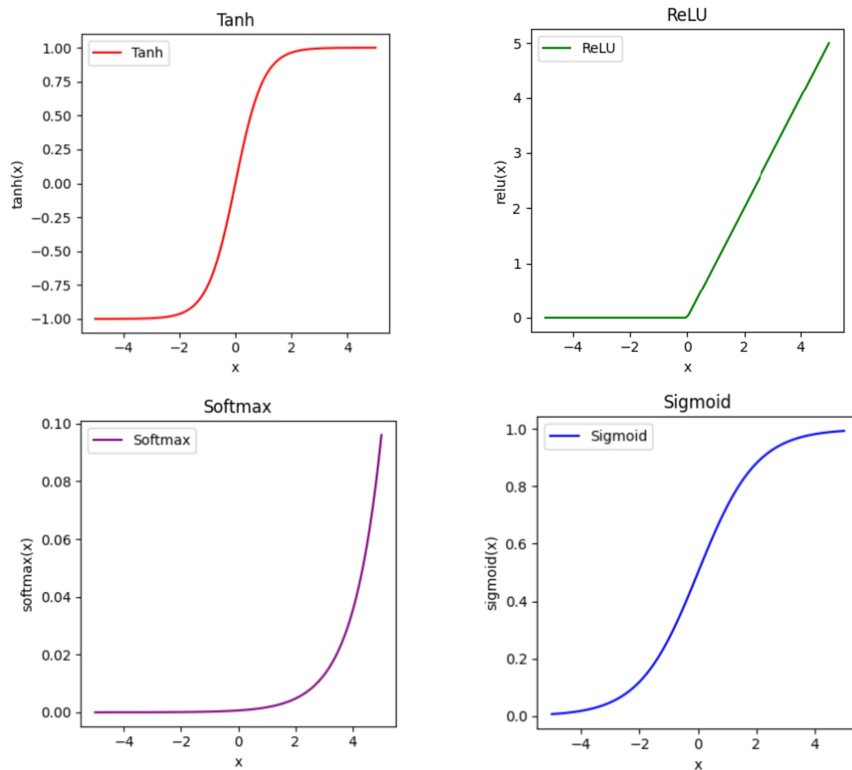


Figure 2.7: Visualizing Common Activation Functions: (Sigmoid, Tanh, ReLU, and Softmax).

The Tanh Function

The hyperbolic tangent function [39, 68], denoted by $tanh$, is defined as follows:

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \quad \forall x \in \mathbb{R}. \tag{2.7}$$

This function normalizes input values to produce outputs in the range of $[-1, 1]$. Unlike the sigmoid function, which gives values between 0 and 1 only, $\tanh$ is particularly beneficial in binary classification models where massive variation should be dealt with properly and data points must be well separated because it captures much of their meaning.

The ReLU Function

The rectified linear unit (ReLU) [39] activation function is the most widely used activation function in deep neural networks (DNNs). Its particularity lies in the fact that it only activates positive neurons, defined as

$$ReLU(x) = max(0, x), \quad \forall x \in \mathbb{R}. \tag{2.8}$$

Compared to the sigmoid function, what makes $ReLU$ better is that it overcomes the vanishing gradient problem by allowing faster convergence, which implices that DNNs can be trained more efficiently.

The Sigmoid Function

The sigmoid function [51], denoted by sig, is defined as follows:

$$sig(x) = \frac{1}{1+e^{-x}}, \quad \forall x \in \mathbb{R}. \tag{2.9}$$

The sigmoid function normalizes input data by producing output values within the range of $[0, 1]$. However, it is important to note that sigmoid functions suffer from limitations. When inputs are either too large or too small, the gradient of the sigmoid function tends to approach zero, leading to a slow convergence rates during the training of deep neural networks. This problem, known as the vanishing gradient.

The Softmax Function

The Softmax function [100] is commonly used in classification problems, such as image categorization. It is defined as:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}, \quad \forall x \in \mathbb{R}. \tag{2.10}$$

Mainly, it is a method of normalizing the outputs of a neural network layer such that the outputs sum to one. This allows the output values to be interpreted as class of probabilities, easing decision-making within the classification context. In other words, the Softmax function changes the output values into probabilities and thus enhances the intuitiveness of predictions made by networks, making them simpler for computer vision, natural language processing, and many other application domains.

2.2 Formal Method and Tools for the Verification of Neural Networks

Image recognition [121] and Natural Language Processing [81] are among the many applications that rely heavily on neural networks. In critical contexts, it is essential to ensure the security and correctness of these networks to avoid potential failures. This is where formal verification methods [113] come in handy. These formal approaches are based on logic and mathematics to determine if a given neural networks meets its requirements or not. We can distinguish two types of formal verification techniques: Complete [83, 85, 63] and incomplete methods [112, 119, 124], depending on the supported neural networks and the underlying verification techniques used. A review of existing methods and tools is provided in Table 2.1.

Before delving into several different verification techniques available, let us clarify what verification stands for in the case of neural networks. The main idea behind verifying neural networks is to prove that they meet specific criteria and behave correctly under various conditions. Among the properties that are often sought to be proven we find:

- Soundness [76]: This refers to a verification approach that gives an explicit answer as to whether a neural networks satisfies a given property or not. For instance, consider the property that a neural networks remains robust against input perturbations, such as variations or noise added to input images during image processing tasks. A sound verification approach rigorously tests the networks is performance under varying levels of disturbance. If the networks consistently maintains accurate classifications despite perturbations, it is considered robust.

- Completeness [76]: A verifier is deemed complete by its ability to detect all violations of a given property, thus aiming to minimize false negatives. False negatives occur when the verifier fails to report an actual violation of a property, leaving a potential failure undetected. For example, in the case of the robustness property of a neural network against perturbations, a complete verifier examines all conditions where the network fails to maintain its accuracy after a perturbation. This ensures that all situations where the network fails to satisfy the robustness property are identified.

2.2.1 Parameters Influencing the Design of Formal Methods

In this section, we will talk about the main characteristics of several formal verification methods for the neural networks. Each method acts differently on the different parts of a neural networks. More precisely:

1. **Architecture of Neural Networks**: As discussed in Section 2.1, there are different types of neural networks (DNN, FCNN, RNN), each with various characteristics such as layer depth, connectivity patterns, etc. that impact the way formal methods are applied. Each method is designed to support a specific type of NN, and the ultimate goal of current research is to develop methods to support a large range of network types.

2. **Activation Function**: Neural networks stability and performance depend on activation functions [82, 68]. Each formal method considers different kinds of functions (see Section 2.1.4). For example, some methods focus only on ReLU functions, whereas others improve their performance by employing $tanh$ or sigmoid functions. For example NNenum only considers ReLU and splits the input domain in order to have the same sign for all the values at the entry of the ReLU.

3. **Norm**: Evaluating the robustness of a neural networks is usually done by using a norm measuring the distance between an input x and a perturbed input y. There exists three common norms:

   - L_1 **Norm**: Measures the sum of the absolute values of the input. Mathematically, for input $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, the L_1 norm is defined as:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i| \ .$$

- L_2 **Norm**: Measures the Euclidean distance between two points. Mathematically, for input $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, the L_2 norm is defined as:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} \ .$$

- L_∞ **Norm**: Measures the largest absolute value among the components of the input, for input $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, the L_∞ norm is defined as:

$$\|\mathbf{x}\|_\infty = \max_{1 \leq i \leq n} |x_i| \ .$$

4. **DataSets**: Used for training and validating neural models play a crucial role in the effectiveness of different formal approaches, influencing the scalability of these approaches. Among the most frequently used datasets for machine learning applications are:

- **MNIST [33] (National Institue of Standard and Technology)** is a database of handwritten digits. It consists of $70,000$ images of size (28×28) classified into 10 classes. The dataset is divided into 2 sets: The training set which contains $60,000$ images and the test set with $10,000$ images.

- **CIFAR-10 [1] (Canadian Institue For Advanced Research)** is used as a reference by many machine learning algorithms. It consists of $60,000$ images of size $(32 \times 32 \times 3)$ classified into 10 classes. The dataset is divided into 2 sets: The training set which contains $50,000$ images and the test set with $10,000$ images.

- **ImageNet** [94] is the standard benchmark for large-scale object recognition. It consists of more than 14 million images classified into $1,000$ classes. The dataset is divided into 2 sets: the training set which contains 1.2 million images and the test set with $50,000$ images.

- **ACAS [60] (The Airborne Collision Avoidance System)** is typically employed to train and test decision-making algorithms for airborne collision avoidance systems by simulating various flight scenarios and assessing the system ability to make safe and effective decisions in these situations. Contrarily to the former datsets, ACAS is not a set of images but a set of abstract scenarios representing aerial collision avoidance situations.

- **ADULT [10]** Also known as the **Census Income** dataset, this dataset was extracted by Barry Becker from the 1994 Census databases. Its main role is to predict whether an individual income exceeds $50,000$ per year based on census data.

- **COMPAS [7] (Correctional Offender Management Profiling for Alternative Sanctions)** is used to analyze the risk scores assigned to offenders to predict the likelihood of recidivism. COMPAS is widely known for its bias against African Americans.

2.2.2 Complete Formal Method

The use of optimal methods to ensure the correctness of neural networks is sometimes referred to as complete formal methods. This is usually done by using SMT (Satisfiability Modulo Theories) [32] solvers, MILP (Mixed Integer Linear Programming) [12] techniques, and global optimization [99]. The main objective of these approaches is to provide a mathematical guarantee of the correctness of neural networks, which is mandatory for critical applications. Nonetheless, these complete formal methods have certain limitations. In general, they are unsuitable for large-scale neural networks due to search space complexity and computational time consumption. Also, they might be restricted to certain architectures of neural networks, which narrows their applicability in some use cases.

Satisfiability Modulo Theories (SMT)

SMT verification methods [17] are based on satisfiabilty of constraints, which involves determining whether a given expression on Boolean variables is true [32]. In the context of neural networks (NN), this approach can be conceived as encoding an NN as a set of constraints and negating the desired safety property. A counter-example is generated (if its exists) as output if these constraints are satisfied. Conversely, if no solution has been found, the safety property is proved to be satisfied. This method has been used these last years by various tools.

One of the first SMT formal verification method for neural networks was introduced by Pauline and Tacchella [85]. In this approach, the authors ensure that the output of a fully connected feed-forward neural network with sigmoid activation functions always falls within certain safety bounds. They achieve this by representing the neural network as a Boolean expression of linear constraints over a product of intervals. Instead of activation functions, they use piecewise linear approximations based on intervals. In other words, the network is encoded with the safety constraints and given to a black-box SMT solver. An improvement consists of refining the encoding by interval spliting to deal with spurious counter-examples. The authors evaluated this technique using small neurons that do not exceed one layer. According to the results obtained, it can be seen that scaling up this method becomes very difficult because refirements increase exponentially, thus leading to larger and larger encoding sizes.

The Planet tool [37], proposed by Ehlers, is helpful in validating feed-forward neural networks with piecewise linear activation functions, It takes advantage of approximations to improve the efficiency of SMT solvers. The tool starts off by finding the lower and upper bounds of neurons using interval arithmetic, then encodes network behavior into linear constraints that approximate ReLU activation functions. Verification problems are encoded as a mix of network approximations and safety properties. The search for a solution entails fixing ReLU activation statuses as well as back tracking when necessary. Although the performance of Planet is promising in verifying neural networks, especially in scenarios like collision avoidance, this approach works only for small networks.

Reluplex [62], introduced by Katz et al., aims to be an efficient SMT solver for verifying

the safety properties of feed-forward fully-connected neural networks with ReLU activation functions. This approach builds upon the simplex algorithm, adapting it to handle ReLU constraints effectively. Reluplex keeps the value assignments continuous over all variables during the process of verification, even though some restrictions are violated at first glance. The assignment continuously changes in such a way that it tries to resolve the violated constraints by splitting frequently occurring ReLU constraints into two cases. This enables breakdown of problems into simpler linear subproblems, thus ensuring the problem converges. If there is no possible assignment satisfying every constraint, then this means that the safety property instance holds true. It has been specifically designed to verify the ACAS Xu neural network [60] for air collision avoidance systems. Experiments have shown satisfactory results, but it takes a lot of time to process very large DNNs.

Marabou [63], as the successor of Reluplex, aims at a unifying platform for verifying and analyzing DNNs with an emphasis on scalability and versatility across multiple applications. Marabou maintains symbolic lower and upper bounds on every neuron, which is represented as a linear combination of input neurons. This enhances Marabou performance in verification tasks since it helps it determine tighter bounds, thereby reducing the input space further. The tests show better performance for Marabou compared to Planet and Reluplex.

The DLV [55] tool was proposed by Huang et al. Its goal is to protect the image classifications against adversarial perturbations which is an intentionally manipulating input data designed to deceive the model into making incorrect predictions or decisions [76]. It allows an exhaustive search of regions of interest through discretization and layer-by-layer analysis. Unlike previous methods, it guarantees finding adversarial examples for specific regions. The exprimental results show that DLV can find the adversarial examples in some seconds for NN trained on the MNIST [33] and ImageNet [94] datasets.

Naroditska et al. [83] suggested an alternative way of checking binarized neural network [56] properties, which is based on boolean satisfiability (SAT) with a counterexample guided search algorithm. The outcomes of their experiments indicated that it was possible to apply this method in practice to mid-sized deep neural networks for image classification problems.

Mixed Integer Linear Programming (MILP)

Mixed Integer Linear Programming (MILP) [12] is a mathematical optimization method that solves problems in which some variables are integers while others are real. In order to solve a MILP problem, one needs to find the values of the variables that either maximize or minimize an objective function subject to a set of linear constraints for both real and integer variables.
When considering the robustness of neural networks, MILP is commonly employed to identify counter-examples or to verify that no counter-examples exists within a given region of the input space. In this case, perturbations applied to the network inputs can be represented by integer variables, whereas linear limitations can indicate the desired behavior of the network and the bounds for these perturbations. Nevertheless, it should be mentioned that solving MILP problems may be computationally expensive, particularly in the case of

deep neural networks, requiring additional optimization techniques to make verification feasible in practice.

The BAB framework developed by Bunel et al. [18] addresses the problems of neural network verification and transforms them into satisfiability problems, which are then solved using the branch and bound method. This technique aims to determine the overall best values for a given neural network by dividing all possible input regions iteratively and calculating upper and lower bounds on their minimum output. BAB integrates existing verification methods such as Planet [37] and Reluplex [62]. Experimental results demonstrate the effectiveness of BAB compared to other verification methods such as Planet and Reluplex. Indeed, BAB achieves an impressive speedup of two orders of magnitude, positioning it as a promising solution for large-scale, high-complexity DNN verification.

Sherlock, proposed by Dutta et al. [35], aims to determine ranges for the outputs of fully connected neural networks with ReLU functions. Sherlock employ an approach based on Mixed Integer Linear Programming (MILP), combined with local search to expedite the process. This method aims to improve the lower and upper bounds of the outputs by iterating between the use of the MILP solver and less costly local searches, based on piecewise linear approximations of the activation functions. The experimental results has been shown that Sherlock outperforms Reluplex.

Tjeng et al. developed MIPVerify [111], which is a tool for finding contradictory instances in special distance metrics (L_1, L_2, L_∞), with a focus on neural network resilience against adversarial examples. This tool specializes in verifying piecewise linear neural networks and uses mixed-integer programming (MIP) to find regions of the input space where desired properties of the network are violated. The results indicate that MIPVerify has allowed determining the exact adversarial accuracy of an MNIST [33] classifier under perturbations bounded by the L_∞ norm for the first time. Moreover, MIPVerify is 2 to 3 orders of magnitude faster than Reluplex [62] when looking for the nearest adversarial example with L_∞.

Global Optimisation

In this approach, to ensure that the model is working properly, global optimization in neural network verification is performed under various conditions. This consists of searching for the parameters of neural networks that minimize a loss function while maximizing their efficiency and resilience over the entire input range. Global optimization creates models that are resilient to unexpected or malicious inputs. They are not only accurate but also trustworthy and applicable in real life scenarios.

DeepGO [91] proposed by Ruan et al., is a solution for verifying continuous Lipschitz neural networks. In essence, it assumes that there is an input set of a neural network, and on its output, a continuous Lipschitz function can be defined whose lower and upper bounds are known. Experimental investigations have revealed that among recent tools like Sherlock [35] and Reluplex [62], DeepGO is the most cost-effective one with better results. Furthermore, it has been observed that DeepGO can run up to 100 times faster than Reluplex, and it has demonstrated its scalability by successfully processing networks

composed of millions of neurons.

The approach of [92] depends on deep neural networks (DNNs) to protect critical systems. The authors propose an iterative approach to compute lower and upper bounds on the networks robustness. They specifically focus on the Hamming distance and investigate the challenge of quantifying the global robustness of trained DNNs. In this context, global robustness refers to the expected maximum safe radius across a testing dataset. This approach provides intermediate results at any time, including upper and lower bounds and robustness estimates. Moreover, it is Tensor-based, conducting computations over sets of inputs simultaneously to leverage efficient GPU processing.

Other Complete Formal Method

ReluVal [117] is a new method for formally verifying the security properties of Deep Neural Networks (DNNs) without employing Satisfiability Modulo Theory (SMT) solvers. ReluVal performs interval arithmetic to establish rigorous range bounds for DNN outputs, resulting in better efficiency and adaptability compared to SMT solvers. This framework operates by propagating input intervals and safety properties through the layers of the DNN, thereby delivering formal guarantees. Symbolic intervals and iterative interval refinement are used by ReluVal in order to mitigate problems of overestimation related to intervals. It significantly enhances the accuracy of analysis. The evaluations show that on average, ReluVal is 200 times faster than Reluplex [62] and also outperforms Marabou [63] in the same setting (when running on 4 cores), thereby making it an important development in formal verification of DNN security.

Neurify [116] is a tool designed to check the safety properties of neural networks efficiently. It can handle safety properties for networks ten times larger than existing approaches. Neurify uses formal analysis to estimate tight and rigorous output ranges of a network, which can guide the training process of robust networks and improve the explainability of decisions made by neural networks. The tool focuses on identifying crucial nodes and refining output approximations iteratively with the help of a linear solver. The outcomes indicate that neurify outperforms current verifiers greatly, being up to 5000 times faster than Reluplex and 20 times faster than ReluVal.

2.2.3 Incomplete Formal Method

To ensure the safety and reliability of large DNN which cannot be verified with the complete methods discribed in section 2.2.2, incomplete formal verification methods are necessary. Indeed, their main advantage is that they can handle large-scale networks more effectively than complete methods. For example, they can run neural networks with more than $1,000$ hidden layers within a couple of minutes [38]. Incomplete methods include abstract interpretation, duality and linear approximation, which have been proven to be sound in general. This gives confidence in their capability to ensure the correctness of NNs. However, sometimes, they can output false alarms, also known as false positives, where a property is mistakenly considered verified even though it is not true; despite this drawback, their

contribution to making NN safer must not be underestimated.

Abstract Interpretation

Abstract interpretation is a theory introduced by Cousot and Cousot [27] in 1977, aimed at modeling the behavior of a system both by simplifying it, and also by approximating sets of values (inputs, intermediary results, etc.) using abstract domains such as zonotopes [44], boxes [80] or polyhedra [29], etc. Thanks to this approach, it is possible to formally analyze a system and verify certain properties without executing it for all possible inputs. In the field of neural network verification, abstract interpretation has been used to study the robustness and reliability of NN.

AI^2 [38] is the first framework able to automatically prove the safety properties of deep neural networks based on abstract interpretation. It was proposed by Gehr et al. AI^2 considers any picewise linear layers (fully connected, convolutional, and max pooling) with ReLU activation functions. The process of evaluating the safety of neural networks with AI^2 has several steps. AI^2 works on NN taking images as inputs. Firstly, it creates a new initial abstract element out of the original input image by adding in a single step a set of perturbations. Doing so, it captures any possible perturbed image in an abstract value (zonotope, polyhedra, interval, etc). Then, this abstract element is propagated through the different layers of the network, requiring the definition of abstract layers to compute the effects of each layer on the abstract elements. These abstract transformers are essential to ensure the accuracy of the analysis. Finally, at the end of the analysis, AI^2 calculates an abstract output, thereby over-approximating all potential outputs produced by perturbed images. This abstract output allows AI^2 directly prove important safety properties such as robustness to perturbations. The experimental results have shown that AI^2 can efficiently handle neural networks that are not handled by existing methods such as Reluplex.

DeepZ [104] is a novel system for certifying the robustness of neural networks based on abstract interpretation. It is different from AI^2 in that it handles various activation functions such as ReLU, $tanh$ and $sigmoid$, and also supports different architectures: feed-forward and convolutional. By using specially designed abstract zonotope transformers, DeepZ ensures increased scalability and accuracy while conforming to the floating-point arithmetic, which is essential for neural network computations. The main outcomes of DeepZ include verification with 97% accuracy under L_∞ attack on a large network with $88,500$ hidden units taking an average time of 133 s as well as local robustness properties check within minutes done for large networks.

DeepPoly [105] is another abstract interpretation tool developed to verify the scalability and precision of neural networks. Its main strength lies in an abstract interpreter specifically designed for neural networks, combining floating-point polyhedra with intervals and abstract transformers for common neural network functions. These transformers attempt to benefit from the main properties of such operations (activation functions, linear transformations, pooling operations, etc.) while, at the same time, keeping the analysis scalable and precise. Concretely, each neuron of a neural network is related to specific bounds based on linear combinations of neurons in previous layers. The results indicate that DeepPoly is generally faster and often more precise than DeepZ. It also succeeds in proving, for the

first time, the robustness of neural networks when the input image undergoes intricate perturbations, like rotations employing linear interpolation.

k-ReLU [103] proposes a new framework to compute precise and scalable convex relaxations for certifying neural networks. Instead of approximating the output of multiple ReLU operations in a layer separately, k-ReLU does it jointly, allowing one to capture dependencies between the inputs of different ReLU operations in a layer. This overcomes the limitations of existing methods by providing a more precise certification while maintaining scalability. The authors apply the framework using the DeepPoly domain, resulting in the creation of a new abstract domain named k-Poly.

The three frameworks, DeepZ [104], DeepPoly [105], and k-Poly [103], developed by Singh et al., have been integrated into a single tool named ERAN.

Urban et al. introduced Libra [112] as a technique that aims to ensure fairness which means ensuring that the predections made by the NN do not exhibit bias or discrimination towards individuals based on sensive attributes e.g., race, sex, age. Libra achieves this in feed-forward neural networks by merging both forward and backward static analysis. The process begins by dividing the input space into independent partitions during the forward pass, thereby reducing the overall analysis effort. Subsequently, through backward analysis, Libra certifies the fairness of classification within each partition, taking into account sensitive features. This approach identifies areas of the input space where the network is fair and those where bias exists, providing adaptive certification. Experiment results on diverse datasets, including Adult [10], COMPAS, German Credit [53], and Japanese Credit [96] demonstrate that Libra performance is predominantly influenced by the scale of the analyzed input space rather than the magnitude of the neural networks. Additionally, this method allows for adjusting the trade-off between scalability and precision by excluding partitions that do not meet the specified configuration, thereby offering flexibility in resource utilization.

Duality

Wong & Kolter [119] have presented a method for training deep ReLU networks in face of norm-bounded perturbations. This approach relies on an external convex approximation of the activations reachable by these perturbations, leveraging a robust optimization method based on linear programming. Its distinguishing feature is the representation of the dual problem of this program in the form of a deep neural network, enabling efficient optimization with guaranteed bounds. Their results are promising, since they have produced a convolutional classifier for MNIST with less than 5.8% test error for any adversarial attack with a L_∞ norm smaller than 0.1. For neural networks having piecewise linear layers and ReLU activations, they employ their approach relying on relative perturbations at various L_p distances. However, while moderately-sized neural networks such as those trained on MNIST and Fashion-MNIST show good performance under their framework, their applicability to more complex architectures like ImageNet classifiers remains questionable.

Dvijotham et al. [36] present a new way of solving optimization problems with Lagrangian relaxation, which bypasses their inherent non-convexity and constraint problems.

Their approach efficiently transforms the dual problem into an unconstrained convex optimization problem, enabling an effective resolution. Additionally, this method provides flexibility by allowing it to be interrupted at any time to obtain an upper bound on the maximum violation of specifications. The authors demonstrate that their approach is effective for developing specialized verification algorithms with accuracy guarantees in different verification contexts. In contrast to other techniques, it presents remarkable advances in the assessment of neural network robustness, particularly for models trained by MNIST.

Brown et al. [16] propose an exact, convex formulation of the verification problem in the form of a completely positive program (CPP) [20]. This approach makes it possible to solve linear optimization problems on the cone of completely positive matrices, thus offering a clear separation between accuracy and feasibility in relaxed verification processes. They also analyze how the formulation remains largely unchanged when the complete positivity constraint is relaxed; hence, it is considered equally useful even in semi-definite programming relaxations (SDP). This helps build reliable SDP checkers and compare them with previous work.

Linear Approximation

FastLin/FastLip [118] are two algorithms developed to compute tight and certified lower bounds of the robustness of fully connected feedforward networks with ReLU activation functions. FastLin utilizes a linear approximation of the ReLU units similar to Neurify [116]. FastLip relies on bounding the local Lipschitz constant of the network. Unlike existing methods based on per-layer operator norms, these algorithms produce more accurate lower bounds, enabling a more reliable evaluation of network robustness. Additionally, experiments demonstrate that FastLin and FastLip are respectively at least four and two orders of magnitude faster than Reluplex [62], allowing for the computation of a reasonable robustness lower bound in less than a minute for ReLU networks with up to 7 layers with more than ten thousand neurons.

Crown [124] is a framework used to certify neural networks that are not necessarily piece-wise linear. It uses upper and lower bounds that are linear or quadratic for typical activation functions. Empirical results demonstrate that CROWN can scale to large networks of over ten thousand neurons and produce a certified lower bound in one minute when executed on one CPU core. Furthermore, it has been demonstrated that Crown can achieve up to 20% improvement in certified lower bounds compared with FastLin [118].

CNN-Cert [14] is a general framework for verifying the robustness of convolutional neural networks with different convolutional, pooling, batch normalization layers etc., and standard activation functions like ReLU, $tanh$, and $arctan$ using FastLin, and CROWN. According to experimental results, CNN-Cert obtains equivalent or even better verification boundaries than FastLin/FastLIP and CROWN, with up to 17 times quicker speeds.

Table 2.1: Neural Network Verification Tools: Essential Overview

	Activation Function	Norm	Dataset	Network	Soundness
Satisfiability modulo therorie: SMT					
Planet	ReLU	L_∞	MNIST	CNN	Yes
ReluPlex	ReLU	L_∞	ACAS	DNN	Yes
Marabou	ReLU	L_∞	ACAS	DNN	Yes
DLV	ReLU	L_∞	MNIST, CIFAR-10, ImageNet	DNN	Yes
Mixed Integer Linear Programming: MILP					
BAB	ReLU	L_∞	ACAS	DNN	Yes
Sherlock	ReLU	L_1, L_∞	MNIST	DNN	Yes
MIPVerify	ReLU	L_1, L_2, L_∞	MNIST, CIFAR-10	DNN	Yes
Global Optimisation and Others methods					
DeepGO	ReLU, $Tanh, Sigmoid$	L_∞	MNIST	Lipschitz	Yes
ReluVal	ReLU	L_∞	MNIST, ACAS	DNN	No
Neurify	ReLU	L_∞	MNIST	DNN	Yes
Abstract Interpretation					
AI^2	ReLU	L_∞	MNIST, CIFAR-10	CNN, DNN	Yes
DeepZ	ReLU, $Tanh, Sigmoid$	L_∞	MNIST, CIFAR-10	DNN, CNN	Yes
DeepPoly	ReLU, $Tanh, Sigmoid$	L_∞	MNIST, CIFAR-10	DNN, CNN	Yes
K-Relu	ReLU	L_∞	ACAS	DNN	Yes
Eran	ReLU	L_∞	MNIST, CIFAR-10	DNN, CNN	Yes
Libra	ReLU	L_∞	ADULT, COMPAS	DNN	Yes
Linear Approximation					
FastLin	ReLU	L_∞	MNIST	CNN	Yes
Crown	ReLU	L_∞	ACAS	DNN	Yes
CNN-Cert	ReLU	L_1, L_2, L_∞	ACAS	DNN	Yes

Chapter 3

Numerical Abstract Domains

3.1 Abstract Domains 32
3.2 Interval Arithemtic 34
3.3 Affine Forms 34
3.3.1 Notations 34
3.3.2 Elementary Operations Among Affine Forms 36
3.3.3 Extensions of Affine Forms 37
3.4 Abstract Transformer of Activation Functions 40
3.5 Neural Network Verification 42
3.6 AFFapy 44
3.6.1 Overview 44
3.6.2 Elementary Arithmetic Functions with Affapy 44
3.7 Conclusion 46

Although being efficient, static analysis by abstract interpretation [88] has limitations, such as the high number of false positives that are linked to the precision of the abstract domains employed. Over time, different abstract domains [29, 79, 49] have been developed to overcome these limitations, each with a more precise representation of specific program behaviors. These abstract domains encode linear or non-linear invariants and have various computational and space complexity suited for different application domains, such as program verification [69] and neural networks verification [104, 105, 38].

In Section 3.1 of this chapter, we shall introduce some of these abstract domains. In Sections 3.2 and 3.3, we will focus on two specific domains: interval arithmetic and affine forms. We will then discuss the extensions of the affine forms domain. Next, in Section 3.4, we will introduce an abstract transformer for activation functions. Finally, we will introduce the library AffaPy in Section 3.6, which we use to develop our own verification tool in the next chapters.

3.1 Abstract Domains

Abstract domains [27, 29, 79] are important basic blocks in static analysis [40] used to obtain an approximation of computer programs behaviors. They permit to define and predict properties of programs, like possible values of variables, without actually running the program. Before we define what an abstract domain is, let us briefly recall some mathematical concepts that we use in our definition.

Definition 2 Posets (Partial Order). A partial order is a set $\mathcal{D}$ equiped with partial order relation $(\mathcal{D}, \sqsubseteq)$ where $\sqsubseteq$ is reflexive ($\forall X \in \mathcal{D}, X \sqsubseteq X$), transitive ($\forall X, Y, Z \in \mathcal{D}, X \sqsubseteq Y \wedge Y \sqsubseteq Z \Rightarrow X \sqsubseteq Z$) and antisymmetric ($\forall X, Y \in \mathcal{D}, X \sqsubseteq Y \wedge Y \sqsubseteq X \Rightarrow X = Y$).

Definition 3 Lower and Upper Bounds. Let $\mathcal{D}'$ be a subsets of poset $(\mathcal{D}, \sqsubseteq)$. An element $u \in \mathcal{D}$ is an upper bound of $\mathcal{D}'$ if $\forall a \in \mathcal{D}', a \sqsubseteq u$. Similarly, An element $l \in \mathcal{D}$ is an lower bound of $\mathcal{D}'$ if $\forall a \in \mathcal{D}', l \sqsubseteq a$. A least upper bound $\sqcup$ of a subset $\mathcal{D}'$, is an upper bound of $\mathcal{D}'$ that is less than or equal to any other upper bound of $\mathcal{D}'$. This is often represented as $\sqcup\mathcal{D}'$. Conversely, a greatest lower bound of a subset $\mathcal{D}'$, is a lower bound $\sqcap$ of $\mathcal{D}'$ that is greater than or equal to any other lower bound of $\mathcal{D}'$. This is often represented as $\sqcap\mathcal{D}'$.

Definition 4 Lattice. A lattice $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap)$ is a poset in which every pair of element $x, y \in \mathcal{D}$ has a least upper bound $(x \sqcup y)$ and great upper bound $(x \sqcap y)$.

Definition 5 Monotonic Function. Consider two posets $(\mathcal{D}_1, \sqsubseteq_1)$ and $(\mathcal{D}_2, \sqsubseteq_2)$. A function $f : \mathcal{D}_1 \to \mathcal{D}_2$ is called monotonic if it satisfies the following condition:

$$\forall x, y \in \mathcal{D}_1, x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y) \quad . \tag{3.1}$$

Definition 6 Galois Connections [27]. let $\mathcal{D}$ and $\mathcal{D}^\sharp$ be two posets. A galois connections is defined by two monotic functions $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ and $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ such that:

$$\forall x, x^\sharp, \alpha(x) \sqsubseteq^\sharp x^\sharp \Leftrightarrow x \sqsubseteq \gamma(x^\sharp) \quad . \tag{3.2}$$

$\mathcal{D}$ is called the concrete domain, $\mathcal{D}^\sharp$ is called the abstract domain, α is called the abstraction function and γ called the concretisation function. Abstract domains can be classified into two main categories:

Abstract non-relational domains: The variables are treated individually without accounting for any relationship between them. The interval abstract domain is a classic example of a non-relational abstract domain.

Definition 7 Interval Abstract Domain

The interval abstract domain, also known as the box abstract domain, is based on the classical concept of interval arithmetic introduced by Moore [80]. It comprises boxes defined by a set of constraints in the form:

$$[a, b] = \{x \in \mathbb{R} : a \leq x \leq b\} \quad . \tag{3.3}$$

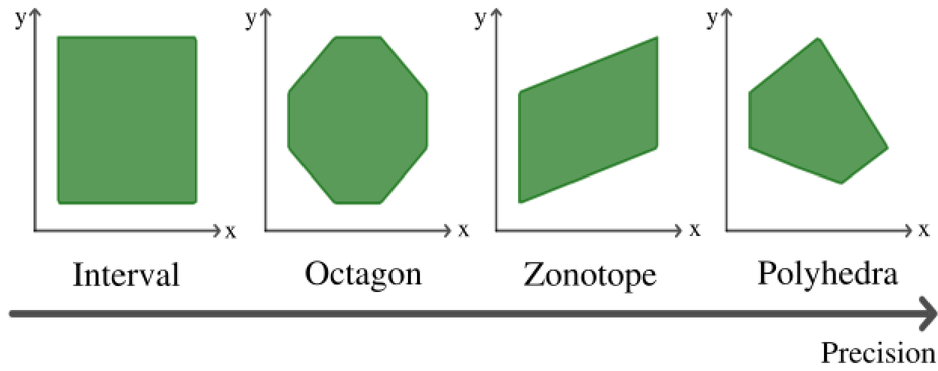


Figure 3.1: Examples of Abstract Domains

Abstract relational domains: The domain takes into consideration the relationships between the variables. Some examples of relational abstract domains include octagons, zonotopes and polyhedra.

Definition 8 Octagon Abstract Domain
The octagon abstract domain, introduced by Miné [79], is defined as a set of linear constraints of the form:

$$\pm x \pm y \leq c \ . \tag{3.4}$$

Where x and y represent variables, and c is a constant.

Definition 9 Polyhedra Abstract Domain
The polyhedra domain introduced by Cousot and Halbwachs [29] consists in a set of linear constraints. These constraints are usually represented as a system of linear inequalities such as:

$$Ax \leq b \ , \tag{3.5}$$

where A is a matrix and b is a vector. It means that each polyhedron in this domain represents a region of n dimensional space, satisfying a set of linear conditions, bounded by planes and half-spaces.

We can note that two domains represent the extremes of accuracy and computational cost. This is shown in Figure 3.1. On one side, there is the interval domain [28] that gives a fast, easy answer but is often not very accurate, while on the other side, there is the polyhedron domain [29] that captures linear relationships between variables, resulting in higher accuracy, yet this comes at the expense of more computationally intensive calculations. All other domains, such as octagons [79] and zonotopes [44], fall between these two extremes in terms of accuracy and computational cost. They offer better precision than the interval domain and are less costly than the polyhedron domain.

In this thesis, all types of abstract domains will not be addressed except for the interval domain and zonotope abstract domain, upon which we shall base our work.

3.2 Interval Arithemtic

Interval arithmetic, introduced by Moore [80] in 1966, is a method for estimating numerical errors in computations by representing a real number x with a pair of floating-point numbers $[\underline{x}, \overline{x}]$ that define the smallest interval enclosing the real number. Mathematically, an interval $[\underline{x}, \overline{x}] \in I$ is defined by

$$[\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\} \ . \tag{3.6}$$

All elementary operations are defined for interval arithmetic. Let $a \in I$ and $b \in I$ be two intervals, and let $+$, $-$, $\times$ denote addition, subtraction and multiplication respectively. The elementary operations between a and b are given by

$$a + b = [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \ , \tag{3.7}$$

$$a - b = [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \ , \tag{3.8}$$

$$a \times b = [min\{\underline{a} \times \underline{b}, \underline{a} \times \overline{b}, \overline{a} \times \underline{b}, \overline{a} \times \overline{b}\}, \tag{3.9}$$

$$max\{\underline{a} \times \underline{b}, \underline{a} \times \overline{b}, \overline{a} \times \underline{b}, \overline{a} \times \overline{b}\}] \ . \tag{3.10}$$

An interval might lack accuracy and be affected by the dependency [77] issue frequently seen in interval computations. This issue arises because each instance of the same variable in a function expression is handled independently. Consequently, the resulting interval can become imprecise. For example, consider the function $f(x) = x - x$ with x in the interval $[-1, 1]$. Applying interval arithmetic to this function yields an enclosure result of $[-2, 2]$, whereas we would expect the result to be $[0, 0]$.
To mitigate these negative effects, several techniques have been introduced, such as affine forms, which we will discuss in detail in the next section.

3.3 Affine Forms

Affine arithmetic [31] finds wide application in different fields, from software verification to neural network verification [45, 42]. Similarly to other relationnal numerical domains, they offer more accurate results than the usual interval arithmetic [49, 47] by capturing linear relationships between variables. Thus, they reduce over-approximations resulting from the loss of correlations in arithmetic intervals. In this section, we provide some background on affine forms and show how calculations can be performed among them.

3.3.1 Notations

The standard affine forms are an extension of interval arithmetic [80]. They have been introduced in 1993 by Comba et Stolfi [67]. their principle consists of preserving linear information between variables during calculations. An affine form is represented by the addition of a constant and a linear combination of the interval $[-1, 1]$. Formally, an affine

form whose symbols is belong a finite set $\mathcal{I}$ is represented as follows:

$$\hat{x} = c + \sum_{i \in \mathcal{I}} x_i \varepsilon_i \quad , \tag{3.11}$$

where c represents a real number, ε_i represents noise symbols whose values are unknown in the interval $[-1, 1]$ and $x_i \in \mathbb{R}, i \in \mathcal{I}$ are called noise weights they correspond intuitively to the intensity of the noise symbolized by ε_i. In the case of neural networks taking images as inputs the center of the affine form c is the value of the pixel and the noise symbols represent the perturbations associated with each pixel.

The relation between an interval and an affine form [25] is obtained by a conversion defined as follows:

- Affine form to interval: Let $\hat{x} = c + \sum_{i \in \mathcal{I}} x_i \varepsilon_i$ be an affine form. The interval associated to $\hat{x}$ is defined by

$$\gamma_I(\hat{x}) = [c, c] + \left(\sum_{i \in \mathcal{I}} |x_i| \right) \times [-1, 1] \quad . \tag{3.12}$$

- Interval to affine form: Let $X = [\underline{x}, \overline{x}]$ be an interval. The affine form associated to X is defined by

$$\alpha_I(X) = \frac{\underline{x} + \overline{x}}{2} + \frac{\underline{x} - \overline{x}}{2} \varepsilon_k \quad . \tag{3.13}$$

  Where ε_k is uknow.

Example 2 Let $\hat{a}$ be an affine form defined by

$$\hat{a} = 6 + \varepsilon_1 + 2\varepsilon_2 - \varepsilon_3 \quad . \tag{3.14}$$

The interval concretization of $\hat{a}$ is given by

$$\gamma_I(\hat{a}) = 6 + (1 + 2 + 1) \times [-1, 1] = [2, 10] \quad . \tag{3.15}$$

Now, we introduce some notations usefull to manage tuples of affine forms, which we refer to as affine sets. We also describe the geometric concretization of sets of values taken by these affine sets.

Definition 10 **(Zonotope [43])** Let $A = (a_1, a_2, ...a_p)$ denote a set of p affine forms, where each affine form is composed of n noise symbols $(\varepsilon_1, \varepsilon_2 ... \varepsilon_n)$. This set of affine forms can be represented by a matrix $C \in M(n + 1, p)$. The concretisation of this set is called the zonotope $\gamma(A)$ and is defined as:

$$\gamma(A) = \{C\epsilon \mid \epsilon \in \{1\} \times \mathcal{B}^n\} \subseteq \mathbb{R}^p \quad . \tag{3.16}$$

where C is given by:

$$C = \begin{pmatrix} \alpha_{01} & \cdots & \alpha_{n1} \\ \vdots & \ddots & \vdots \\ \alpha_{0p} & \cdots & \alpha_{np} \end{pmatrix} .$$

And $\mathcal{B}^n$ is defined as:

$$\mathcal{B}^n = \{\epsilon \in \mathbb{R}^n \mid \|\epsilon\|_\infty \leq 1\} \ . \tag{3.17}$$

Example 3 Let B be the set of 2 affine forms given by

$$B = \begin{pmatrix} \hat{a_1} \\ \hat{a_2} \end{pmatrix} = \begin{pmatrix} 2 + \varepsilon_1 \\ 3 + \varepsilon_1 + \varepsilon_2 \end{pmatrix} .$$

The zonotope $\gamma(B)$ is given in Figure 3.2.

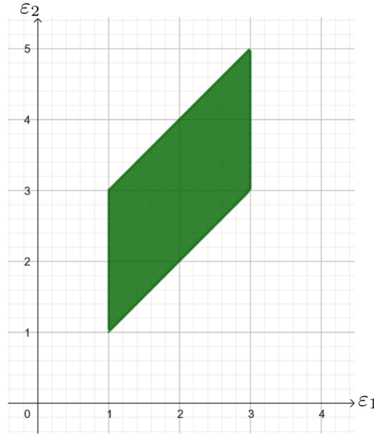


Figure 3.2: The geometrical concretisation $\gamma(B)$.

3.3.2 Elementary Operations Among Affine Forms

In this section, we introduce the elementary operations among affine forms. Let $\hat{a}$ and $\hat{b}$ be two affine forms defined by

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \quad \text{and} \quad \hat{y} = y_0 + \sum_{i=1}^{n} y_i \varepsilon_i \ . \tag{3.18}$$

Let $+$, $-$, $\times$ denote the addition, subtraction, and multiplication by a constant and let $a \in \mathbb{R}$ be a constant. The elementary operations between $\hat{x}$ and $\hat{y}$ are defined by

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i)\varepsilon_i \ , \tag{3.19}$$

$$\hat{x} \pm a = (x_0 \pm a) + \sum_{i=1}^{n} x_i \varepsilon_i \ , \tag{3.20}$$

$$\hat{x} \times a = (a x_0) + \sum_{i=1}^{n} (a x_i)\varepsilon_i \tag{3.21}$$

The multiplication of two affine forms is a non-affine operation, but its result must be approximated using new affine form. Consequently, to represent the error arising from this

non linear operation, an additional noise symbol must be incorporated. Thus, the multiplication of two affine forms is defined by

$$\begin{aligned}\hat{x} \times \hat{y} &= (x_0 + \sum_{i=1}^{n} x_i\varepsilon_i) \times (y_0 + \sum_{i=1}^{n} y_i\varepsilon_i) \\ &= x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\varepsilon_i + \left(\sum_{i=1}^{n}|x_i| \times \sum_{i=1}^{n}|y_i|\right)\varepsilon_{n+1} \quad .\end{aligned} \tag{3.22}$$

where, ε_{n+1} is a new noise symbol used to represent the over-approximate due to the non-linear operation. We can note that this approximation appears to be the most effective. It strikes a balance between the complexity of the multiplication and the accuracy of the bounds.

3.3.3 Extensions of Affine Forms

Predicting the size of affine forms is challenging due to the way non-linear operations are carried out, which necessitates introducing a new variable for each non-linear operation. To address this issue, Messine [78, 77] provides two extensions (referred to as AF1 and AF2). These extended affine forms maintain the uncertainty that arises from approximating non-affine terms but limits the number of variables involved in an expression.

- **AF1 Forms [78, 77]**, are based on the same ideas as standard affine forms, with the exception that all terms resulting from approximation errors are now combined into a single term. As a result, The number of noise symbols does not change during a computation. The format of an AF1 form is

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i + x_{n+1}\varepsilon_{n+1} \quad . \tag{3.23}$$

  Where ε_{n+1} represents a new noise symbol that accounts for all the errors. The elementary operations among AF1 forms are defined as follows:

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{n}(x_i \pm y_i)\varepsilon_i + (x_{n+1} \pm y_{n+1})\varepsilon_{n+1} \quad , \tag{3.24}$$

$$\hat{x} \pm a = (x_0 \pm a) + \sum_{i=1}^{n}(x_i)\varepsilon_i + |x_{n+1}|\varepsilon_{n+1} \quad , \tag{3.25}$$

$$\hat{x} \times a = (ax_0) + \sum_{i=1}^{n}(ax_i)\varepsilon_i + |a|x_{n+1}\varepsilon_{n+1} \quad , \tag{3.26}$$

$$\begin{aligned}\hat{x} \times \hat{y} = x_0y_0 + \sum_{i=1}^{n}(x_0y_i + y_0x_i)\varepsilon_i + (|x_0|y_{n+1} + \\ |y_0|x_{n+1}) + \sum_{i=1}^{n+1}|x_i| \times \sum_{i=1}^{n+1}|y_i|\varepsilon_{n+1} \quad .\end{aligned} \tag{3.27}$$

- **AF2 Forms [78, 77]**, are based on AF1. Although there is always the same number of variables, the error is recorded using three distinct terms: the indeterminate, negative, and positive noise symbols. An AF2 forms $\hat{x}$ is defined by

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i + x_{n+1}\varepsilon_{n+1} + x_{n+2}\varepsilon_{n+2} + x_{n+3}\varepsilon_{n+3} \quad . \tag{3.28}$$

where x_{n+1}, x_{n+2} and x_{n+3} are non-negative numbers.

The elementray operations among AF2 forms $\hat{x}$ and $\hat{y}$ are as follows

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{n}(x_i \pm y_i)\varepsilon_i + \sum_{i=1}^{n}(x_{n+1} + y_{n+1})\varepsilon_{n+1} + \sum_{i=1}^{n}(x_{n+2} + y_{n+2})\varepsilon_{n+2} + \sum_{i=1}^{n}(x_{n+3} + y_{n+3})\varepsilon_{n+3} \quad , \tag{3.29}$$

$$\hat{x} \pm a = (x_0 \pm a) + \sum_{i=1}^{n} x_i\varepsilon_i + x_{n+1}\varepsilon_{n+1} + x_{n+2}\varepsilon_{n+2} + x_{n+3}\varepsilon_{n+3} \quad , \tag{3.30}$$

$$\hat{x} \times a = (ax_0) + \sum_{i=1}^{n} ax_i\varepsilon_i + |a|x_{n+1}\varepsilon_{n+1} + |a|x_{n+2}\varepsilon_{n+2} + |a|x_{n+3}\varepsilon_{n+3} \quad . \tag{3.31}$$

For the multiplication of AF2 forms, the complexity increases as for standard affine forms. We have:

$$\hat{x} \times \hat{y} = x_0y_0 + \sum_{i=1}^{n}(x_0y_i + x_iy_0)\varepsilon_i + K_1\varepsilon_{n+1} + K_2\varepsilon_{n+2} + k_3\varepsilon_{n+3} \quad , \tag{3.32}$$

where K_1, K_2 and K_3 are three positive constants given by

$$K_1 = |x_0|y_{n+1} + |y_0|x_{n+1} + \sum_{i=1}^{n+3}\sum_{j=1, j\neq i}^{n+3} |x_iy_i| \quad , \tag{3.33}$$

$$K_2 = A + \sum_{i=1, x_iy_i>0}^{n+3} x_iy_i \quad , \tag{3.34}$$

$$K_3 = B + \sum_{i=1, x_iy_i<0}^{n+3} |x_iy_i| \quad , \tag{3.35}$$

and where A and B are given by

$$A = \begin{cases} x_0y_{n+2} + y_0x_{n+2} & \text{si } x_0 < 0 \text{ and } y_0 > 0 \\ x_0y_{n+2} - y_0x_{n+3} & \text{si } x_0 > 0 \text{ and } y_0 < 0 \\ -x_0y_{n+3} + y_0x_{n+2} & \text{si } x_0 < 0 \text{ and } y_0 > 0 \\ -x_0y_{n+3} - y_0x_{n+3} & \text{si } x_0 < 0 \text{ and } y_0 < 0 \end{cases} \quad ,$$

$$B = \begin{cases} x_0 y_{n+3} + y_0 x_{n+3} & \text{si } x_0 > 0 \text{ and } y_0 > 0 \\ x_0 y_{n+3} - y_0 x_{n+2} & \text{si } x_0 > 0 \text{ and } y_0 < 0 \\ -x_0 y_{n+2} + y_0 x_{n+3} & \text{si } x_0 < 0 \text{ and } y_0 > 0 \\ -x_0 y_{n+2} - y_0 x_{n+2} & \text{si } x_0 < 0 \text{ and } y_0 < 0 \ . \end{cases}$$

Note that there exists other extensions, such as the quadratic form (QF) [77], which record information about the quadratic terms of certain quadratic or polynomial calculations.This extension is defined as:

$$\hat{x} = x_0 + \sum_{i=1}^{n} (x_i \varepsilon_i + x_{i+n} \varepsilon_i^2) + x_{2n+1} \varepsilon_{2n+1} + x_{2n+2} \varepsilon_{2n+2} + x_{2n+3} \varepsilon_{2n+3} \ . \tag{3.36}$$

Where $\varepsilon_i \in [0, 1]$ represents the new square symbolic variable and $\varepsilon_{2n+1}, \varepsilon_{2n+2}, \varepsilon_{2n+3}$ are non-negative numbers.

Elementary operations ($+$, $-$, multiplication by a constant), follow the same principle as the $AF1$ and $AF2$ forms. It involves the addition or subtraction of the same terms together. The elementary operations of two quadratic forms $\hat{x}$ and $\hat{y}$ are defined by

$$\begin{aligned} \hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{2n} (x_i \pm y_i) \varepsilon_i + (x_{2n+1} + y_{2n+1}) \varepsilon_{2n+1} + \\ (x_{2n+2} + y_{2n+2}) \varepsilon_{2n+2} + (x_{2n+3} + y_{2n+3}) \varepsilon_{2n+3}, \end{aligned} \tag{3.37}$$

$$a \pm \hat{x} = (a \pm x_0) + \sum_{i=1}^{2n} x_i \varepsilon_i + x_{2n+1} \varepsilon_{2n+1} + x_{2n+2} \varepsilon_{2n+2} + x_{2n+3} \varepsilon_{2n+3}, \tag{3.38}$$

$$a \times \hat{x} = \begin{cases} a x_0 + \sum_{i=1}^{2n} a x_i \varepsilon_i + a x_{2n+1} \varepsilon_{2n+1} + a x_{2n+2} \varepsilon_{2n+2} + a x_{2n+3} \varepsilon_{2n+3}, & \text{if } a > 0, \\ a x_0 + \sum_{i=1}^{2n} a x_i \varepsilon_i + |a| x_{2n+1} \varepsilon_{2n+1} + |a| x_{2n+3} \varepsilon_{2n+2} + |a| x_{2n+2} \varepsilon_{2n+3}, & \text{else.} \end{cases} \tag{3.39}$$

However, complexity arises, particularly in the multiplication of two quadratic forms. The multiplication of two quadratic (QF) forms $\hat{x}$ and $\hat{y}$ is defined by

$$\begin{aligned} \hat{x} \times \hat{y} = x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + x_i y_0) \varepsilon_i + \sum_{i=1}^{n} (x_0 y_{i+n} + x_{i+n} y_0 + x_i y_i) \varepsilon_{i+n} \\ + K_1 \varepsilon_{2n+1} + K_2 \varepsilon_{2n+2} + K_3 \varepsilon_{2n+3} \ , \end{aligned} \tag{3.40}$$

where $K_1, K_2$and K_3 are 3 non-negative numbers defined by

$$\begin{aligned} K_1 = A + x_{2n+1} y_{2n+1} + \sum_{i=1}^{n} (|x_i| (y_{2n+2} + y_{2n+3}) + |y_i| (x_{2n+1} + x_{2n+2} + x_{2n+3}) \\ + |x_{i+n}| y_{2n+1} + |y_{i+n}| x_{2n+1}) + \sum_{i=1}^{n} \sum_{j=1}^{n} (|x_i y_{j+n}| + |y_i x_{j+n}|) + \sum_{i=1}^{n} \sum_{j=1, i \neq j} x_i y_i \ , \end{aligned} \tag{3.41}$$

$$K_2 = B + x_{2n+2}y_{2n+2} + x_{2n+3}y_{2n+3} + \sum_{i=1, x_i y_{2n+2}>0}^{n} x_i y_{2n+2} + \sum_{i=1, y_i x_{2n+2}>0}^{n} y_i x_{2n+2} + \sum_{i=1, x_i y_{2n+3}>0}^{n} x_i y_{2n+3} + \sum_{i=1, y_i x_{2n+3}>0}^{n} y_i x_{2n+3}^{n} + \sum_{i=1}^{n}{}^{n} \sum_{x_{i+n} y_{j+n}>0}^{n} x_{i+n} y_{j+n} \quad , \tag{3.42}$$

$$K_3 = C + x_{2n+2}y_{2n+3} + x_{2n+3}y_{2n+2} + \sum_{i=1, x_i y_{2n+2}<0}^{n} |x_i y_{2n+2}| + \sum_{i=1, y_i x_{2n+2}<0}^{n} |y_i x_{2n+2}| + \sum_{i=1, x_i y_{2n+3}<0}^{n} |x_i y_{2n+3}| + \sum_{i=1, y_i x_{2n+3}<0}^{n} |y_i x_{2n+3}| + \sum_{i=1, x_{i+n} y_{j+n}<0}^{n} |x_{i+n} y_{j+n}| \quad . \tag{3.43}$$

In equations (3.41) to (3.43) A, B and C are given by

$$A = |x_0| y_{2n+1} + |y_0| x_{2n+1} \quad , \tag{3.44}$$

$$B = \begin{cases} x_0 y_{2n+2} + y_0 x_{2n+2} & \text{if } x_0 > 0 \text{ and } y_0 > 0 \\ x_0 y_{2n+2} - y_0 x_{2n+3} & \text{if } x_0 > 0 \text{ and } y_0 < 0 \\ -x_0 y_{2n+3} + y_0 x_{2n+2} & \text{if } x_0 < 0 \text{ and } y_0 > 0 \\ -x_0 y_{2n+3} - y_0 x_{2n+3} & \text{if } x_0 < 0 \text{ and } y_0 < 0 \quad , \end{cases}$$

$$C = \begin{cases} x_0 y_{2n+3} + y_0 x_{2n+3} & \text{if } x_0 > 0 \text{ and } y_0 > 0 \\ x_0 y_{2n+3} - y_0 x_{2n+2} & \text{if } x_0 > 0 \text{ and } y_0 < 0 \\ -x_0 y_{2n+2} + y_0 x_{2n+3} & \text{if } x_0 < 0 \text{ and } y_0 > 0 \\ -x_0 y_{2n+2} - y_0 x_{2n+2} & \text{if } x_0 < 0 \text{ and } y_0 < 0 \quad . \end{cases}$$

3.4 Abstract Transformer of Activation Functions

Activation functions [82, 51] are essential in the operation of neural networks, as they introduce the non-linearity needed to learn complex tasks. However, a major challenge arises when it comes to verifying neural networks using abstract domains. These abstract domains are not exact for this non-linear functions such ReLU and $Sigmoid$. This creates a difficulty in the verification process, as it is necessary to find a way of approximating these functions while retaining the maximum amount of information.

Over time, several approaches have been developed to approximate these functions. The aim is to find an abstract transformer α_f that acts as an operator between the initial abstract domain $\mathcal{D}^\sharp$ and the new abstract domain $\mathcal{D}^{'}$. In this section, we discuss some state-of-the-art techniques used for constructing an abstract transformer for one type of this function, the ReLU function, since it will be used in the rest of our work.

Abstract Transformer with Interval Domain

As we know, most activation functions are monotonically increasing. Therefore, we can design an abstract transformer for any such function [4] as follows:

$$f^\sharp([l,u]) = [f^\sharp(l), f^\sharp(u)]. \tag{3.45}$$

In the case of ReLU, for example, we apply ReLU to the upper and lower bounds. We have

$$ReLU([l,u]) = [ReLU(l), ReLU(u)]. \tag{3.46}$$

The abstraction takes the form of a box, as seen in the Figure 3.3.

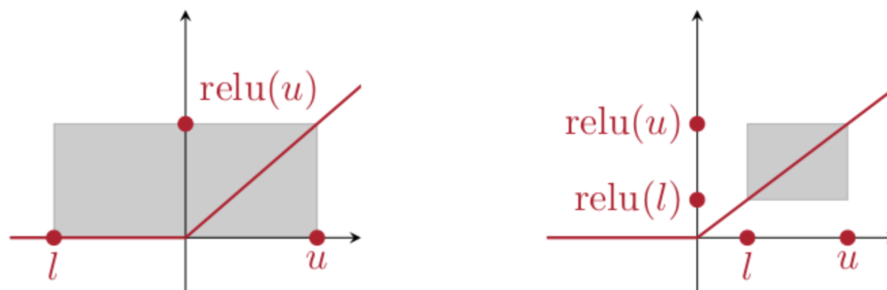


Figure 3.3: Interval approximations for the ReLU function (photo credits from [4]).

The problem with this approximation is that it is an over-approximation. The primary reason for this is that interval domains do not keep track of the relationships between variables. Consequently, we do not know how the input points are related to the output.

Abstract Transformer with Polyhedral domain

This abstraction is discussed in [105]. The main idea involves associating each variable x_i with the polyhedral constraints l_i and u_i, corresponding to the lower and upper bounds for each variable, respectively. The abstraction is given by three cases:

1. $l_i > 0$, all the variables are positive, and the output is identical to the input; thus, the result is exact, and no approximation is needed,

2. $u_i \leq 0$, all the variables are negative, and the output is zero; again, the result is exact, and no approximation is needed,

3. $l_i > 0$ and $u_i \leq 0$ the output can be either positive or negative. Therefore, it is necessary to approximate the ReLU while retaining the maximum amount of information. The authors propose three approaches to approximate this case (see Figure 3.4). The approximation method shown in Figure 3.4(a) aims to minimize the area in the x_i, x_j

plane and sets the following constraints and bounds for x_j:

$$x_i \leq x_j, 0 \leq x_j,$$
$$x_j \leq \frac{u_i(x_i - l_i)}{u_i - l_i},$$
$$l_j = 0, u_j = u_i.$$

However, this approach includes two lower polyhedral constraints for x_j, which can lead to an excessive number of constraints during analysis. To prevent this, the authors simplify the approximation by allowing only one lower bound and consider two alternative methods, as shown in Figures 3.4(b) and 3.4(c), and select the one with the smallest area.

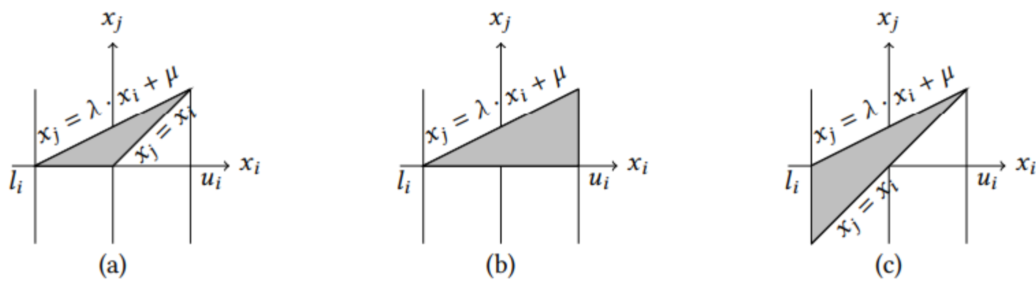


Figure 3.4: Convex approximations for the ReLU function. In the figure, $\lambda = \frac{u_i}{u_i - l_i}$ and $\mu = \frac{-l_i u_i}{u_i - l_i}$ (photo credits from [105]).

There are other methods that use other types of abstract domains, such as Zonotopes. We will examine these techniques in detail in Chapter 5.

3.5 Neural Network Verification

As discussed in Chapters 1 and 2, the verification of neural networks has become essential to ensure their proper functioning, especially when deployed in critical systems. The verification of a neural network involves determining whether a model satisfies certain properties (such as robustness and safety), generally defined by a set of constraints on its inputs and outputs.

Several methods have been developed to ensure this verification (see Section 2.2). However, in this section, we will focus on two of these methods: interval arithmetic and affine forms. Both methods determine how perturbations on the inputs propagate through the network (see Figure 3.5 and 3.6).

Interval arithmetic [80] offers a structured approach to manage uncertainties in computations by representing values as ranges defined by lower and upper limits. In the context of verifying neural networks, this technique provides an effective means of establishing bounds on the outputs when a range of input values is considered. The primary objective of verification using interval arithmetic [105] is to derive guaranteed output ranges for a

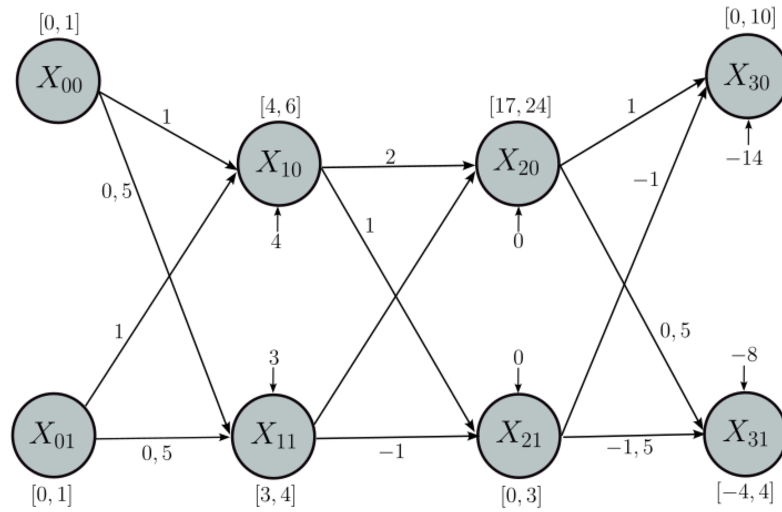


Figure 3.5: Examples of Propagation of Intervals in a Neural Network Across Multiple Layers

neural network, based on intervals that reflect the possible variations in the inputs. This approach enhances the robustness of the model by taking into account the uncertainties present in the input data [90].

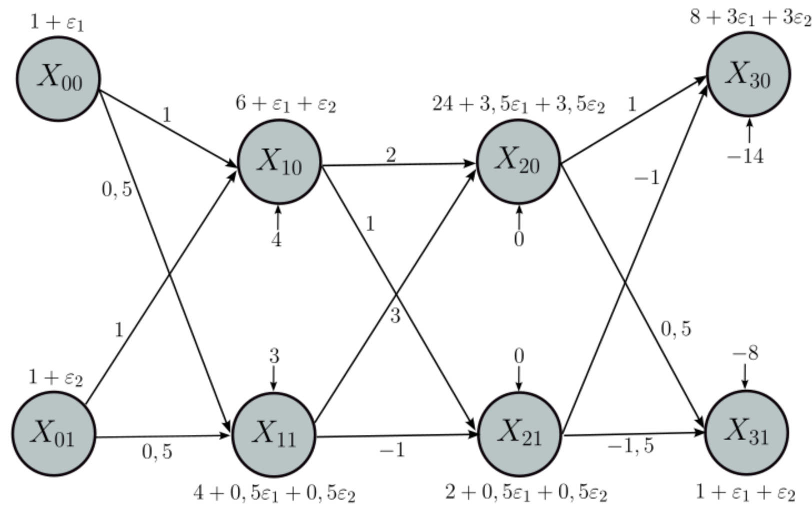


Figure 3.6: Examples of Propagation of affine forms in a Neural Network Across Multiple Layers

On the other hand, Affine forms [31] represent an effective method for managing uncertainties in computations by modeling values as a linear combination of the value itself and noise symbols that reflect all possible perturbations. This technique allows for the preservation of information throughout the calculation process, as it takes into account the relationships between variables. This makes affine forms particularly well-suited for complex verification tasks [104].

The main difference between these two methods lies in their precision and their ability to capture relationships between variables.

3.6 AFFapy

In this section, we define the Affapy library, a crucial tool for our work in representing affine forms. Affapy is a library for affine arithmetic that provides a Python implementation of standard operations on affine forms. This library is essential for accurately modeling uncertainties and performing computations with affine representations, which are central to our analysis. The Affapy enhances the reliability of our models by ensuring that the affine forms we utilize are correctly represented and manipulated throughout our computations. We start this section with an overview of the library and then present the main features that we will use throughout our work.

3.6.1 Overview

Affapy [50] is a Python library developed by Thibault Hilaire that is able to perform any arithmetic operation on intervals as well as affine arithmetic. It consists of 4 modules:

- Affine (aa): This module creates affine forms and performs various arithmetic operations among them,
- Interval (ia): This module creates interval forms and performs various arithmetic operations among them,
- Error: manages errors (warnings, exceptions),
- Precision: Manages the precision of computaions. Users can select the precision with which they wish to work.

3.6.2 Elementary Arithmetic Functions with Affapy

Here, we go into the details of the features of the Affine module, which will be especially useful for our work. We examine the available arithmetic operations as well as the ability to manipulate affine forms. These features will be widely employed in the verification methodology we are going to develop later.

The function *Affine* generates an affine form. It takes two parameters:

- *x0* : The center of the form,
- *xi*: The noise symbols.

Below is an example of affine form definition using Affapy

```
From affapy.aa import Affine
a = affine(x0 = 2, xi={1:0.02, 2:0.04})
>>2.0 + 0.02e1 + 0.04e2
```

To retrieve any of the parameters (center, noise symbols, radius or interval), we can use the following properties

- *x0()*: Returns the center,
- *xi()*: Returns the noise symbols,
- *rad()*: Returns the radius,
- *interval()*: Returns the interval (converting affine forms into an interval).

Below is an example to show how to extract the center and noise symbols of an affine form, as well as how to calculate the radius of the affine form using Affapy

```
a.x0
>> 2.0
a.xi
>> {0.02, 0.04}
a.rad
>> 0.06
```

The operator $+$ can be used for the addition of two affine forms or for adding an integer or float to an affine form. Below is an example to show how to calculate the addition of two affine forms using affapy.

```
From affapy.aa import Affine
b = affine(x0 = 2, xi={1:0.02, 2:0.04})+ affine(x0 = 4, xi={1:-0.05,
    2:0.03})
>> 6.0 - 0.03e1 + 0.07e2
```

The operator $-$ can be used for substraction of two affine forms or for substracting an integer or float from an affine form. Below is an example to show how to calculate the subtraction of an affine form and a constant using affapy

```
From affapy.aa import Affine
a = affine(x0 = 2, xi={1:0.02, 2:0.04}) - 4
>> -2.0 + 0.02e1 + 0.04e2
```

The operator $\times$ can be used for multiplication of two affine forms or for multiplying an integer or float by an affine form. Below is an example to show how to calculate the multiplication of two affine forms using affapy.

```
From affapy.aa import Affine
a = affine(x0 = 3, xi={1:0.02, 2:0.04}) * affine(x0 = 1, xi
   ={1:0.04})
>> 3.0 + 0.14e1 + 0.04e2+ 0.0024e3
```

Affapy has several other functions that can be briefly mentioned:

- abs(): return the absolute value of an affine form,
- *sqrt()*: return the square root of an affine form,
- exp(): return the exponential of an affine form,
- log(): return the logarithm of an affine form,
- tanh(): return the tangent value of an affine form,
- sin(): return the sinus value of an affine form,
- cos(): return the cosinus value of an affine form.

3.7 Conclusion

In this chapter, we have presented essential concepts related to the abstract domains used in our study, placing particular emphasis on a key domain: affine forms. We discussed the various extensions of this domain, the essential operations associated with it, and its implementation. This allows us to deepen our understanding of their applications in the context of our work.

Additionally, we have introduced an important element: the existing methods for representing activation functions, which constitute a crucial aspect of our approach. By combining these elements, we lay the foundation necessary for the development and verification of neural networks in the following chapters.

Part II

Contribution

Chapter 4

Efficient Neural Network Validation with Affine Forms

4.1	Compressed Affine Forms	50
4.1.1	Principe	50
4.1.2	Abstract Domain of Compressed Affine Forms	52
4.2	Validating Neural Networks with Affine Forms	54
4.2.1	Experimental Setting	54
4.2.2	Influence of Merging Symbols on Accuracy	56
4.2.3	Influence of Merging Symbols on Time	57
4.2.4	Case Study of Convolutional Neural Networks	58
4.3	Local Noise	60
4.3.1	Accuracy in Function of Local Noise	60
4.3.2	Execution Time in Function of Local Noise	61
4.4	Conclusion	62

Affine forms [31, 67] are widely used in software verification [45, 42]. Like other relational domains [79, 29], they make it possible to obtain more precise results than the usual interval arithmetic [49, 47] by recording linear relations between variables. Doing so, they fight against the over-approximations introduced by the loss of correlations in the interval arithmetic. More recently, affine forms have been successfully used for the validation of neural networks [104]. Indeed, neural networks are used in more and more domains including critical systems and, in this context, ensuring the safety of the system, and consequently the safety of the neural network, becomes mandatory. Affine forms are well-suited to the verification of neural networks since they capture affine relations between variables and since neural networks perform mostly affine computations. However, an important drawback of affine forms, used in the context of neural network validation, is that they are time and memory consuming while in the same time neural networks become larger and larger. Then it becomes difficult (or impossible) to run industrial-size neural networks, with affine

forms, for memory and execution-time limitations. For example, Lenet can be verified in a few minutes, whereas for more complex networks like VGG16, which has 138 million parameters, verification becomes practically impossible due to excessive time and memory consumption.

In this chapter, we propose our first contribution in the field of neural network verification using abstract interpretation. We suggest a new technique called “Compressed Affine Forms”. Compressed affine forms are an abstraction (in the sense of abstract interpretation) of affine forms. They use less noise symbols and, consequently, they are less time and memory consuming for an acceptable precision loss.

Section 4.1 defines our specific approach. Next, Section 4.2 presents some experiments to assess the effectiveness of compressed affine forms in practice. Section 4.2.4, investigates our approach in the case of convolutional networks. As another way of reducing the number of noise symbols in affine forms, local noise is finally presented in Section 4.3.

Excepted Section 4.2.4, the work presented below is excerpted from the article "Efficient neural networks with affine forms", which has been published in ICSRS (The 8th International Conference on System Reliability and Safety, 2022)[106].

4.1 Compressed Affine Forms

In this Section, we introduce compressed affine forms. Intuitively, we merge several noise symbols into a single noise symbol in a way which ensures the soundness of the computations. This means that the interval concretization of the usual affine form is included in the interval concretization of the compressed affine form. From the theoretical point of view, we show that compressed affine forms are an abstraction, in the sense of abstract interpretation, of affine forms (see Section 3.1). Compressed affine forms being an abstraction of affine forms, they are less precise. Meanwhile, since they use less noise symbols, computing with them is significantly faster (and less memory consuming) than computing with usual affine forms. In other words, compressed affine forms represent a trade-off between, in one hand precision and, in the other hand, execution time and memory consumption.

4.1.1 Principe

In this section, we introduce informally the compressed affine forms and we show how to compute with them. These ideas are presented in [93]. We denote $\mathcal{F}(\mathcal{I})$ the set of affine forms whose indexes belong to the finite set $\mathcal{I}$. In the following we take for $\mathcal{I}$ a bounded set of integers $[1, N]$ for sufficiently large value of N. Let $\hat{x} \in \mathcal{F}(\mathcal{I})$ and $\hat{y} \in \mathcal{F}(\mathcal{I})$ be two affine forms such that

$$\hat{x} = 4 + 0.1\varepsilon_1 + 0.3\varepsilon_2 + 0.2\varepsilon_3 \quad \text{and} \quad \hat{y} = 1 + 0.2\varepsilon_1 + 0.2\varepsilon_2 + 0.3\varepsilon_4 \ . \tag{4.1}$$

For efficiency reasons, instead of $\hat{x}$ and $\hat{y}$, we want to use two compressed affine forms $x^\sharp$ and $y^\sharp$ inside which ε_1 and ε_2 are merged into a single noise symbol. We aim at doing

Expr.	Value	Concretitization.
$\hat{x}$	$4 + 0.1\varepsilon_1 + 0.3\varepsilon_2 + 0.2\varepsilon_3$	$[3.4, 4.6]$
$\hat{y}$	$1 + 0.2\varepsilon_1 + 0.2\varepsilon_2 + 0.3\varepsilon_4$	$[0.3, 1.7]$
$x^\sharp$	$4 + 0.2\varepsilon_3 + 0.4\varepsilon_5$	$[3.4, 4.6]$
$y^\sharp$	$1 + 0.3\varepsilon_4 + 0.4\varepsilon_6$	$[0.3, 1.7]$
$\hat{x} + \hat{y}$	$5 + 0.3\varepsilon_1 + 0.5\varepsilon_2 + 0.2\varepsilon_3 + 0.3\varepsilon_4$	$[3.7, 6.3]$
$\hat{x} - \hat{y}$	$3 - 0.1\varepsilon_1 + 0.1\varepsilon_2 + 0.2\varepsilon_3 - 0.3\varepsilon_4$	$[2.3, 3.7]$
$\hat{x} \times \hat{y}$	$4 + 0.9\varepsilon_1 + 1.1\varepsilon_2 + 0.2\varepsilon_3 + 1.2\varepsilon_4 + 0.42\varepsilon_h$	$[0.18, 7.82]$
$x^\sharp + y^\sharp$	$5 + 0.2\varepsilon_3 + 0.3\varepsilon_4 + 0.4\varepsilon_5 + 0.4\varepsilon_6$	$[3.7, 6.3]$
$x^\sharp - y^\sharp$	$3 + 0.2\varepsilon_3 - 0.3\varepsilon_4 + 0.4\varepsilon_5 - 0.4\varepsilon_6$	$[1.7, 4.3]$
$x^\sharp \times y^\sharp$	$4 + 0.2\varepsilon_3 + 1.2\varepsilon_4 + 0.4\varepsilon_5 + 1.6\varepsilon_6 + 0.42\varepsilon_h$	$[0.18, 7.82]$

Table 4.1: Elementary operations between affine forms and compressed affine forms and their concretizations.

that soundly, i.e. we want that what we compute with the compressed affine forms over-approximates what we compute with the original affine forms. In other terms, let $\gamma_I : \mathcal{F}(\mathcal{I}) \to I$ (I being the set of intervals introduced in Section 3.2) be the concretization function introduced in Equation (3.13). For $* \in \{+, -, \times, \div, \ldots\}$, we want

$$\gamma_I(\hat{x} * \hat{y}) \subseteq \gamma_I(x^\sharp * y^\sharp) \quad , \tag{4.2}$$

The interval concretizations of $\hat{x}$ and $\hat{y}$ are given in Table 4.1. Recall that, the concretization function γ_I is formally defined in Section 3.2. We aim at merging ε_1 and ε_2 into a new symbol named, e.g., ε_{12}. However we cannot use the same symbol ε_{12} in both $\hat{x}$ and $\hat{y}$. If we would do so, we would have the undesired following result:

$$\begin{aligned} x^\sharp = 4 + 0.1\varepsilon_{12} + 0.3\varepsilon_{12} + 0.2\varepsilon_3 &= 4 + (|0.1| + |0.3|)\varepsilon_{12} + 0.2\varepsilon_3 \\ &= 4 + 0.4\varepsilon_{12} + 0.2\varepsilon_3 \end{aligned} \tag{4.3}$$

and

$$\begin{aligned} y^\sharp = 1 + 0.2\varepsilon_{12} + 0.2\varepsilon_{12} + 0.3\varepsilon_4 &= 1 + (|0.2| + |0.2|)\varepsilon_{12} + 0.3\varepsilon_4 \\ &= 1 + 0.4\varepsilon_{12} + 0.3\varepsilon_4 \quad . \end{aligned} \tag{4.4}$$

Then, when subtracting $y^\sharp$ to $x^\sharp$, we would obtain

$$x^\sharp - y^\sharp = 3 - 0.0\varepsilon_{12} + 0.2\varepsilon_3 - 0.3\varepsilon_4 \tag{4.5}$$

while

$$\hat{x} - \hat{y} = 3 - 0.1\varepsilon_1 + 0.1\varepsilon_2 + 0.2\varepsilon_3 - 0.3\varepsilon_4 \quad . \tag{4.6}$$

In this example, $\gamma_I(\hat{x} - \hat{y}) = [2.3, 3.7] \not\subseteq \gamma_I(x^\sharp - y^\sharp) = [2.5, 3.5]$ which is not what we

expect. Instead of that, we are going to use another way to compress affine forms. Consider again the affine forms $\hat{x} \in \mathcal{F}(\mathcal{I})$ and $\hat{y} \in \mathcal{F}(\mathcal{I})$ of Equation (4.1) .We compress them into

$$\begin{aligned} x^\sharp &= 4 + 0.2\varepsilon_3 + (|0.1| + |0.3|)\varepsilon_5 \\ &= 4 + 0.2\varepsilon_3 + 0.4\varepsilon_5 \end{aligned} \tag{4.7}$$

and,

$$\begin{aligned} y^\sharp &= 1 + 0.3\varepsilon_4 + (|0.2| + |0.2|)\varepsilon_6 \\ &= 1 + 0.3\varepsilon_4 + 0.4\varepsilon_6. \end{aligned} \tag{4.8}$$

Let us already underline that this new method introduces some new over-approximation. Nonetheless, with this new techniques

$$x^\sharp - y^\sharp = 3 + 0.2\varepsilon_3 + 0.3\varepsilon_4 + 0.4\varepsilon_5 - 4\varepsilon_6 \tag{4.9}$$

Here, $\gamma_I(\hat{x} - \hat{y}) = [2.3, 3.7] \subseteq \gamma_I(x^\sharp - y^\sharp) = [1.7, 4.3]$. The result of the other operations (addition and multiplication) of $\hat{x}$ and $\hat{y}$ as well as their compressed versions are given in Table 4.1.

4.1.2 Abstract Domain of Compressed Affine Forms

In this section, we introduce the abstract domain of compressed affine forms. Let $\mathcal{I}$ be a finite set and let $\mathcal{F}(\mathcal{I})$ be the set of affine forms whose noise symbols are indexed by the elements of $\mathcal{I}$. Recall that an affine form $\hat{x} \in \mathcal{F}(\mathcal{I})$ is defined by

$$\hat{x} = c + \sum_{i \in \mathcal{I}} \alpha_i \varepsilon_i \ \ , \tag{4.10}$$

where the $c \in \mathbb{R}$ and for all $i \in \mathcal{I}$, $\alpha_i \in \mathbb{R}$ and the ε_i and $i \in \mathcal{I}$, are formal variables.

To merge noise symbols, we must satisfy the following conditions:

- **Condition 1**: All indices of the original affine form included (in some way) into the compressed form.
- **Condition 2**: Each index must be included exactly once into the compressed affine form.

To do that, let $\mathcal{J}$ be a partition of $\mathcal{I}$ made of k classes $c_1 \subseteq \mathcal{I} \dots c_k \subseteq \mathcal{I}$ and such that $c_1 \cup \dots \cup c_k = \mathcal{I}$ and for all $1 \leq u, v \leq k$ with $u \neq v$, $c_u \cap c_v = \emptyset$. We want to use only one noise symbol j for each class c_j of $\mathcal{I}$ and this symbol must be new for each affine form that we compress for soundness reason (in order to avoid the problem of Equations (4.5) and (4.6)).

An important point is that for the classes $c_j \in J$ which are singletons, no new noise symbols

are assigned. Instead, we keep the old noise symbols. Let $\nu(c_j)$ be a new fresh symbol for the class c_j of partition $\mathcal{J}$. The compression in $\mathcal{F}(\mathcal{J})$ of affine form $\hat{x} \in \mathcal{F}(\mathcal{I})$, for a partition $\mathcal{J}$ of $\mathcal{I}$ is defined by

$$\alpha_{\mathcal{I},\mathcal{J}} : \begin{array}{ll} \mathcal{F}(\mathcal{I}) & \to \mathcal{F}(\mathcal{J}) \\ \hat{x} \to & \alpha_{\mathcal{I},\mathcal{J}}(\hat{x}) = c + \sum_{c_j \in J} \beta_{c_j} \varepsilon_{\nu(c_j)} \quad \text{where} \quad \beta_{c_j} = \sum_{i \in c_j} |\alpha_i| \end{array} \tag{4.11}$$

Example: Let $\hat{x} \in \mathcal{F}(\mathcal{I})$ and $\hat{y} \in \mathcal{F}(\mathcal{I})$ such that

$$\hat{x} = 4 + 0.1\varepsilon_1 + 0.3\varepsilon_2 + 0.2\varepsilon_3 \quad \text{and} \quad \hat{y} = 1 + 0.2\varepsilon_1 + 0.2\varepsilon_2 + 0.3\varepsilon_4 \ .$$

Here $\mathcal{I} = \{1, 2, 3, 4\}$ and let us chose the partition $\mathcal{J} = \{\{1, 2\}, \{3\}, \{4\}\}$ of $\mathcal{I}$ with $c_1 = \{1, 2\}, c_2 = \{3\}$ and $c_3 = \{4\}$. We have

$$\alpha_{\mathcal{I},\mathcal{J}}(\hat{x}) = 4 + 0.2\varepsilon_3 + 0.4\varepsilon_5 \tag{4.12}$$

and,

$$\alpha_{\mathcal{I},\mathcal{J}}(\hat{y}) = 1 + 0.3\varepsilon_4 + 0.4\varepsilon_6 \ . \tag{4.13}$$

Note that in equations (4.12) and (4.13), ε_5 and ε_6 are two new fresh symbols for the class c_1 of $\mathcal{I}$. As already mentioned this avoids the issue illustrated in Section 4.1.1.

Remarks:

- To ensure both conditions 1 and 2, $\mathcal{J}$ must be a partition of $\mathcal{I}$. However if $\mathcal{J}$ is a cover of $\mathcal{I}$, we satisfy condition 1 but not necessarily condition 2.
- The weights of the symbols of a class c_j of $\mathcal{J}$ which are accumulated into a new weight in Equation (4.11) are added in absolute value. This may introduce an over-approximation but it is necessary to avoid the undesirable behavior illustrated in equations (4.3) to (4.6).

Let $\mathcal{I}$ be a finite set of symbols and $\mathcal{J}$ a partition of $\mathcal{I}$. The abstraction function $\alpha_{\mathcal{I},\mathcal{J}}$ merges into one noise symbol all the noise symbols of $\mathcal{I}$ which belong to the same class of $\mathcal{J}$. The coefficient β_{c_j} contains the sum of the $|\alpha_i|$.

For a compressed affine forms $x^\sharp = \sum_{j \in \mathcal{J}} \beta_j \varepsilon_j$ the concretization function $\gamma_{\mathcal{J},\mathcal{I}}$is defined by

$$\gamma^f_{\mathcal{J},\mathcal{I}}(x^\sharp) = \left\{ \hat{x} : \hat{x} = c + \sum_{i \in \mathcal{I}} \alpha_i \varepsilon_i \quad \text{and} \quad \forall c_j \in \mathcal{J}, \sum_{i \in c_j} |\alpha_i| = \beta_j \right\}, \tag{4.14}$$

Property 1 (Soundness of the abstract transfer functions)
Let $\mathcal{I}$ be a finite set of symbols and let $\mathcal{J} = \{c_1, \ldots c_k\}$ be a partition of $\mathcal{I}$. Let

$$x^\sharp = \alpha_{\mathcal{I},\mathcal{J}}(\hat{x}) = c + \sum_{c_j \in \mathcal{J}} \alpha_{c_j} \varepsilon_{\nu(c_j)} \quad \text{and} \quad x'^\sharp = \alpha_{\mathcal{I},\mathcal{J}}(\hat{x'}) = c' + \sum_{c_j \in \mathcal{J}} \alpha'_{c_j} \varepsilon_{\nu(c_j)} \tag{4.15}$$

Figure 4.1: Images used to test the efficiency of affine forms.

be two compressed affine forms of $\mathcal{F}(\mathcal{J})$. For any operation $* \in \{+, -, \times\}$ and for all $\hat{x} \in \gamma_{\mathcal{J},\mathcal{I}}(x^\sharp)$ and $\hat{x}' \in \gamma_{\mathcal{J},\mathcal{I}}(x'^\sharp)$ we have

$$\hat{x} * \hat{x}' \in \gamma_{\mathcal{J},\mathcal{I}}(x^\sharp * x'^\sharp). \tag{4.16}$$

In addition, $\kappa \times \hat{x} \in \gamma_{\mathcal{J},\mathcal{I}}(\kappa \times \hat{x}^\sharp)$.

4.2 Validating Neural Networks with Affine Forms

In this section, we study the influence of the number of noise symbols in affine forms in terms of accuracy and execution time. The results presented hereafter have been obtained with the prototype NNAFF developed during this thesis and introduced in detail in Chapter 6. The experimental protocol is introduced in Section 4.2.1. It is important to note that our choices are representative of other types of architectures, demonstrating that the gains obtained can be generalized to various neural network models. For instance, more complex networks such as AlexNet and VGG16 are examined in Chapter 6, where we will discuss their performance in relation to our findings. The results for the present experiments are provided in sections 4.2.2 and 4.2.3.

4.2.1 Experimental Setting

In this section, we describe our methodology to evaluate the impact in terms of accuracy and execution time of merging noise symbols in neural networks validation. We use a fully connected neural network that serves as a classifier. The network takes as inputs a grayscale image of size $n \times n$, the pixel values being between -1 and 1 (due to some normalization).

Each layer of the network has $n \times n$ neurons, each accepting $n \times n$ inputs. The number of output classes is equal to $n \times n$ where n corresponds to the dimensions of the input image.

Remarks

- Let us underline that we aim to verify the robustness of already trained neural networks without modifying them (typically, these neural networks have been designed by specialists in application domains who do not want the networks to be altered).
- We associate one noise symbol with each pixel, so no compression makes sense before evaluation of the first layer. The compression are applied before the evaluation of the first layer. The compressions are applied starting at the output of the first layer.

For our experiments, the weights of the network have been chosen randomly between -1 and 1 (we use the same network in all our experiments and we have noted that the results do not change significantly when taking another network generated randomly with the same parameters). We consider networks with $L = 2$, 4 and 6 layers. In this experiment, no activation functions (ReLU, $Sigmoid$, $\tanh$, etc.) are taken into account.

The inputs given to our neural networks are square subsets of the six images displayed in Figure 4.1. The original images have a size of 24×24. We use subsets of size 12×12 and 18×18. When working with (usual) affine forms, the input x_0 of the network is a vector of $n \times n$ affine forms which also have $n \times n$ noise symbols each (one noise symbol per pixel). Let c_{ij} be the colour of Pixel (i, j) and ν the intensity of its noise. This pixel corresponds to the component $i \times n + j$ of the input vector x_0 of affine forms whose value is:

$$x_0[i \times n + j] = c_{ij} + \nu c_{ij} \varepsilon_{i \times n + j} \quad . \tag{4.17}$$

In this equation, the term $\varepsilon_{i \times n + j}$ represents white noise. It is important to note that there is no correlation between the noise applied to different pixels.
Note that we associate the same intensity of noise to all the pixels and we use $\nu = 0.2$.

Now, let us focus on the case of compressed affine forms. First, let us underline that we need to use the same neural network that we are validating without modification (otherwise the validation would be biased). We chose to merge each β consecutive noise symbols of the same line in the same new noise symbols. Thus, we considered the case $\beta = 2$ and $\beta = 4$ [1]. Formally, our affine forms compressed have $k = \frac{n \times n}{\beta}$ noise symbols in place of $n \times n$ symbols.

In our experiments, we have measured two quantities: The time needed to execute the neural network with compressed affine forms and the mean width w of the intervals corresponding to the concretizations of compressed affine forms. Since an output vector x_L

[1] In our experiments, we chose a factor β that divides the size of the image to avoid cumbersome implementation details

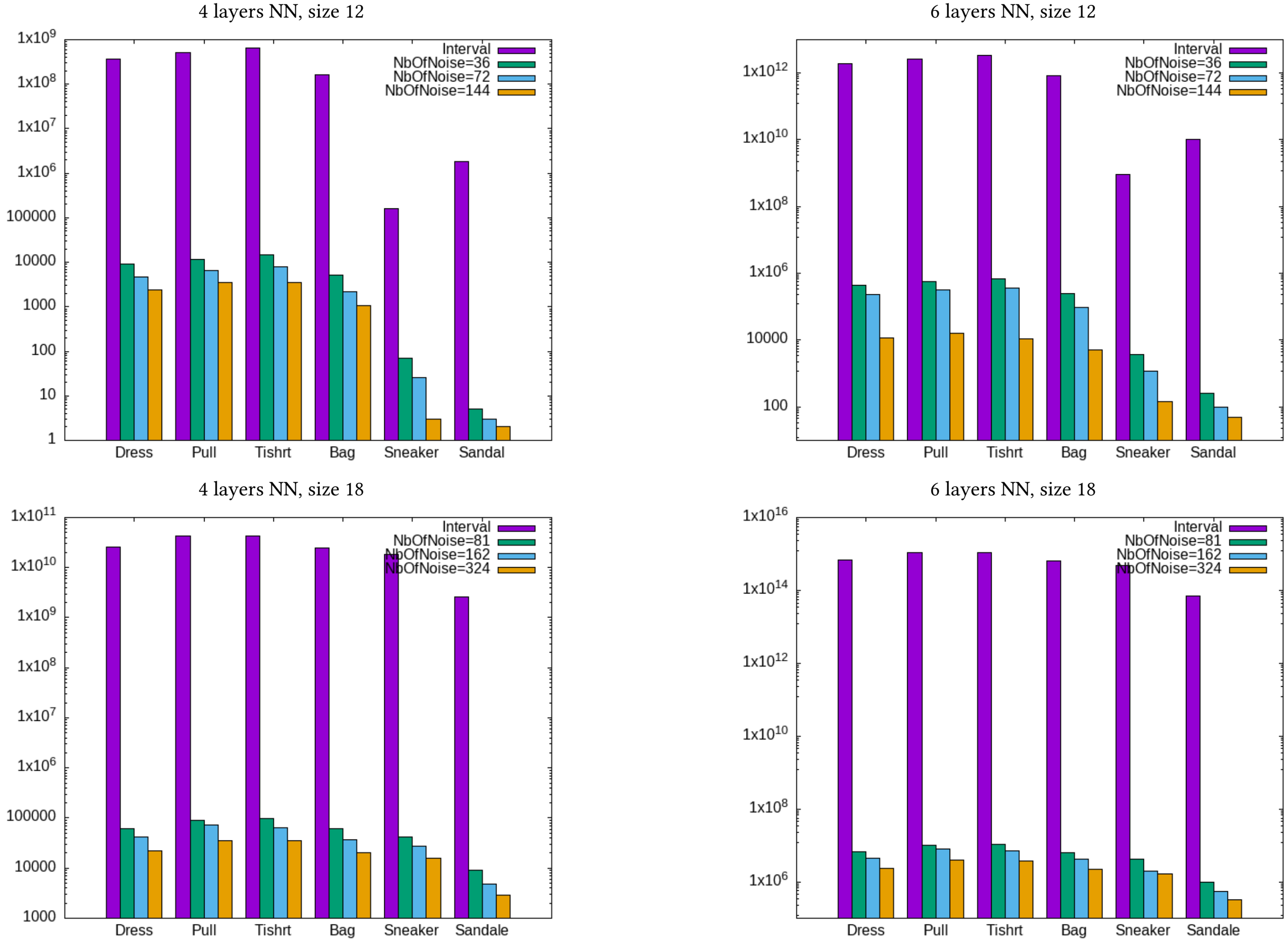


Figure 4.2: Propagation of the affine forms corresponding to our set of images throughout a fully connected neural network with 4 and 6 layers. Input images have size 12×12 (left graph) and 18×18 (right graph).

is made of $N = n \times n$ affine forms, it follows that the output size of the compressed affine forms is also equal to $n \times n$. The use of compressed forms does not change the number of outputs; there are still $n \times n$ compressed affine forms. What changes is the number of ε in each forms. we have

$$w = \frac{\sum_{i=0}^{N} \text{width}\big(\gamma_I(x_L[i])\big)}{N} \quad . \tag{4.18}$$

4.2.2 Influence of Merging Symbols on Accuracy

In this Section, we introduce our experimental results, displayed in Figure 4.2. The histograms display the mean widths of the intervals obtained by concretization of the results, following Equation (4.18). The leftmost histogram corresponds to a neural network made of 4 layers while the rightmost histogram corresponds to a neural network made of 6 layers.

Each histogram is made of 4 sets of bars one for each of the 6 images of Figure 4.1 translated into affine forms as shown in Equations (4.17). For each image of Figure 4.1 in the histogram, we have a set of four bars. Each set of 4 consecutive bars corresponding to the same noise ν is composed as follows: the first bar corresponds to the interval arithmetic, the next 2 bars corresponds to compression noise symbols with $k = 36, 72$ for images of size 12×12 and $k = 81, 162$ for images of size 18×18 and the last bar correspond to the concrete affine form ($k = 144$ for images of size 12×12 and $k = 324$ for images of size 18×18).

As expected, the precision of the affine forms decreases as we move to compressed affine forms, resulting in an increase in the width of the interval concretization. This is because reducing the number of noise symbols decreases the precision of the analysis, with each level of compression further amplifying this effect. Note that, for our examples, the usual interval arithmetic returns an interval whose width has far more orders of magnitude. For example, for $L = 6$, $\nu = 0.2$, for the *Dress* image with size 18×18, the mean widths of the intervals obtained by concretizations are $6.87 \cdot 10^{14}$, $6.51 \cdot 10^{9}$, $4.08 \cdot 10^{8}$ and $2.32 \cdot 10^{7}$, for the interval arithmetic, affine form arithmetic with 81 noises symbols, affine form arithmetic with 162 noises symbols, and affine form arithmetic with 324 noises symbols, respectively. These numbers are representative of what we obtain for the other images and parameters. It shows that the compressed affine forms preserve most of the precision of the affine forms. The loss of precision due to compression seems quite acceptable in regard to the gains in terms of execution time.

We can note that as the number of layers increases, the width of the affine forms increases and we can also remark that the results are similar for all the images.

4.2.3 Influence of Merging Symbols on Time

In this section, we display in Figure 4.3 the execution times taken by the affine forms to evaluate a neural network with images of size 18×18 in function of the number of the noise symbols. Each histogram is made of 4 sets of bars one for each of the 6 images of Figure 4.1. For each image of Figure 4.1 in the histogram, we have a set of four bars. Each set of 4 consecutive bars corresponding to the same noise ν is composed as follows: The first bar corresponds to the interval arithmetic, the next 2 bars correspond to compressed affine forms ($k = 81, 162$) and the last bar corresponds to the concrete affine form.

In the case of 4 layer networks, we remark that the execution time is roughly divided by 2 to 4 when we use compressed affine forms (the execution time increases as the number of noise symbols increases). These results are in adequation with the theoretical complexity: The additions and constant-vector products are linear for the affine form and compressed affine. Note that the observed speedups are important and assess that compressed affine forms are of great interest for neural network verification.

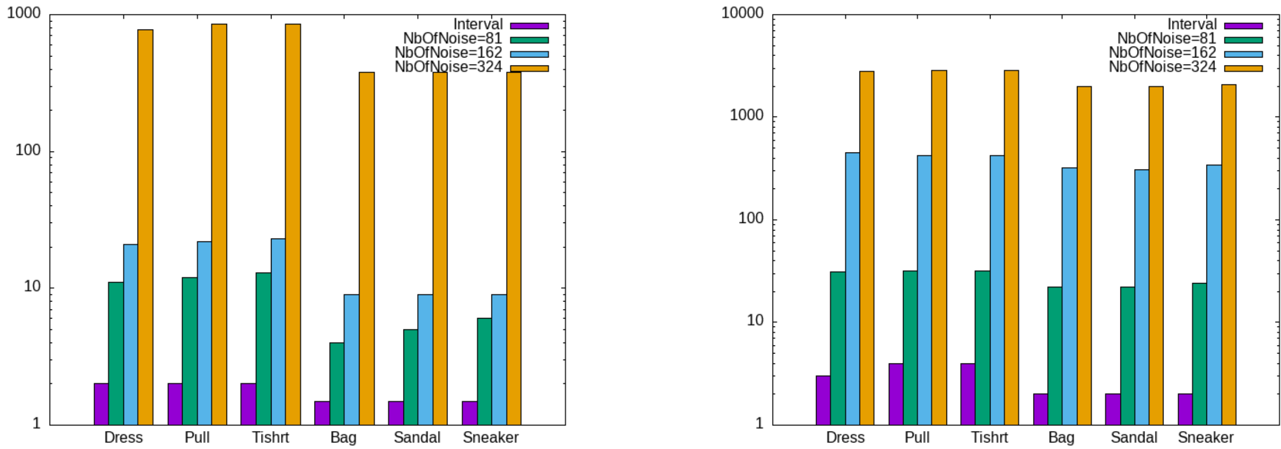


Figure 4.3: Execution time in function of the number of noise symbols. Input images have size 18×18 with 4 layers (leftmost graph) and 6 layers (rightmost graph). We use the six images of Figure 4.1.

4.2.4 Case Study of Convolutional Neural Networks

In this section we will test how well our approach performs on convolutional neural networks. We detail the protocol used in our experiments and present the various results obtained. Furthermore, in Chapter 6, we will test our method on more established networks, such as AlexNet and VGG16, to compare the performance and validate our approach against these well-known architectures.

Protocol

For our experiments, we considered a CNN classifier trained on the Fashion MNIST dataset. The architecture of this CNNs is illustrated in Figure 4.4. We trained our CNNs using a subset of images extracted from the Fashion MNIST dataset, which consists of images of size 28×28. Figure 4.1 shows some examples from this dataset.

In our experiments, we associated the same intensity of noise to all pixels in the images and we considered two cases: $\nu = 0.02$ and $\nu = 0.04$. The input vector was structured as described in Equation (4.17).

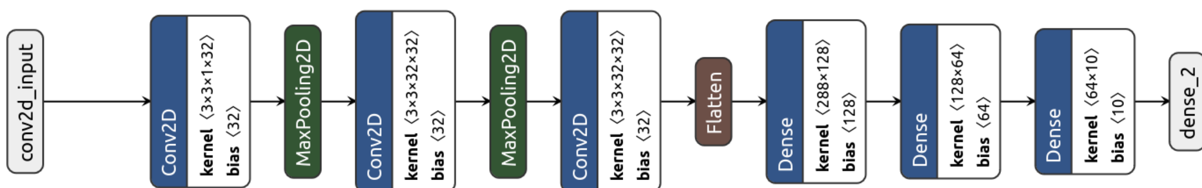


Figure 4.4: The architecture of Model-CNN.

For the case of compressed affine forms, we chose to merging each β consecutive noise symbols of the same line in the same new noise symbols. Thus, we considered the case $\beta = 2$ and 4. We measured two key quantities:

- **Execution Time:** The time needed to execute the CNNs using the traditional affine form, interval arithmetic and our new approach compressed affine forms.
- **Mean Interval Width:** The mean width of the interval corresponds to the concretization of affine forms, interval arithmetic and compressed affine forms, as given in Equation (4.18).

Experimental Results

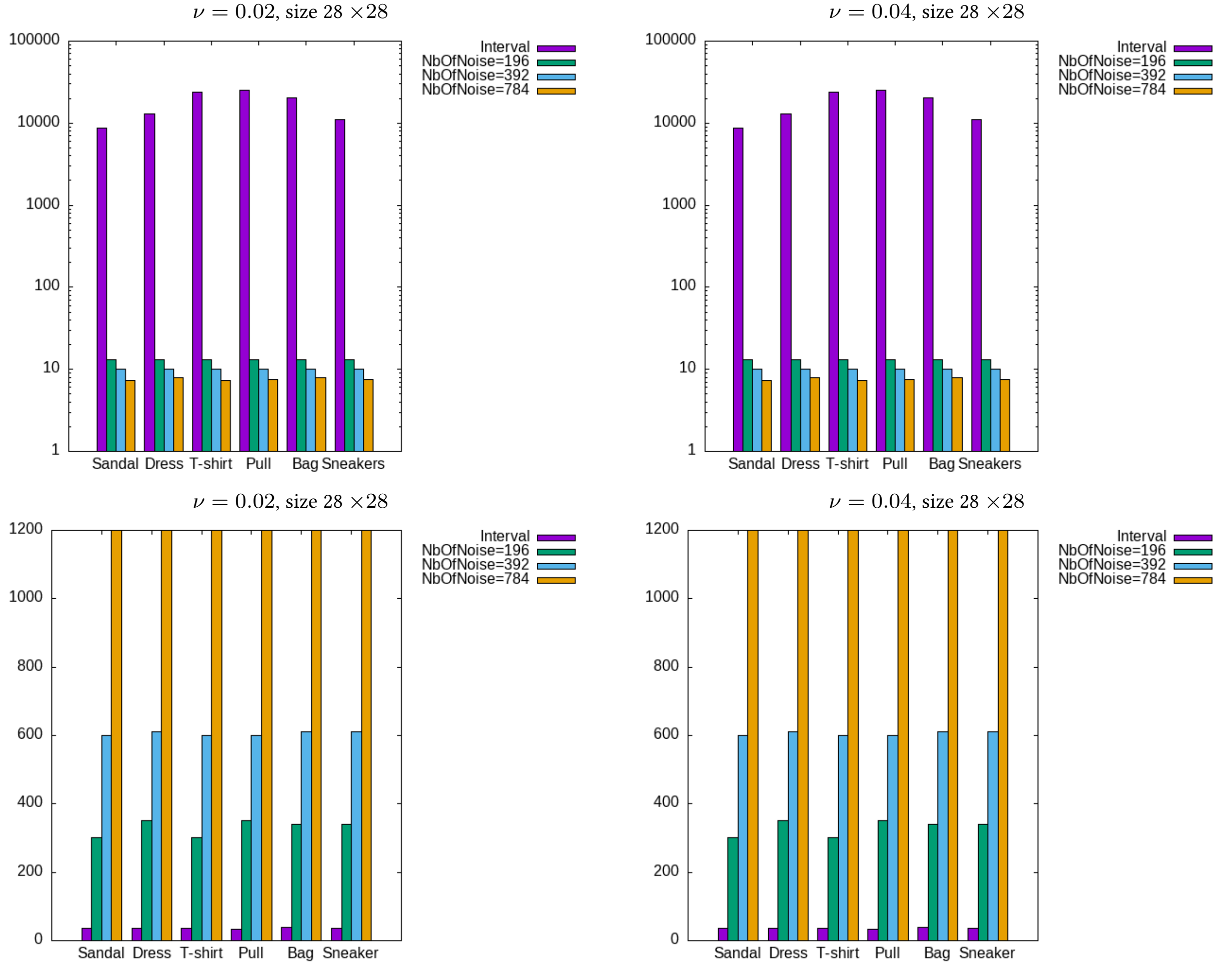


Figure 4.5: Propagation of the affine forms corresponding to our set of images throughout a convolutional neural network. Widths of the concretizations (top row) and execution times (bottom row).

In this Section, we introduce our experimental result, displayed in Figure 4.5. The top column gives the mean widths of the intervals obtained by concretization of the result, fol-

lowing Equation (4.18), and the bottom column gives the execution times for two different cases of noise intensity ($\nu = 0.02$, $\nu = 0.04$). Each histogram is made of 4 sets of bars corresponding to the 6 images of Figure 4.1: the first bar corresponds to the interval forms, the next 2 bars correspond to compression factors of $\beta = 2$ and 4, and the last bar corresponds to the standard affine form.

Concerning the precision of compressed affine forms, we can observe that dividing the number of noise symbols by β roughly multiplies the widths of the intervals obtained by concretization. Comparing this results to interval arithmetic, we can observe that interval arithmetic returns intervals whose widths have far more orders of magnitude and the compressed affine forms preserve most of the precision of the standard affine forms.

The loss of precision due to compression seems quite acceptable in regard to the gains in terms of execution time. We can observe that the execution time is roughly divided by β when we use a compresion factor of β.

Furthermore, we can also remark that the value of ν does not affect the precision and time significantly, and the results are similar for all images.

4.3 Local Noise

In this Section, we discuss another way of reducing the number of noise symbols in affine forms in the context of neural networks taking images as inputs. We split our images into smaller zones and we study the influence of noise symbols in terms of accuracy and times for each zone separately, for which a rationale is provided later. This approach is motivated by the fact that, in order to assert the robustness of neural networks, the users may want, in practice, to know what happens if some part of the image is perturbed. In practice, the perturbation could correspond to a halo of light, a drop of water, fog, etc.

For example, for each image of size 24×24, we divide it into 4 square regions of size 12×12. Then we take the first square and assign to each pixel a noise symbol as defined in Equation (4.17) and we set the noise symbols of other pixels outside the zone to zero. We run our neural networks and repeat the same process for all the other regions.
Note that, we use the experimental setting described in Section 4.2 and we measure two quantities in our experiments: the mean width of the interval corresponding to the concretization of affine forms and the time needed to execute the neural network.

4.3.1 Accuracy in Function of Local Noise

In this section, we experiment with local noise, i.e., we only use non-zero noise symbols for some (square) region of the images.

We present our experimental results, given in Figure 4.7. Figure 4.7 displays the average

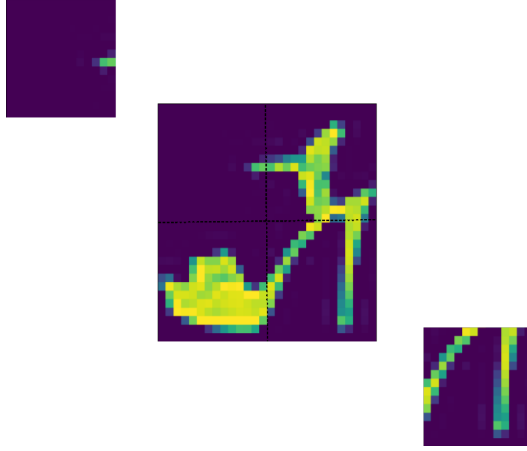


Figure 4.6: The upper left and lower right regions of the Sandal image.

widths of the intervals obtained by concretizing the results. The top line corresponds to the neural networks made of 4 and 6 layers with local size 12×12, while the bottom line corresponds to neural networks made of 4, 6 layers with local size 8×8 (note that, the size of our images is 24×24).

Each histogram is made of sets of bars corresponding to the 4 images of Figure 4.1 (*T-shirt, Bag, Sandal, Sneaker*) translated into affine forms as shown in Equation (4.17). For each frame in the histogram, we have a series of bars:

- For each image with local size 12×12 we have 4 bars, the first corresponds to the upper left region of our images and the third corresponds to the lower right region,
- For each image with local size 8×8 we have 6 bars composed as follows: the first corresponds to the upper left region of our images, the third corresponds to the middle region, and the fifth corresponds to the lower right region.

Let us also mention that, in all our experiments, we have used the same initial noise $\nu = 0.2$.

Regarding the accuracy of the affine forms, it can be seen that the widths obtained by concretization varie in function of each image and region. For example, for the *Sandal* image with local size 8×8, the mean widths of the intervals obtained by concretization are 0, 7.32×10^2, 3.15×10^4 for the upper left region, middle region and the lower right region, respectively. It shows that the lower right region of this image (*Sandal*) introduces the overapproximation.

4.3.2 Execution Time in Function of Local Noise

In this section, we display in Figure 4.8 the execution times taken by the affine forms to run a neural networks with images of size 24×24 with local size 12×12 and 8×8 in function of the number of layers.

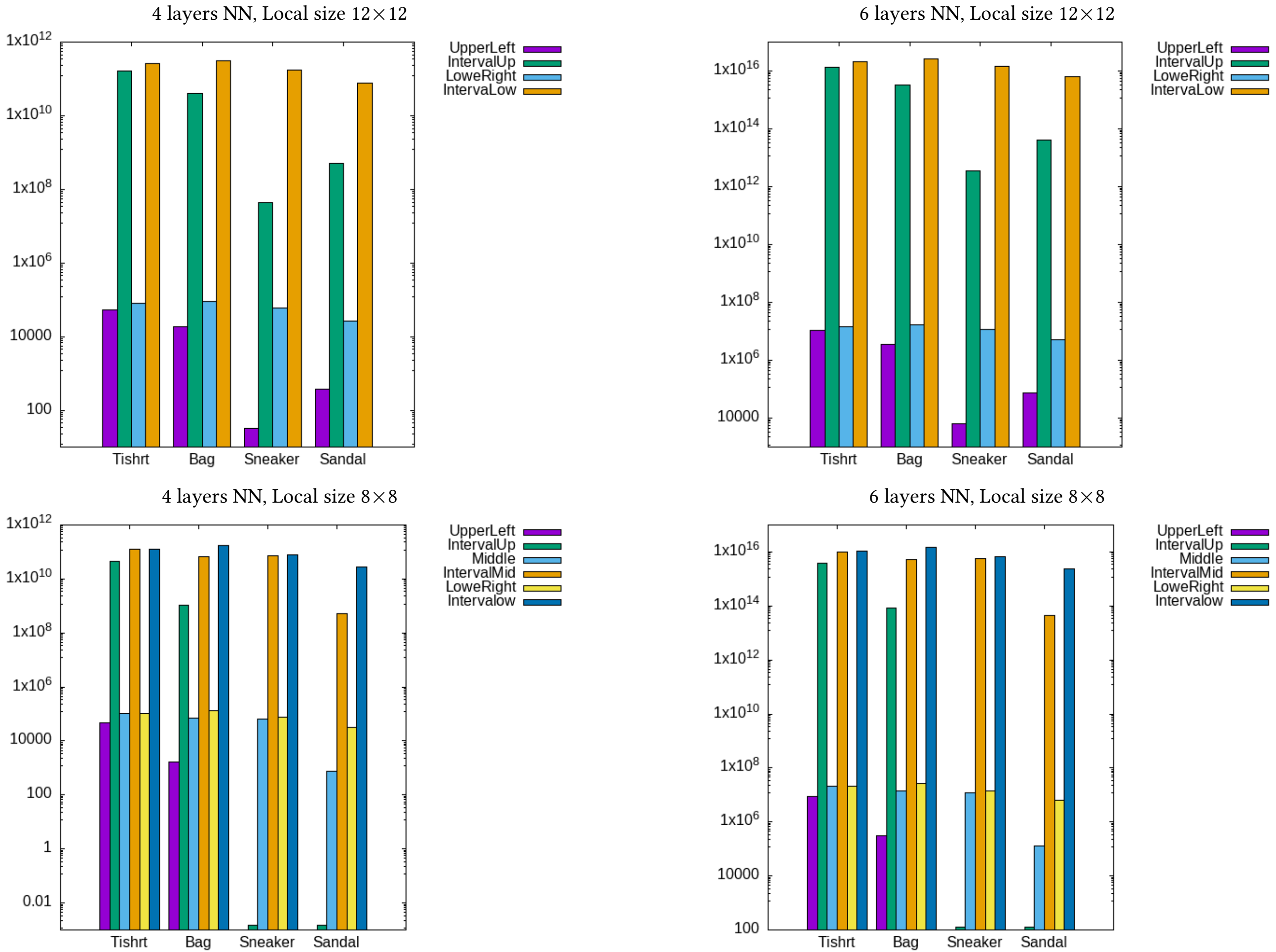


Figure 4.7: Propagation of the affine forms corresponding to our set of images throughout a fully connected neural network with 4 and 6 layers. Input images have size 24×24.

The leftmost graphs of Figure 4.8 correspond to the execution time for the upper left region, while the rightmost graphs correspond to a execution time for the lower right region for images with a local size 12×12 and the middle region for images with a local size 8×8.

We can remark that the execution time differs from one image to another and in function of region that we consider. This is mainly due to the fact that our images have more or less pixels set to 0 in the considered zones. Nevertheless, the speed-ups naturally increase as the number of noise symbols that we merge increases. In addition, let us mention that these speed-ups are important in regards to the acceptable precision losses observed in Figure 4.7.

4.4 Conclusion

In this chapter, we have studied the influence of merging or discarding noise symbols in affine forms in the context of neural network validation. The affine forms are more precise

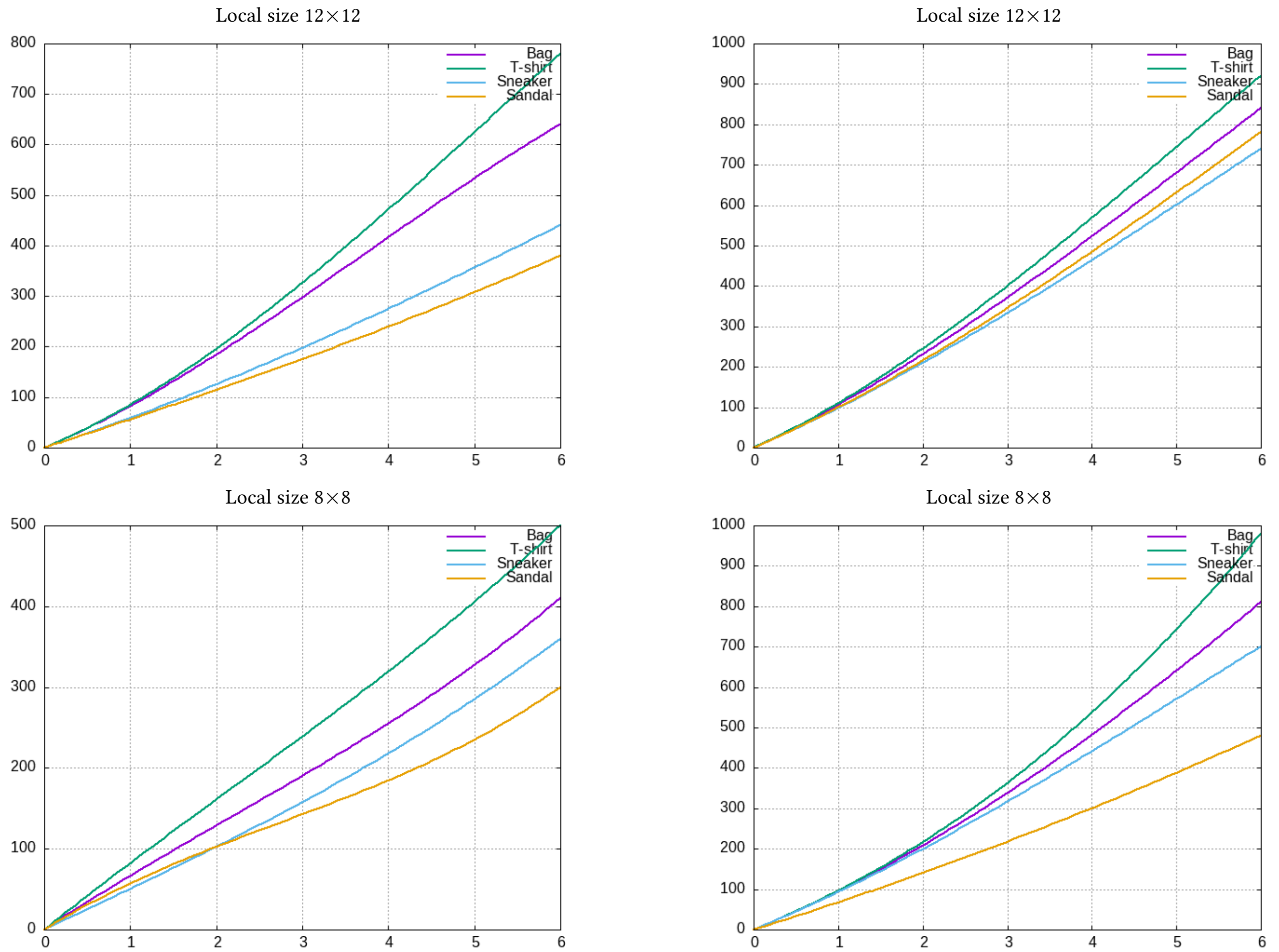


Figure 4.8: Execution time in function of the number of layers. Input images have size 24×24. We consider the four images of Figure 4.1.

than interval arithmetic but they take a lot of memory and compute more slowly when the number of noise symbols increases. The experimental results of Section 4.2 confirm that compressed affine forms offer important gains in terms of execution time with an acceptable penalty in terms of precision.

Chapter 5

Scaling-up the Analysis of Neural Networks by Affine Forms

5.1 Overview 66
5.1.1 Activation Functions 67
5.2 Block-wise Noising 68
5.3 Efficiency of Block-Wise Noising 71
5.3.1 Experimental Setting 71
5.3.2 Execution Time 72
5.3.3 Experimenting with Trained Neural Networks 74
5.4 Block-Wise Noising Parallelism 75
5.4.1 Execution Time 77
5.4.2 Speedup Analysis 78
5.5 Conclusion 78

The efficiency of neural networks in handling visual perturbations is frequently assessed using abstract transforms [38, 105], such as affine transformations [104]. However, these transforms may forfeit precision and be computationally expensive (time and memory consuming). We suggest in this chapter 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 perturbated 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 its scalability and accuracy.

This chapter is organized as follows. Section 5.1 explains how block-wise noising works and outlines how to over-approximate the ReLU activation function; Section 5.2 introduces the block-wise noising approach in detail. Section 5.3 focuses on the efficiency of the method by providing an experimental setup and exprimental results on the time spent during the evaluation of neural networks. Parallel block wise noising is reviewed in Section

5.4. Section 5.5 concludes.

The work introduced below is excerpted from the article “Scaling-up the Analysis of Neural Networks by Affine Forms: A Block-Wise Noising Approach”, which has been published at SYNASC (International Symposium on Symbolic and Numeric Algorithms for Scientific Computing 2023)[107].

5.1 Overview

In this section, we introduce informally our block-wise noising technique and illustrate how to compute with it. These ideas are formalized further in Section 5.2. In our example, illustrated in Figure 5.1, 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 tailored for the analysis of neural networks. So noise is attached to the input x while W and b correspond to the weights of the NN and contain scalar coefficients.

x

$2+\varepsilon_1$	$3+2\varepsilon_2$
$1-\varepsilon_3$	$4+2\varepsilon_4$

W

1	0	2	1
1	-1	0	1

b

-1
0

(a) Inputs given to a fully connected layer.

$$\left(\underbrace{\begin{bmatrix} 1 & 0 & 2 & 1 \end{bmatrix}}_{W} \times \underbrace{\begin{bmatrix} 2+\varepsilon_1 \\ 3+2\varepsilon_2 \\ 1-\varepsilon_3 \\ 4+2\varepsilon_4 \end{bmatrix}}_{x} + \underbrace{\begin{bmatrix} -1 \end{bmatrix}}_{b} \right)$$

(b) The first operation of the fully connected layer.

$$\left(\underbrace{\begin{bmatrix} 1 & -1 & 0 & 1 \end{bmatrix}}_{W} \times \underbrace{\begin{bmatrix} 2+\varepsilon_1 \\ 3+2\varepsilon_2 \\ 1-\varepsilon_3 \\ 4+2\varepsilon_4 \end{bmatrix}}_{x} + \underbrace{\begin{bmatrix} 0 \end{bmatrix}}_{b} \right)$$

(c) The second operation of the fully connected layer.

Figure 5.1: Fully connected layer operations.

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:

$$y_1 = 7 + \varepsilon_1 - 2\varepsilon_3 + 2\varepsilon_4, \quad y_2 = 3 + \varepsilon_1 - 2\varepsilon_2 + 2\varepsilon_4. \tag{5.1}$$

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 5.4). For example, in the case of Block 1 we add non-

Block 1

$2+\varepsilon_1$	$3+0\varepsilon_2$
$1-\varepsilon_3$	$4+0\varepsilon_4$

Block 2

$2+0\varepsilon_1$	$3+2\varepsilon_2$
$1+0\varepsilon_3$	$4+2\varepsilon_4$

Figure 5.4: Splitting of x into two block-wise noising.

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 zero noise symbols for the blue parts. We compute fully connected layers for Block 1 and Block 2. For Block 1, we have

$$b_{11} = 7 + \varepsilon_1 - 2\varepsilon_3, \quad b_{12} = 3 + \varepsilon_1, \tag{5.2}$$

and for Block 2, we have

$$b_{21} = 7 + 2\varepsilon_4, \quad b_{22} = 3 - 2\varepsilon_2 + 2\varepsilon_4. \tag{5.3}$$

We obtain that

$$b_{11} + b_{21} = 14 + \varepsilon_1 - 2\varepsilon_3 + 2\varepsilon_4 \quad = y_1 + 7, \tag{5.4}$$

and,

$$b_{12} + b_{22} = 6 + \varepsilon_1 - 2\varepsilon_2 + 2\varepsilon_4 \quad = y_2 + 3. \tag{5.5}$$

The constants, 7 and 3 subtracted from y_1 and y_2, are due to the fact that centers are added twice in the formula and must be removed (see Section 5.2 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.

5.1.1 Activation Functions

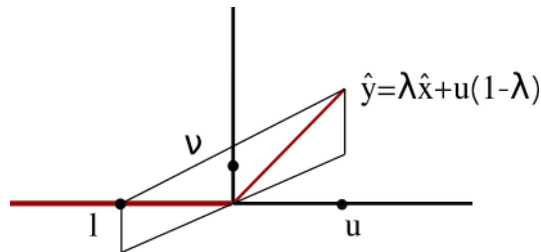


Figure 5.5: The zonotope approximation of the Relu function.

Affine forms are a successful approach for verifying neural networks, as they are fast and exact for affine transformations [106]. However, when it comes to modeling nonlinear activation functions such as ReLU, the zonotope abstraction [27] is not exact. Therefore, an approximation technique [38, 62] that creates a tradeoff between computational cost and precision is necessary. Following the approach developed in [104], we present hereafter an approximation method that strikes a balance between computational cost and precision.

Let $\hat{x}$ be an affine form $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i$ given to a ReLU function (y = ReLU($\hat{x}$)). 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 $\hat{x}$. The abstract transformer of the ReLU activation function is given by:

$$ReLU(\hat{x}) = \begin{cases} \hat{x} & \text{for } l_x \geq 0, \\ 0 & \text{for } u_x \leq 0, \\ \lambda \hat{x} + \nu + \nu\varepsilon & \text{otherwise.} \end{cases}$$

where $\lambda = \frac{u_x}{u_x - l_x}$ and $\nu = \frac{u_x(1-\lambda)}{2}$ 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 5.5).

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

$$ReLU(\hat{x}) = 1.8 + 1.2\varepsilon_1 + 1.8\varepsilon_2 + 1.2\varepsilon_3 \ . \tag{5.6}$$

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 3.11 into an interval.

$$\gamma_I(\hat{x}) = x_0 + \left(\sum_{i=1}^{n} |x_i| \right) \times [-1, 1]. \tag{5.7}$$

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

5.2 Block-wise Noising

In this section, we describe in details how our approach called block-wise noising works. 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 use the example of Figure 5.6 in order to clarify this approach. We split our input into n regions (in our example we take $n = 4$). Then, we add non-zero noise symbols only

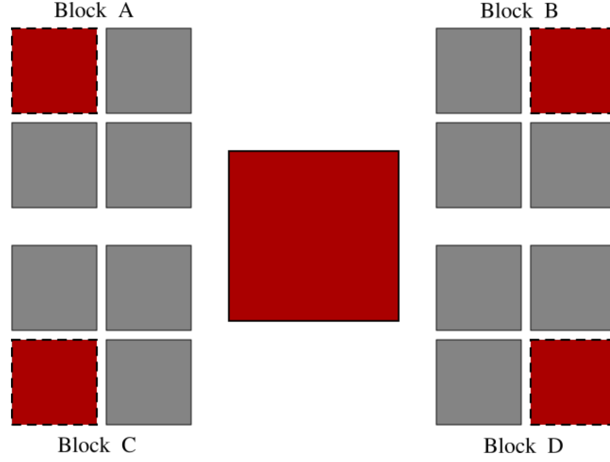


Figure 5.6: Splitting input into block-wise noising.

to one region (red parts) and zero noise symbols for all other regions (gray parts). We repeat this process for 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 1 *Let $\hat{x} \in \mathcal{F}(\mathcal{I})$ and $\hat{y} \in \mathcal{F}(\mathcal{I})$ be two affine forms defined by :*

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \ , \ \text{ and } \ \hat{y} = y_0 + \sum_{i=1}^{n} y_i \varepsilon_i \ . \tag{5.8}$$

If $n = 2^u$, then $\hat{x}$ and $\hat{y}$ can be split into k pieces of size 2^v and $k \times 2^v = 2^u$. Let

$$\hat{x}^{'} = \sum_{i=1}^{k} a_i \ , \ \text{ with } \ a_i = x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \ , \tag{5.9}$$

and,

$$\hat{y}^{'} = \sum_{i=1}^{k} b_i \ , \ \text{ with } \ b_i = y_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} y_j \varepsilon_j \ . \tag{5.10}$$

Then we have,

$$\hat{x} + \hat{y} \ = \ \hat{x}^{'} + \hat{y}^{'} - (k-1)(x_0 + y_0) \ . \tag{5.11}$$

However, this equality is valid only when $k \times 2^v = 2^u$. If this condition is not satisfied, the result does not hold.

Proof Let

$$\hat{x}^{'} = \sum_{i=1}^{k} a_i \ , \ \text{ with } \ a_i = x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \ , \tag{5.12}$$

And,

$$\hat{y}' = \sum_{i=1}^{k} b_i \ , \ \text{with} \ b_i = y_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} y_j \varepsilon_j \ . \tag{5.13}$$

We have,

$$\begin{aligned}
\hat{x}' + \hat{y}' &= \sum_{i=1}^{k} (a_i + b_i), \qquad (5.14)\\
&= \sum_{i=1}^{k} \left(x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \right) + \sum_{i=1}^{k} \left(y_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} y_j \varepsilon_j \right)\\
&= \sum_{i=1}^{k} (x_0 + y_0) + \sum_{i=1}^{k} \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} (x_j + y_j)\varepsilon_j\\
&= \sum_{i=1}^{k} (x_0 + y_0) + \sum_{i=1}^{n} (x_j + y_j)\varepsilon_j \qquad (5.15)
\end{aligned}$$

Then,

$$\hat{x} + \hat{y} \ = \ \hat{x}' + \hat{y}' - (k-1)(x_0 + y_0) \ . \tag{5.16}$$

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

Theorem 2 *Let $\hat{x} \in \mathcal{F}(\mathcal{I})$ be an affine form defined by :*

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \ . \tag{5.17}$$

If $n = 2^u$, then $\hat{x}$ can be split into k pieces of size 2^v and $k \times 2^v = 2^u$. Let

$$\hat{x}' = \sum_{i=1}^{k} a_i \ , \ \textit{with} \ a_i = x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \ . \tag{5.18}$$

Then we have,

$$\hat{x} \times c = \hat{x}' \times c - (k-1)(x_0 \times c) \ .$$

However, this equality is valid only when $k \times 2^v = 2^u$. If this condition is not satisfied, the result does not hold.

PROOF Let

$$\hat{x}' = \sum_{i=1}^{k} a_i \ , \ \text{with} \ a_i = x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \ . \tag{5.19}$$

We have,

$$\begin{aligned}
\hat{x}' \times c &= \sum_{i=1}^{k} c \times a_i && (5.20)\\
&= \sum_{i=1}^{k} c \times \left(x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \right)\\
&= \sum_{i=1}^{k} x_0 \times c + \sum_{i=1}^{k} \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \times c\\
&= \sum_{i=1}^{k} x_0 \times c + \sum_{i=1}^{n} x_j \varepsilon_j \times c.
\end{aligned}$$

Then,

$$\hat{x} \times c = \hat{x}' \times c - (k-1)(x_0 \times c) \ .$$

5.3 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 5.3.1 and report the results in Sections 5.3.2 and 5.3.3.

5.3.1 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.

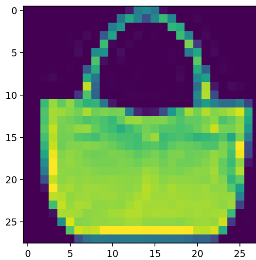


Figure 5.7: Image used to test the efficiency of block-wise noising.

Fully Connected Feedforward Networks: we use networks consisting of $L = 2, 4$ and 6 layers. Each layer contains 28×28 neurons. The input given to our neural networks is

displayed in Figure 5.7. The network 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:

$$x_0[i \times n + j] = c_{ij} + \nu c_{ij} \varepsilon_{i \times n + j} \quad . \tag{5.21}$$

Note that we associate the same intensity of noise with 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 of 28×28 with 28×28 noise symbols (one noise symbol per pixel). Each convolutional network consists 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 cases where our input is split into 4, 8 and 16 blocks.

Dataset	1 blocks	4 blocks	8 blocks	16 blocks
Sac	5 h 33 min	2 h 32 min	2 h 13 min	2 h 12 min
Sneakers	5 h 17 min	2 h 25 min	1 h 58 min	1 h 55 min
Dress	5 h 50 min	2 h 33 min	2 h 15 min	2 h 12 min
T-shirt	5 h 50 min	2 h 33 min	2 h 15 min	2 h 12 min

Table 5.1: Execution time for dataset MNIST with fully connected neural networks at 2 layers.

In our experiments, we have 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.

5.3.2 Execution Time

Our experimental results are displayed in Figures 5.8 and 5.9 which represent the execution time taken to run a fully connected feedforward neural network and a 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.

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

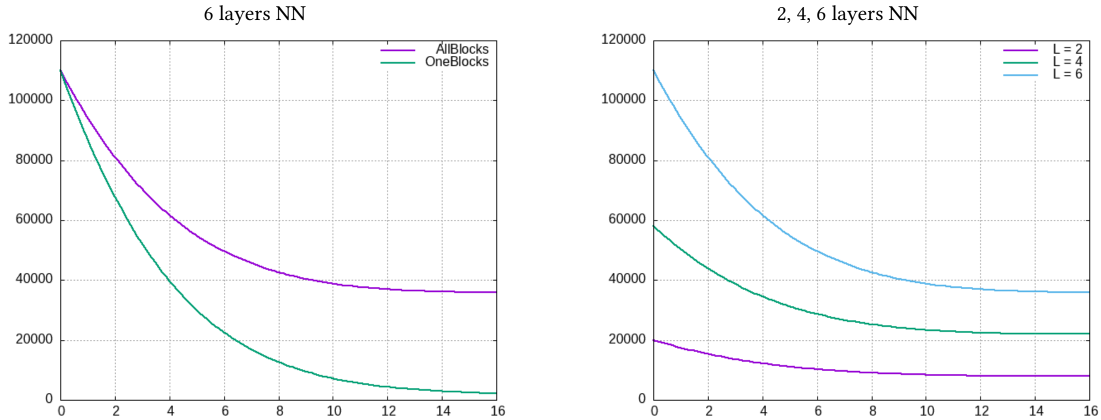


Figure 5.8: Execution time in function of the number of blocks for AllBlocks (Block A, Block B, Block C, Block D) and OneBlock (Block A) of figure 5.6. Input image has a size 28×28 with 2, 4 and 6 layers for the fully connected neural network. We use the image of Figure 5.7.

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 $58000s$, $35000s$, $22000s$ and $21000s$. For 1, 4, 8, and 16 blocks, respectively. The results show that the block-wise noising divided the execution times by 2. As seen in Table 5.1, these results hold true for all input types.

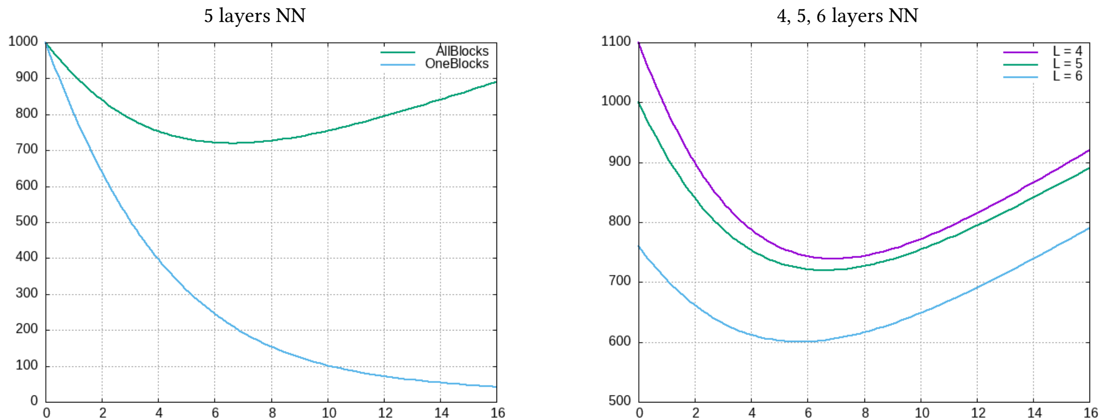


Figure 5.9: Execution time in function of the number of blocks for AllBlocks (Block A, Block B, Block C, Block D) and OneBlock (Block A) of Figure 5.6. Input image has a size 28×28 with 4, 5 and 6 layers for the convolutional neural network. We use the image of Figure 5.7.

Additionally, in the case of 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 8. For example, for $L = 5$ the execution times taken by the neural networks are $1000s$, $750s$, $720s$ and $890s$. 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 evaluated than fully connected layers. This can be explained by the fact that convolutional layers use spatial locality by sharing weights and performing 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.

5.3.3 Experimenting with Trained Neural Networks

In this section, we introduce some experimental results using five trained neural networks. All these NNs are classifiers. They are described in Table 5.2. 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.1.

We use three popular datasets for our experiments: MNIST, Fashion-MNIST [33] and CIFAR-10 [1]. MNIST and Fashion-MNIST contains $60,000$ grayscale images of size (28×28) and Cifar-10 contains $60,000$ grayscale images of 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 defined in Section 5.1.1.

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	$111,146$

Table 5.2: configurations of the model used in our experiments.

All experiments were performed with Python on a cluster named MUSE [1], which consisted of 308 PowerEdge $C6320$ servers based on Intel Xeon $E5-2680v4$ chips, and each

[1]is a high-performance computing cluster at the University of Montpellier, delivering 280 teraflops of com-

server had $128GB$ of RAM.

Figure 5.3 displays the results of our measurement of the time required to run each neural network with various amounts of blocks.

We note 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 uses the Fashion-MNIST dataset, the execution times are 3 h 37 min, 1 h 35 min and 1 h 25 min for 1, 4 and 8 blocks, respectively. We observe that when we use our block-wise noising method with fourth block, the execution time is roughly divided by 2.

Model	Input	1 blocks	4 blocks	8 blocks
FashionMNIST				
CNN	(28×28)	1 h 35 min	1 h 6 min	1 h 15 min
FC	(28×28)	3 h 37 min	1 h 35 min	1 h 25 min
CIFAR				
CNN	(32×32)	5 h 33 min	4 h 10 min	4 h 27 min
MNIST				
CNN	(28×28)	1 h 33 min	1 h 12 min	1 h 28 min
FC	(28×28)	18 min	8 min	7 min

Table 5.3: Execution time with fully connected and convolutional neural networks at different depths.

We can observe that in the case of convolutional networks, the execution times decrease as the number of input blocks grows, but they start to increase once the number of input blocks exceeds 4. For example, for CNN that uses the Fashion-MNIST dataset, the execution times are 1 h 35 min, 1 h 6 min and 1 h 15 min, 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 redundant calculations. In other words, convolution operations are repeated for pixels that are common to different blocks, resulting in an increase of the execution time compared to fully connected layers.

5.4 Block-Wise Noising Parallelism

The computational cost of training and inference increases along with the complexity of neural networks. To address this, parallelization methods have emerged as a crucial way of

puting power. Developed with Dell EMC, it supports advanced research and simulations for local scientists, industrialists, and startups.

reducing neural network execution time. Data parallelism [70] and model parallelism [58, 101] are some of the commonly used ways to parallelize neural networks. Nevertheless, in our study, we are interested in parallelizing 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 5.2) to generate a simple model for parallel implementation.

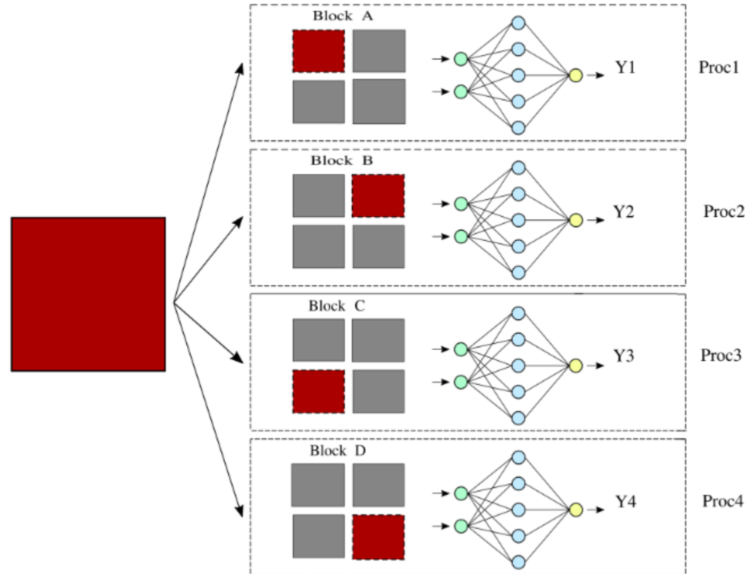


Figure 5.10: Block-Wise Noising Parallelism.

One strategy to parallelize the block-wise noising is to distribute the computations of each block (Block A, Block B, Block C and Block D) as shown in Figure 5.10 between the different processors and performs those computations simultaneously:

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

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 5.3.3 and we measure in our experiments the time needed to evaluate 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.

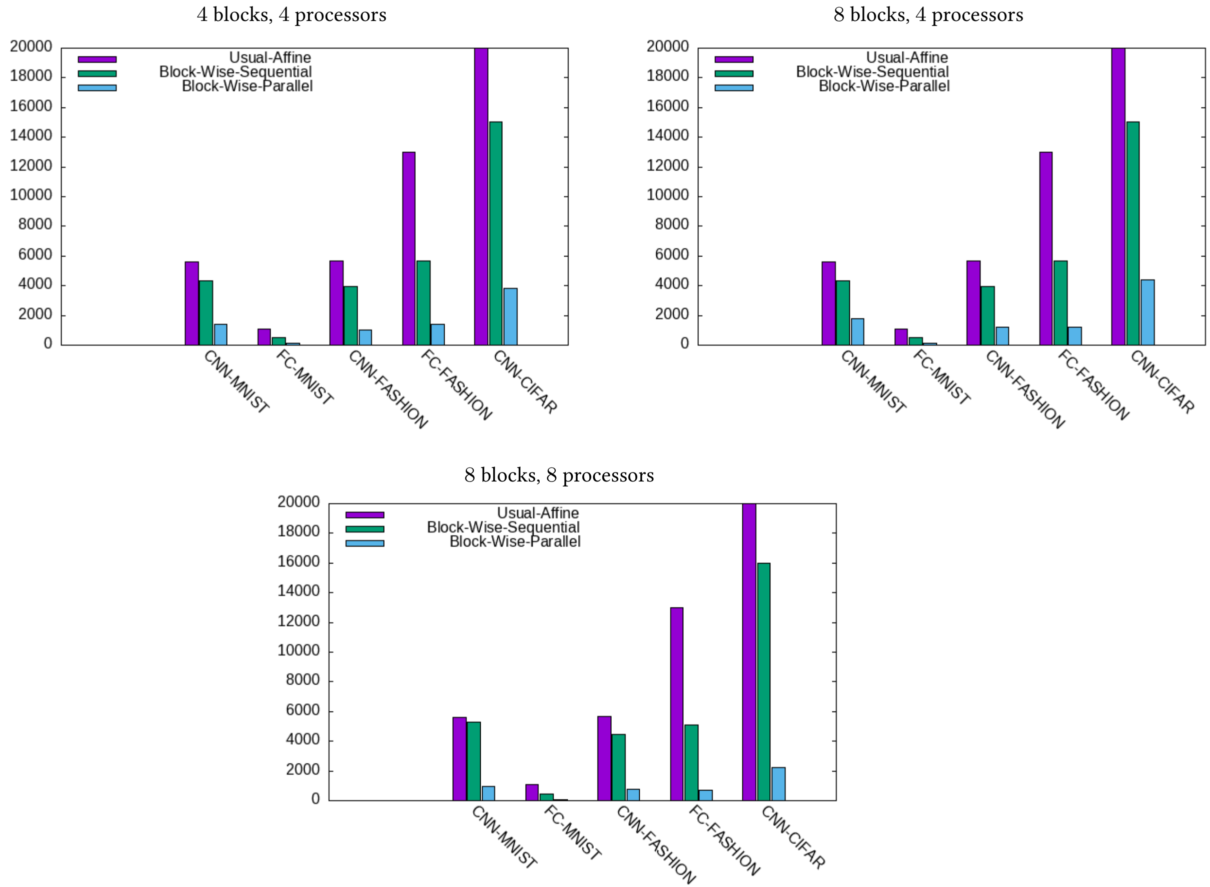


Figure 5.11: Execution time of Block-Wise Noising parallelism.

5.4.1 Execution Time

The execution time of the trained neural networks (NNs) described in Appendix A.1 is shown in Figure 5.11. The top-right histogram represents NN evaluations with 8 blocks and 4 processors (each processor calculates 2 blocks), the top-left histogram corresponds to NNs evaluated with 4 blocks and 4 processors, and the bottom histogram corresponds to NNs evaluated 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 results show that the execution time obtained using 4 and 8 processors are faster than those obtained with the sequential version. As an example, for the CNN-CIFAR model the execution times obtained with the 4 blocks and 4 processors are 5 h 33 min, 4 h 10 min and 56 min, for usual affine forms, block wise noising sequential and block wise noising parallel, respectively. In addition the execution time starts to increase when the number of blocks exceeds the number of processors (8 blocks for 4 processors, with each processor computing 2 blocks). For instance, the CNN-CIFAR model execution time is 1 h 13 min.

5.4.2 Speedup Analysis

In order to evaluate the block-wise noising parallel performance, we compute the speedup [114], which represents a correlation between the block-wise noising serial execution time and the parallel execution time in p processors ($p = 4$ and $p = 8$). In other words, we calculate the speedup s defined by

$$s = \frac{T_s}{T_p}.$$

The speedup is given in Table 5.4.

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 5.4: Speed-Up results.

We note that block-wise-noising parallel has been found to achieve a speedup of 3.1 to 4 for $p = 4$ and 5.5 to 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.

5.5 Conclusion

In this chapter, we have presented our second contribution in the field of neural network verification using abstract interpretation. We have developed sequential and parallel versions of block-wise noising for the analysis of neural networks by affine forms. We evaluated the efficiency of block wise noising on the 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 cases.

Chapter 6

The NNaff Tool

6.1 Mixed Affine Approach 80
6.1.1 Principle of The NNaff Tool 80
6.2 The NNaff Tool 82
6.2.1 Architecture of The Tool 82
6.2.2 Using the Tool 83
6.2.3 Experimental Results 85
6.3 Summary 87

The current chapter introduces NNaff, short for Neural Networks Affine Forms, software tool that has been developed as part of this thesis for neural network verification. This tool relies on the concepts and methodologies developed in chapters 4 and 5, respectively. It is another contribution to this thesis.

This chapter is organized as follows. Section 6.1 presents the Mixed Affine Approach, which combines our two approaches, affine compressed and block-wise noising. Section 6.2 takes an in-depth look at the NNaff tool itself. We start by providing an overview of its architecture, highlighting key components and how they interact with each other. Then we investigate how users can best use the tool with detailed usage instructions, including some examples. Next, we give experimental results that show the usefulness of NNaff in practical cases. Finally, we end up with a summary of this chapter, which recaps all the discussed issues and reinforces the position of the NNaff tool in relation to neural network verification.

6.1 Mixed Affine Approach

In this section, we will introduce our new approach called mixed affine, and we will illustrate how calculations are performed using this method.

6.1.1 Principle of The NNaff Tool

The mixed affine approach merges the principles of compressed affine forms as introduced in Chapter 4 and block-wise noising described in Chapter 5 at the same time. In this method, given an input x, we divide x into n blocks. Within each region of every block, compressed affine forms are used. In other words, we aggregate sets of k successive noise symbols into a single new symbol (see Figure 6.1). This approach aims to achieve more efficient results in terms of execution time compared to employing each method separately. In order to

Figure 6.1: Mixed affine approach for $n = 4$.

understand the principle of mixed affine approach, we present the following theorem for the case of the addition of two affine forms.

Remark:

The difference between Theorems 3 and 4, compared to Theorems 1 and 2, is that since the first compression of symbols $\alpha_{\mathcal{I},\mathcal{J}}$ introduces an overapproximation, an overapproximation will also result from the composition of this compression with the block-wise noising technique.

Theorem 3 *Let $\hat{x} \in \mathcal{F}(\mathcal{I})$ and $\hat{y} \in \mathcal{F}(\mathcal{I})$ be two affine forms defined by*

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \text{ and } \hat{y} = y_0 + \sum_{i=1}^{n} y_i \varepsilon_i \ . \tag{6.1}$$

Let us assume that $n = 2^u$ and that, consequently, we can split $\hat{x}$ and $\hat{y}$ into k blocks of size 2^v with $k \times 2^v = 2^u$. Let

$$\hat{x}' = \sum_{i=1}^{k} a_i \;\; \textit{with} \;\; a_i = x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \;\; , \tag{6.2}$$

and

$$\hat{y}' = \sum_{i=1}^{k} b_i \;\; \textit{with} \;\; b_i = y_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} y_j \varepsilon_j \;\; . \tag{6.3}$$

Let $\mathcal{J} = \{c_1, \ldots c_l\}$ a partition of $\mathcal{I}$ such that each interval $[(i-1)2^v+1, i\times 2^v], 1 \le i \le k$ is included in the same class of $\mathcal{J}$: $\forall i, 1 \le i \le k, \exists j : [(i-1)2^v+1, i \times 2^v] \subseteq c_j$. Let

$$x^\sharp = \sum_{i=1}^{k} a_i \;\; \textit{with} \;\; a_i = x_0 + \sum_{c_j \in \mathcal{J}} \beta_{c_j} \varepsilon_{\nu(c_j)} \quad \textit{where} \quad \beta_{c_j} = \sum_{i \in c_j} |x_i| \;\; , \tag{6.4}$$

and

$$y^\sharp = \sum_{i=1}^{k} a_i \;\; \textit{with} \;\; a_i = y_0 + \sum_{c_j \in \mathcal{J}} \beta_{c_j} \varepsilon_{\nu(c_j)} \quad \textit{where} \quad \beta_{c_j} = \sum_{i \in c_j} |y_i| \;\; . \tag{6.5}$$

Then

$$\hat{x} + \hat{y} \;\; \in \;\; \gamma_{\mathcal{J},\mathcal{I}} \left(x^\sharp + y^\sharp - (k-1)(x_0 + y_0) \right) \;\; . \tag{6.6}$$

However, this equality is valid only when $k \times 2^v = 2^u$. If this condition is not satisfied, the result does not hold.

Let us consider now the multiplication of a mixed affine form by a constant. The following theroem provides a formal statement for this operation:

Theorem 4 *Let $\hat{x} \in \mathcal{F}(\mathcal{I})$ be an affine form defined by*

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i \;\; . \tag{6.7}$$

Let us assume that $n = 2^u$ and that, consequently, we can split $\hat{x}$ into k blocks of size 2^v with $k \times 2^v = 2^u$. Let

$$\hat{x}' = \sum_{i=1}^{k} a_i \;\; \textit{with} \;\; a_i = x_0 + \sum_{j=(i-1)\times 2^v+1}^{2^v \times i} x_j \varepsilon_j \;\; . \tag{6.8}$$

Let $\mathcal{J} = \{c_1, \ldots c_l\}$ a partition of $\mathcal{I}$ such that each interval $[(i-1)2^v+1, i\times 2^v], 1 \le i \le k$ is included in the same class of $\mathcal{J}$: $\forall i, 1 \le i \le k, \exists j : [(i-1)2^v+1, i \times 2^v] \subseteq c_j$. Let

$$x^\sharp = \sum_{i=1}^{k} a_i \;\; \textit{with} \;\; a_i = x_0 + \sum_{c_j \in \mathcal{J}} \beta_{c_j} \varepsilon_{\nu(c_j)} \quad \textit{where} \quad \beta_{c_j} = \sum_{i \in c_j} |x_i| \;\; , \tag{6.9}$$

Then

$$\hat{x} \times c \;\; \in \;\; \gamma_{\mathcal{J},\mathcal{I}} \left(x^\sharp \times c - (k-1)(x_0 \times c) \right) \;\; . \tag{6.10}$$

However, this equality is valid only when $k \times 2^v = 2^u$. If this condition is not satisfied, the result does not hold.

6.2 The NNaff Tool

In this section, we will introduce our tool, NNaff, for verifying neural networks with affine forms. We will give an overview of the architecture of the tool, its inputs and outputs, and how it can be used via several examples as well as detailed experimental results.

6.2.1 Architecture of The Tool

In this section, we will provide an overview of the architecture of our tool. NNaff, is implemented in Python. The following is an illustration of the tool

Parameters

The parameters used in our tool are as follows:

- **Model:** refers to a file with the extension `.h5`, which represents the parameters of our trained neural network (weights, biases, activation functions, etc.).
- **Input:** a file with the extension `.Pkl` containing the pixel values of our inputs, along with their size ($n \times n \times 1$). By default, we consider grayscale images and normalize the pixel values between 0 and 1.
- **Other parameters:** represent the parameters that we will use in usual affine, compressed affine, and block-wise noising approaches. The key parameters are
    1. α: the intensity of noise, represents the noise level associated with each pixel,
    2. β: the compression factor, represents the number of noise symbols that we want to merge together,
    3. γ: the number of blocks, represents the number of blocks into which we want to divide our input.

Modules

The tool consists of several modules, each implementing different aspects of the usual affine forms, compressed affine, bloc wise noising and mixed approaches:

- **Affine:** implements the forward and backward propagation of the model using standard affine forms, as described in Chapter 3.
- **Compressed Affine:** implements forward and backward propagation of the model using compressed affine forms, as defined in Chapter 4.

- **Bloc wise noising:** implements forward and backward propagation of the model using the block-wise noising approach defined in Chapter 5.
- **Mixed approach:** implements forward and backward propagation of the model using a mixed approach, combining compressed and block-wise noising methods as defined in Section 6.1.

Libraries

Our tool relies on the use of three main libraries:

- **Affapy:** This is a Python library designed for multiprecision Affine Arithmetic. Affapy enables operations with affine forms and was developed by Thibaut Hilaire [50]. It was previously introduced in Chapter 2.
- **Keras:** It is an open-source neural network API written in Python. It allows deep neural network experiments to be done quickly. Keras has a user-friendly interface for building, training, and evaluating different types of neural networks, and it can run on top of popular deep learning frameworks like TensorFlow. It supports both convolutional networks and recurrent networks.
- **Pickle:** is a Python library used to facilitate the conversion of Python objects into byte streams and vice versa. It enables the efficient storage and retrieval of complex data structures. As one of the many libraries included in the Python Standard Library, Pickle may serialize objects such as lists, dictionaries and even user-defined classes into a byte stream that can be stored within a file. Later this stream of bytes can be read from a file and converted back to initial objects containing structure and data details. The library has an easy-to-understand API that includes functions like **pickle.dump()** for serialization and **pickle.load()** for deserialization that help it be easily implemented by users. To see how Pickle is utilized for storing and loading image data in practice, refer to Appendix A.2, where we provide an example demonstrating these functionalities.

6.2.2 Using the Tool

In this section, we are going to show how our tool, NNaff, is used. The general command of execution is as follows:

```
./nnaff --model_path './file.h5' --data_path './file.pkl' --noise α
```

Here, *model_path* represents the neural network we wish to analyze, *data_path* represents the image we intend to use, and *noise* represents the noise value associated with each pixel.

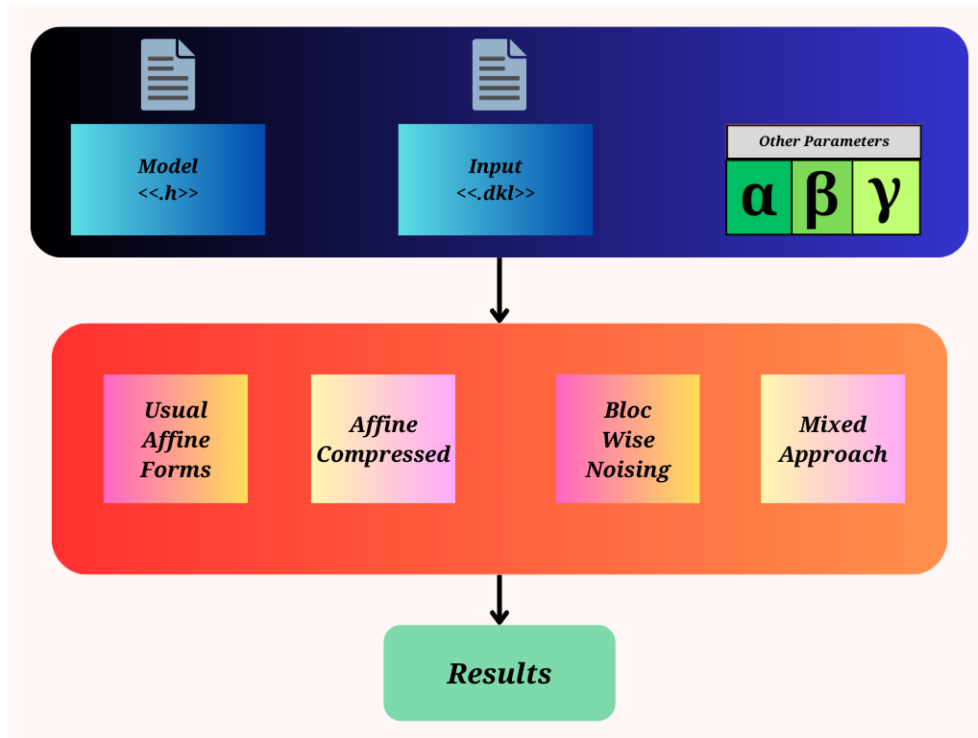


Figure 6.2: Functional architecture of the NNaff tool.

As an example, if we want to add 0.04 noise to each pixel, we simply need to set $\alpha = 0.04$. By running the command above, one obtains an analysis of the model using standard affine forms.

If we aim at using our compressed affine form approach described in Chapter 4, we only need to add the parameter *–factor_comp* to the previous command:

```
./nnaff --model\_path './file.h5' --data\_path './file.pkl'\\ --
    noise α --factor\_comp β
```

The *factor_compressed* parameter represents the compression factor applied to the noise symbols in the compressed affine form approach. It indicates how many noise symbols are combined together to reduce the overall computational complexity and memory while preserving the precision of the analysis of the model. In addition to this, users can vary their analysis accuracy or computational efficiency by modifying the *factor_comp*.

To apply the block-wise noising approach, you should just include the parameter *–block*:

```
./nnaff --model\_path './file.h5' --data\_path './file.pkl' \\--
    noiseα --bloc γ
```

Here, *block* means the number of blocks into which we divide our input (e.g., 4, 8, etc.).

If we want to mix both methods, i.e., compressed affine and block-wise noising, we should mention simultaneously these two parameters *bloc* and *factor_comp* in the command

line:

```
./nnaff --model\_path './file.h5' --data\_path './file.pkl' \\--
  noise α --factor\_comp β --bloc γ
```

It should be noted that our tool supports the dense, flatten, conv2d and maxpool layers. Regarding activation functions, it only considers the ReLU function, which we define in Chapter 4, representing an affine form approximation of the original ReLU function.

6.2.3 Experimental Results

In this section, we aim at evaluating the performance of NNaff, on different types of trained neural networks. The experimental protocol and results are provided in the following paragraphs.

Protocol

We describe here after the architectures of various trained neural networks (models) employed in our study. The architectural specifications are summarized in the table 6.1.

Dataset	Models	Types	layers	Params
MNIST	**Conv1**	convolutional	11	$136,906$
	Fc1	fully connected	8	593322
FASHION_MNIST	**Conv2**	convolutional	7	$431,242$
	Fc2	fully connected	6	$575,050$
CIFAR 10	**LeNet**	convolutional	8	$44,426$
	AlexNet	convolutional	12	$2,910,730$
	Vgg16	convolutional	18	$30,613,572$

Table 6.1: Neural Network Architectures used for Evaluating the Performance of the NNaff Tool.

The first column of the table 6.1 gives the dataset, the second column gives the model

name, the third column shows the type of the model, the fourth column gives the number of layers, and the last column gives the number of parameters.

Note that we are working with images of size $(28 \times 28 \times 1)$ extracted from the MNIST, Fashion MNIST dataset [33] and images of size $(32 \times 32 \times 1)$ extracted from the CIFAR dataset [1]. For efficiency purposes, we normalize the images by scaling pixel values to the range $[0, 1]$.

For the compressed affine form approach, we have decided to merge the noise symbols by 2 (*factor_comp* = 2) and for the block-wise noising approach, we divide the images by 4 (*bloc* = 4). In the case of the mixed approach, we opt for a combination where (*factor_comp* = 2) and (*bloc* = 4) for efficiency reasons based on the types of networks we employ.

Execution time

In this section, we present the results obtained by NNaff on several models. The results are provided in the table 6.2:

Mnist				
Models	**Affine**	**Compressed**	**Bloc wise**	**Mixed**
Conv1	52 min	34 min	49 min	28 min
Fc1	1 h 50 min	38 min	53 min	24 min
Fashion Mnist				
Models	**Affine**	**Compressed**	**Bloc wise**	**Mixed**
Conv2	47 min	22 min	35 min	20 min
Fc2	1 h45 min	35 min	49min	22 min
Cifar				
Models	**Affine**	**Compressed**	**Bloc wise**	**Mixed**
LeNet	32 min	10 min	16 min	9 min
AlexNet	×	34 h	2 days 12 h	32 h
VGG16	×	×	20 days	9 days

Table 6.2: Execution Time with Different Models.

In our analysis of the results, we observe a significant improvement in execution times for different types of neural networks. Particularly, in the case of fully connected neural networks, the execution time is divided by 2. For example, for the FC2 model with Fashion MNIST, the execution times are 1 h, 45 min, 35 min, 49 h and 22 min for the usual affine, affine compressed, bloc-wise noising and mixed approaches. This is particularly remarkable since the operations in a fully connected network are primarily limited to simple multiplications and additions. Conversely, convolutional networks must perform convolutions that involve sliding filters over the input image and conducting multiplications and additions for each filter position, which can be more computationally expensive. Hence, fully connected networks tend to have faster execution times.

Furthermore, for convolutional neural networks (CNN), we also notice an improvement in terms of execution time, allowing us to analyze large neural networks such as VGG16 and ResNet, which could not even be analyzed using the classical affine form approach on the same machine. For example, for the AlexNet Model, the execution times are 34 h, 2 days, and 32 h for affine compressed, bloc-wise noising and mixed approaches. Note that due to the long execution time, we are unable to achieve an analysis using the standard affine approach.

By doing more detailed analysis and comparing the four available techniques used in this study, it becomes clear that among all four of them, the mixed approach is much faster, particularly when working with large neural networks like VGG16, where even the compressed affine approach fails to analyze it at all. We can conclude that the mixed approach offers a balance between accuracy and execution time, which makes it a viable choice in many situations.

6.3 Summary

In this chapter, we have introduced our tool NNaff for verifying neural networks. We have presented its architecture, input parameters, and outputs. NNaff has demonstrated its effectivness through exprimental results where still balence between the precison in one hand and the execution time in other hand. This balance is crucial for practical applications, where both accuracy and speed are often essential.

Chapter 7

Conclusion

7.1 Summary of key Results 89
7.2 Perspectives 91
7.2.1 Enhanced Compressed Affine Forms 91
7.2.2 Scalability: Extending to More Complex Networks and Activation Functions 92
7.2.3 Parallelization and GPU Utilization 92

7.1 Summary of key Results

As stated in Chapter 1, the main objective of this thesis was to improve to the scalability of static analysis with affine forms of deep neural networks. More precisely, the aim was to develop new way to represent usual affine forms in neural network verification. Broadly speaking, we sought to design more computationally efficient affine forms that would ultimately verify the robustness of DNNs faster than the usual affine forms while maintaining a similar level of precision.

We started this dissertation by introducing the foundational concepts related to the development of NNaff, which are summarized in Chapters 2 and 3. This included an overview of neural network architectures and a review of existing literature regarding DNN verification. The review highlighted the advantages and limitations of various methods, motivating our research. Building on this foundation, we focused on a particular method from the literature based on abstract interpretation, specifically affine forms.

In Chapters 4 and 5, we proposed two novel approaches for verifying DNNs, called *Compressed Affine Forms* and *Block-Wise Noising*. Both approaches are based on the same principles as traditional affine forms but incorporate mechanisms to reduce the number of noise symbols, addressing a significant challenge in verifying large DNNs with affine forms.

The main idea behind *compressed affine forms* is to merge several noise symbols into a single noise symbol in a way that ensures the soundness of the computations. Compressed affine forms, being an abstraction of affine forms, are less precise. However, since they use fewer noise symbols, computations with them are significantly faster and less memory-consuming than with usual affine forms. Experiments evaluating neural networks have shown that our compressed affine approach is faster and more memory-efficient than traditional affine forms. In other words, compressed affine forms represent a tradeoff between precision on one hand and execution time and memory consumption on the other.

Our second approach *block-wise noising* consists of adding non-zero noise symbols only within a defined region of an image. 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. Additionally, Block-wise noising parallel has also been implemented, and the findings demonstrate that it can speed up computation by 3.1 to 4 times.

In chapter 6 we introduced NNaff, a tool designed for analyzing neural networks. NNaff takes as input neural networks and utilizes one of the developed approaches: Compressed Affine Forms, Block-Wise Noising, or a combined method that integrates both techniques. It returns the necessary execution time required for analyzing the DNN using the selected method. We conclude this chapter by evaluating the performance of NNaff on several examples and presenting a comprehensive comparison of our approaches: usual affine, compressed, block-wise noising, and the mixed approach.

A key point of our results is that, thanks to NNaff, we are now capable of analyzing large-scale neural networks such as AlexNet [64], VGG16 [86], and ResNet [48], which possess millions of parameters. Traditional methods based on usual affine forms cannot handle such networks due to the exponential increase in the number of parameters. However, our new approaches enable efficient analysis of these large networks, overcoming the brawbacks of traditional methods based on affine forms.

To illustrate this achievement, we included Figure 7.1 that compares the networks that could be analyzed using standard affine forms with those that NNaff can handle. The Figure 7.1 clearly shows that NNaff significantly surpasses standard affine forms in terms of the scale of networks it can analyze.

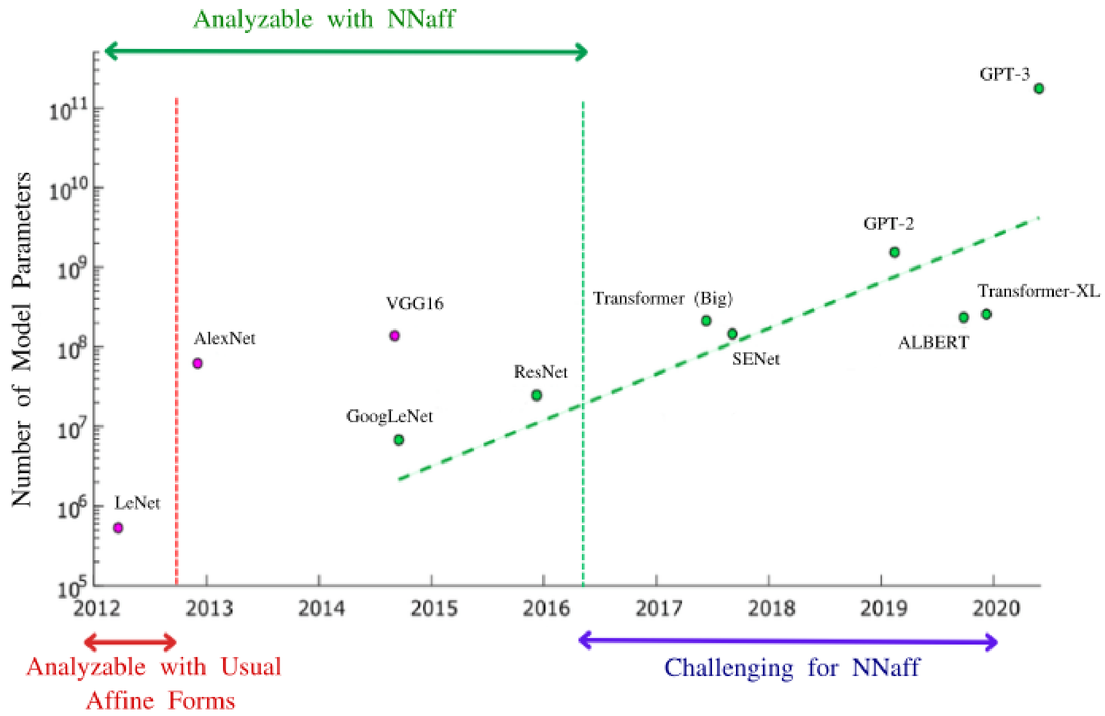


Figure 7.1: Neural Network Models Analyzed by Usual Affine Forms and NNaff.

7.2 Perspectives

The work carried out during this thesis opens the way to many new perspectives, some of which are detailed below.

7.2.1 Enhanced Compressed Affine Forms

Our current method for compressed affine forms involves merging k consecutive noise symbols from the same line into a single new symbol. As a perspective, it would be beneficial to explore other grouping strategies during analysis.

- We could start by merging the smallest noise symbols together. In practice, this method involves identifying the noise symbols that have coefficients below a certain threshold and then merging all these noise symbols into one new symbol. In this way, we can minimize the impact on precision and reduce the risk of over-approximation. However, we should be careful in choosing the threshold because if we group too many noise symbols, we may lose precision, whereas if we group too few, we may not achieve any performance gain. Therefore, we should find a way to determine the optimal threshold. One idea is to start with a very large threshold value. If, because of the over-approximation, some false alarms arise, we can then decrease the threshold value. Obviously, this process can be applied repeatedly until we find the optimal

threshold. We can use techniques such as a bissection algorithm, similar to what is done in interval bissection, to achieve this.

By experimenting with different strategies for noise symbol merging. We hope to enhance the efficiency and precision of our compressed affine forms.

7.2.2 Scalability: Extending to More Complex Networks and Activation Functions

Regarding scalability, our current approach focuses on ReLU activation functions and primarily utilizes convolutional and fully connected layers. To address the scalability challenge, we propose at least two directions for future work.

The first direction is involves expanding the types of activation functions used in our analysis. By incorporating other types of activation functions such as $sigmoid$, $tanh$ and $softmax$ and exploring abstract transformations for these functions, This would allow us to broaden our understanding of the applicability of our approaches. In the same idea, we plan to extend our analysis beyond convolutional and fully connected layers to include other types of layers, such as normalization and residual layers. This extension will improve the scalability of our tool, enabling it to handle more complex models, including those similar to GPT-4 [2] and PaLM [23].

The second direction focuses on expanding the types of neural networks analyzed. Our current work focuses mainly on convolutional networks, but it would be interesting to explore other types of neural networks, such as recurrent neural networks (RNN) [75, 98]. As they are increasingly used in various task, such as natural language processing and learning word embeddings, it would be relevant to study how our methods behave in such models.

By addressing these aspects, we aim to improve NNaff ability to analyze a very wide variety of neural networks.

7.2.3 Parallelization and GPU Utilization

Nowdays, computational tools for rigorously validating the performance of large-scale neural networks models have advanced significantly. The most effective solvers utilize highly specialized, GPU accelerators. These tools are essential for the successful deployment of DNN in safety-critical systems.

In our current work, block-wise noising employs CPU parallelization by dividing the computation of each block across different processors. To further enhance performance, we seek to customize algorithm with explicit memory management to efficiently parallelize our

approaches (Block wise noisng, compressed and mixed approaches) on GPUs. This could yield significant improvements in execution times since we know that using GPUs can provide more computation power thatn general purpose CPUs.

Additionally, we plan to explore the simultaneous parallelization methods, model parallelism [58, 101] and data parallelism [70] We can rely on existing GPU libraries such as cuDNN [21] developed by NVIDIA, which offers highly optimized implementations of neural networks operations. This dual parallelization approach could further reduce computational overhead and enhance efficiency of our tool.

By the end of this thesis, we are confident that our tool NNaff is a promising step towards addressing the challenge of verifying neural networks with respect to both scalability and precision. It opens up many research perspectives for its further improvement and applications.

Part III

Résumé étendu à l'intention du lecteur francophone

Chapter 8

Analyse Statique de réseaux neurones

8.1 Introduction . . . 97
8.2 Aperçu des réseaux de neurones . . . 100
8.2.1 L'architecture d'un réseau de neurones . . . 100
8.2.2 Les types des réseaux neurones . . . 101
8.2.3 Méthodes pour vérifier les réseaux de neurones . . . 102
8.3 Formes affines . . . 103
8.3.1 Notation . . . 103
8.3.2 Les opérations élémentaires . . . 103
8.4 Forme affine compressée . . . 105
8.4.1 Principe . . . 105
8.4.2 Résultats expérimentaux . . . 107
8.5 Bruit par bloc . . . 109
8.5.1 Principe de bruit par bloc . . . 110
8.5.2 Fonction d'activation . . . 112
8.5.3 Résultats expérimentaux . . . 112
8.6 L'outil NNAFF . . . 114
8.6.1 L'utilisation de l'outil . . . 115
8.6.2 Résultats expérimentaux . . . 116
8.7 Conclusion . . . 118
A.1 Neural Networks Evaluated . . . 134
A.2 Loading and saving Images with Pickle library . . . 138

8.1 Introduction

Les réseaux de neurones [66, 6] ont démontré un succès considérable dans la résolution de problèmes complexes dans divers domaines, y compris la vision par ordinateur [122],

la classification d'images [61] et le traitement du langage naturel [81]. De nos jours, les réseaux de neurones sont de plus en plus utilisés dans des systèmes critiques [19]. Par exemple, dans les voitures autonomes [34], les réseaux de neurones sont employés pour des tâches telles que la détection d'objets, le maintien de la voie et les processus de prise de décision afin de garantir une navigation sûre et efficace. De même, dans les systèmes de la santé [126], les réseaux de neurones sont appliqués pour analyser les données médicales, prédire les résultats des patients et aider au diagnostic précoce, améliorant ainsi les soins et les traitements des patients. Cependant, malgré leur application et leur succès répandus, le déploiement de réseaux de neurones dans ces systèmes critiques introduit des défis significatifs [84, 127]. Les réseaux de neurones sont fréquemment construits à partir de jeux de données d'entraînement biaisés ou incomplets, ce qui peut conduire à des résultats peu fiables ou dangereux. Par exemple, si un réseau de neurones dans une voiture autonome interprète mal un panneau stop en raison de données d'entraînement biaisées, cela pourrait entraîner un échec à s'arrêter au stop, pouvant causer un accident grave. Il est donc crucial de garantir la sûreté de ces réseaux de neurones avant leur déploiement. La sûreté, dans ce contexte, se réfère à une catégorie générale de propriétés de correction stipulant qu'un réseau de neurones ne doit pas atteindre un état indésirable. Elle désigne la capacité d'un réseau de neurones à fonctionner sans causer de risques inacceptables de préjudice ou de dommages, même dans des situations inattendues ou face à des données incomplètes ou biaisées. Par exemple, dans le cas des réseaux de neurones conçus pour un système d'évitement de collision aérienne [62], l'une des propriétés de sûreté que nous voulons prouver est que si un un autre avion approche par la gauche, le réseau de neurones doit ordonner à l'avion de tourner à droite.

Le défi clé aujourd'hui est la conception de vérificateurs capables de gérer des réseaux de neurones de grande taille tout en maintenant une précision suffisante pour garantir la sûreté de ces réseaux. La taille des réseaux de neurones a explosé au fil du temps, rendant cette tâche de plus en plus délicate. Plusieurs méthodes ont été développées pour traiter ce problème et prouver automatiquement que les réseaux de neurones fonctionnent correctement dans tous les cas possibles [123, 41]. Ces méthodes vérifient la sûreté du réseau avant le déploiement. Il existe deux principales catégories de méthodes : les méthodes formelles complètes [62, 63, 62, 63, 35, 111, 91] et les méthodes formelles incomplètes [38, 104, 105, 16, 119]. Les méthodes formelles complètes ciblent les réseaux de neurones de petite taille avec une grande précision. Ces techniques sont adéquates et complètes; elles nous indiquent exactement si une propriété donnée est vérifiée ou non pour les réseaux de neurones. En revanche, les méthodes formelles incomplètes peuvent vérifier des réseaux de grande taille en quelques minutes, en calculant une sur-approximation de la sortie du réseau correspondant à l'entrée, mais cette sur-approximation conduit à des faux positifs, où la méthode identifie incorrectement un cas sûr comme étant dangereux. De plus, il existe des méthodes [103, 116] qui intègrent à la fois des méthodes complètes et incomplètes visant à être plus évolutives que les méthodes complètes tout en améliorant la précision des méthodes incomplètes.

Dans cette thèse, nous considérerons les méthodes d'interprétation abstraite basées sur les formes affines. La décision d'utiliser ces formes n'est pas venue au hasard, mais plutôt motivée par le fait qu'elles préservent les relations linéaires entre les variables, et que les réseaux neurones fonctionnent principalement à travers des calculs affines comme les couches totalement connectées ou convolutionnelles. Cependant, un inconvénient important de l'utilisation des formes affines dans le contexte des réseaux neurones est qu'elles ne sont pas exactes pour les fonctions non linéaires, telles que les fonctions d'activation, et qu'elles consomment du temps et de la mémoire, alors que les réseaux de neurones deviennent de plus en plus volumineux. Les tâches complexes nécessitent des modèles de réseaux neurones profonds avec un grand nombre de paramètres qui doivent être entraînés. Par exemple, Resnet-50 nécessite $25,5$ millions de paramètres, AlexNet 61 millions de paramètres, VGG-16 138 paramètres et GPT-3 175 milliards de paramètres pour l'entraînement. La vérification de ces réseaux de neurones à l'aide de l'interprétation abstraite, présente un défi de compromis entre le passage à l'échelle et précision. Par conséquent, l'utilisation de formes affines dans la vérification des réseaux de neurones devient un processus intensif en terme de calcul qui peut prendre plusieurs jours, voire plusieurs mois, pour être achevé. Cela est dû à la nécessité de mettre à jour les paramètres du modèle en ajoutant des symboles de bruit pour chaque pixel de l'entrée. En effet, une forme affine est construite à partir d'un centre correspondant à la valeur du pixel et de symboles de bruit, qui représentent l'intensité du bruit ajouté à chaque pixel. Par exemple, VGG-16 prend une entrée de taille (224×224), nous ajoutons donc (224×224) symboles de bruit dans les premières couches d'entrée du modèle (à mesure que les réseaux de neurones s'étendent et que les ensembles de données disponibles augmentent, le nombre de symboles de bruit augmente également, ce qui entraîne une augmentation de la quantité de calcul).

Cette thèse vise à élaborer des méthodes qui utilisent le même principe que les formes affines afin de trouver un équilibre entre la précision d'un côté et le temps d'exécution de l'autre côté, en comparaison avec les formes affines standard. Notre objectif est de maintenir une précision maximale tout en réduisant considérablement le temps de calcul. En utilisant ces méthodes, nous espérons rendre la vérification des réseaux neurones plus efficace et pratique pour des applications à grande échelle tout en maintenant la robustesse et la fiabilité des modèles analysés.

8.2 Aperçu des réseaux de neurones

Les réseaux de neurones [65] sont au cœur des algorithmes d'apprentissage profond. Leur nom et leur structure sont inspirés du cerveau humain, imitant la façon dont les neurones biologiques communiquant les uns avec les autres. ils s'appuient sur le principe d'entraînement pour apprendre et améliorer leur précision au fil du temps. Nous présentons dans cette section, l'architecture général de ces réseaux de neurones et leurs fonctionnements ainsi que les différents types existants.

8.2.1 L'architecture d'un réseau de neurones

Les réseaux de neurones artificiels (ANN) [89] sont constitués d'une couche d'entrée, une ou plusieurs couches cachées et une couche de sortie. Chaque couche est constituée de nœuds, ou neurones artificiels, une couche se connecte à une autre et chaque noeud possède des poids et un seuil associés. La Figure 8.1 représente l'architecture d'un réseau de neurone. Le vecteur $\mathbf{w} = (w_1, \ldots, w_4)$ représente les poids, b désigne le biais et σ est une fonction non linéaire, monotone et différentiable appelée fonction d'activation.

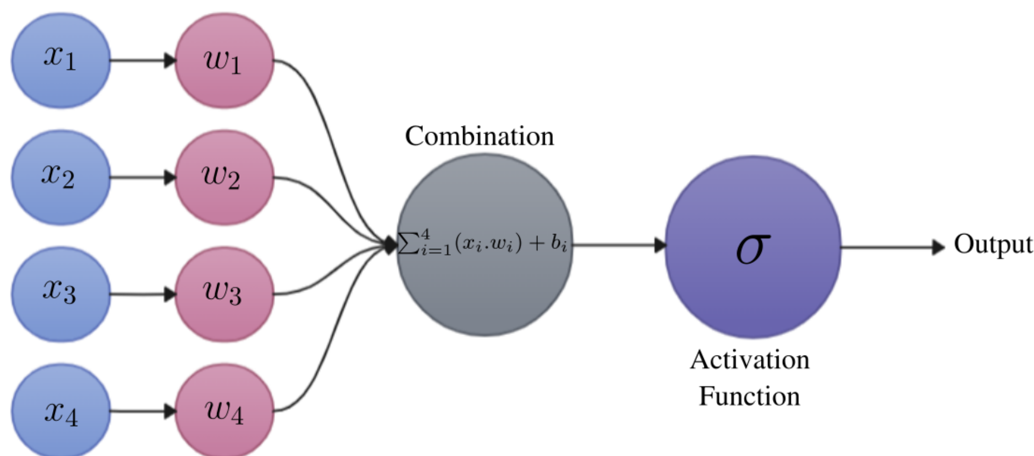


Figure 8.1: l'architecture d'un réseau neuronal.

Les fonctions d'activation sont un élément crucial des réseaux de neurones. Elles introduisent de la non-linéarité dans le modèle, ce qui facilite l'apprentissage des motifs complexes. Sans les fonctions d'activation, un réseau de neurone ne serait qu'une série de transformations linéaires, ce qui limiterait sévèrement ses capacités d'apprentissage. Il existe plusieurs types de fonctions d'activation, et chaque fonction possède ses propres caractéristiques. Par exemple on a la fonction sigmoïde [51], qui est généralement utilisée dans les problèmes de classification binaire, permet de mapper les sorties des neurones à des probabilités comprises entre 0 et 1. Les fonctions tangentes hyperboliques (tanh)[68] sont comparables à la fonction sigmoïde, mais elles transforment les sorties dans un intervalle de -1 à 1. En enfin, la fonction ReLU (rectified linear unit)[39] qui est la fonction d'activation

la plus utilisée dans les réseaux de neurones profonds. Elle renvoie directement l'entrée si elle est positive, et 0 si elle est négative.

8.2.2 Les types des réseaux neurones

Au fil du temps, divers types de réseaux de neurones ont été développés pour résoudre une gamme variée de problèmes, allant de simples calculs à des tâches complexes comme la reconnaissance vocale et la classification. Dans cette section, nous présenterons brièvement les modèles les plus couramment utilisés.

1. **Réseaux Entièrement Connecté (FeedForward) [9]**: Un réseau de neurones FeedForward est un réseau de neurones artificiel dans lequel les connexions entre les nœuds ne forment pas un cycle. Il est la forme la plus simple de réseau de neurones car les informations ne sont traitées que dans un seul sens. Bien que les données puissent traverser plusieurs nœuds cachés, elles se déplacent toujours dans une direction et jamais en arrière.

2. **Réseaux Convolutionnels**[71, 5]: Les réseaux de neurones convolutifs se distinguent des autres réseaux de neurones par leurs performances supérieures avec les entrées de signaux d'image, de parole ou audio. ils sont composés de quatre types de couches:

   - La couche convolutif [120] est la première couche d'un réseau convolutionnelle. Elle est basé sur le principe de convolution qui est une opération linéaire consiste à faire glisser un ensemble de petites matrices, appelées filtres ou noyaux (qui représentent les poids de la couche) sur les données d'entrée en calculant le produit scalaire entre les filtres et l'entrée à chaque position et en produisant des cartes caractéristiques (feature map) en sortie.
   - La couche de pooling [120] est placée souvent entre deux couches convolutionnelles. Elle reçoit en entrée plusieurs entrées et applique à chacune d'entre elles l'opération de pooling qui consiste à réduire la taille des images , tout en préservant les caractéristiques importantes. Il existe deux grands types de pooling: max pooling et average pooling. En max pooling, l'opérateur de pooling attribue à une sous-région d'entrée sa valeur maximale, tandis qu'en average pooling, il attribue à une sous-région d'entrée sa valeur moyenne.
   - Couche de correction Relu (Rectified linear units) [39], désigne la fonction non linéaire définie par :

     $$ReLU(x) = Max(0, x) \tag{8.1}$$

     Remplace toutes les valeurs négatives reçues en entrées par des zéros, elle joue le rôle de fonction d'activation.
   - Les couches entièrement connectées [9] constituent toujours la dernière couche de réseau de neurones. Elle reçoit un vecteur en entrée et produit un nou-

veau vecteur en sortie, elle applique une combinaison linéaire puis une fonction d'activation aux valeurs reçus en entrées.

8.2.3 Méthodes pour vérifier les réseaux de neurones

Comme on l'a vu dans l'introduction, les réseaux de neurones sont omniprésents dans nos vies quotidiennes. Ils sont utilisés dans divers domaines, notamment les domaines critiques tels que le médical et les voitures autonomes. En ce contexte critique, il est essentiel de garantir la sûreté et la fiabilité de ces réseaux pour éviter les défaillances potentielles. C'est ici que les méthodes de vérification formelle entrent en jeu. Ces approches s'appuient sur la logique et les mathématiques pour déterminer si un réseau de neurone donné répond à ces exigences. On peut distinguer deux types de techniques: les méthodes complètes et les méthodes incomplètes, en fonction des réseaux de neurones pris en charge et de la technique sous-jacente utilisée.

1. **Méthodes complètes** : ce sont des méthodes basées sur les mathématiques pour garantir l'exactitude et la sûreté des réseaux de neurones. On peut trouver des méthodes basées sur les solveurs SMT (Planet [37], Reluplex [62], Marabou [63]), MILP (Bab [18], Sherlock [35], MipVerify [111]) et l'optimisation globale (DeepGo [91]). Néanmoins, ces méthodes présentent certaines limites. En général, elles ne conviennent pas aux réseaux neurones à grande échelle en raison de la complexité de l'espace de recherche et de la consommation de temps de calcul. En outre, elles peuvent être limitées à certaines architectures de réseaux de neurones, ce qui réduit leur applicabilité dans certains cas d'utilisation.

2. **Méthodes incomplètes** : lorsque la précision et la robustesse sont requises, les méthodes de vérification formelle incomplètes sont nécessaires pour vérifier la fiabilité et la sécurité des réseaux neurones complexes. La principale caractéristique de ces méthodes est qu'elles peuvent traiter efficacement des réseaux à grande échelle. Par exemple, elles peuvent faire fonctionner des réseaux de neurones avec plus de 1000 de couches cachées en quelques minutes [104]. Ces procédures comprennent l'interprétation abstraite (Ai^2 [38], Libra [112]), la dualité [36, 119] et l'approximation linéaire (Crown [124], CNN-Cert [14], FastLin/FastLip [118]), qui se sont avérées efficaces en général, ce qui nous donne confiance dans leur capacité à garantir un comportement correct des réseaux. Cependant, elles peuvent parfois donner de fausses alertes, également connues sous le nom de faux positifs, lorsqu'une propriété est considérée par erreur comme vérifiée alors qu'elle n'est pas vraie, malgré cet inconvénient, leur contribution à la fiabilité de l'IA ne doit pas être sous-estimée.

8.3 Formes affines

Notre technique d'analyse statique des réseaux de neurones est basée sur le principe de l'arithmétique affine. Dans cette section, nous allons donner une définition simple de ces formes et comment les calculs sont effectués avec.

8.3.1 Notation

Les formes affines sont des extensions de l'arithmétique de l'intervalle, introduites la première fois en 1994 par Kauffman et Stolfo [67]. Leur principe consiste à conserver les relations entre les variables pendant les calculs. En d'autres termes, toutes les variables sont représentées sous forme d'une combinaison linéaire d'intervalle $[-1, 1]$ et d'une constante. Une forme affine est décrite comme ceci :

$$\hat{x} = x_0 + x_1\varepsilon_1 + x_2\varepsilon_2 + ... + x_n\varepsilon_n = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i \quad . \tag{8.2}$$

Ou x_0 représente le centre de la forme représenté par un nombre réel, ε_i représentant les symboles de bruits dont la valeur est inconnue donnée par l'intervalle $[-1, 1]$. Dans le cas des réseaux de neurones, le centre de la forme affine x_0 est la valeur du pixel, et les symboles de bruit représentent les perturbations associées à chaque pixel.

On note une relation étroite entre l'arithmétique d'intervalle et les formes affines [25]. La conversion de l'une à l'autre est donnée comme ceci.

- Forme affine vers intervalle : Soit $\hat{x} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i$ une forme affine. L'intervalle associé à $\hat{x}$ est défini par

$$\gamma_I(\hat{x}) = [x_0, x_0] + \left(\sum_{i=1}^{n} |x_i|\right) \times [-1, 1] \quad . \tag{8.3}$$

- Intervalle vers forme affine : Soit $X = [\underline{x}, \overline{x}]$ un intervalle. La forme affine associée à X est définie par

$$\alpha_I(X) = \frac{\underline{x} + \overline{x}}{2} + \frac{\underline{x} - \overline{x}}{2}\varepsilon_k \quad . \tag{8.4}$$

  ou ε_k représente un nouveau symbole de bruit.

8.3.2 Les opérations élémentaires

Dans cette section, nous allons introduire les opérations élémentaires dans le cadre des formes affines. soit $\hat{x}$, $\hat{y}$ deux formes affines

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i \quad \text{et} \quad \hat{y} = y_0 + \sum_{i=1}^{n} y_i\varepsilon_i \quad . \tag{8.5}$$

Soit $(+, -, \times)$ les opérations d'addition, de soustraction et de multiplication par constante et soit $a \in \mathbb{R}$ une constante. Les opérations élémentaires entre $\hat{x}$ et $\hat{y}$ sont données comme suit :

$$\hat{x} \pm \hat{y} = (x_0 \pm y_0) + \sum_{i=1}^{n} (x_i \pm y_i)\varepsilon_i \quad , \tag{8.6}$$

$$\hat{x} \pm a = (x_0 \pm a) + \sum_{i=1}^{n} (x_i)\varepsilon_i \quad , \tag{8.7}$$

$$\hat{x} \times a = (ax_0) + \sum_{i=1}^{n} (ax_i)\varepsilon_i \quad . \tag{8.8}$$

La multiplication de deux formes affines est une opération non linéaire, son résultat doit être approximé en utilisant une forme affine. Par conséquent, pour représenter l'erreur découlant de cette approximation affine, un terme supplémentaire doit être incorporé. La multiplication de deux formes affines est définie comme:

$$\begin{aligned} \hat{x} \times \hat{y} &= (x_0 + \sum_{i=1}^{n} x_i\varepsilon_i) \times (y_0 + \sum_{i=1}^{n} y_i\varepsilon_i) \\ &= x_0 y_0 + \sum_{i=1}^{n} (x_0 y_i + y_0 x_i) + \left(\sum_{i=1}^{n} |x_i| \times \sum_{i=1}^{n} |y_i| \right) \varepsilon_{n+1} \quad . \end{aligned} \tag{8.9}$$

où, ε_{n+1} représente un nouveau symbole de bruit qui capture la partie non linéaire qui ne peut pas être représentée par une simple forme affine.

La prédiction de la taille de ces formes affines est un grand challenge à cause de leur principe qui consiste à ajouter des symboles de bruits pour représenter les opérations non linéaires, comme on l'a vu dans le cas de la multiplication, pour résoudre ce problème. Messine [78] a proposé 2 extensions AF1 et AF2. Ces deux formes limitent donc le nombre de symboles ajoutés à ces formes pour représenter la non linéarité des opérations tout en préservant les erreurs générées par l'approximation de ces termes non linéaires.

- AF1[78, 77] basé sur le même principe que les formes affines standard, sauf que cette fois, tous les symboles correspondant à l'approximation des termes non linéaires sont combinés dans un seul symbole. Le format d'une forme affine AF1 est donnée par

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i + x_{n+1}\varepsilon_{n+1} \quad . \tag{8.10}$$

- AF2 [78, 77] basé sur le même principe que AF1, le nombre de variables est donc constant. Par contre, cette fois, l'erreur est stockée dans 3 symboles différents: un symbole pour l'erreur positive, l'erreur négative et l'erreur à signe indéterminé. Le format d'une forme affine AF2 est donnée par

$$\hat{x} = x_0 + \sum_{i=1}^{n} x_i\varepsilon_i + x_{n+1}\varepsilon_{n+1} + x_{n+2}\varepsilon_{n+2} + x_{n+3}\varepsilon_{n+3} \quad . \tag{8.11}$$

8.4 Forme affine compressée

Dans cette section, nous allons présenter notre première contribution dans le domaine de la vérification des réseaux de neurones à l'aide d'une interprétation abstraite. Nous proposons une nouvelle technique appelée *forme affine compressée.* Les formes affines compressées sont une abstraction (au sens de l'interprétation abstraite) des formes affines classiques. Elles utilisent moins de symboles de bruit et, par conséquent, elles nécessitent moins de temps et de mémoire, tout en maintenant une perte de précision acceptable.

8.4.1 Principe

Nous introduisons ici de manière informelle les formes affines compressées, et nous montrons intuitivement comment calculer avec elles. Soit $\hat{x} \in \mathcal{F}(\mathcal{I})$ et $\hat{y} \in \mathcal{F}(\mathcal{I})$ deux formes affines telles que

$$\hat{x} = 4 + 0.1\varepsilon_1 + 0.3\varepsilon_2 + 0.2\varepsilon_3 \quad \text{et} \quad \hat{y} = 1 + 0.2\varepsilon_1 + 0.2\varepsilon_2 + 0.3\varepsilon_4 \ . \tag{8.12}$$

Pour des raisons d'efficacité, au lieu de $\hat{x}$ et $\hat{y}$, nous voulons utiliser deux formes affines compressées $x^\sharp$ et $y^\sharp$ où ε_1 et ε_2 sont fusionnés en un seul symbole de bruit. Notre objectif est de faire cela de manière sûre, c'est-à-dire que nous voulons que ce que nous calculons avec les formes affines compressées sur-approxime ce que nous calculons avec les formes affines classique. En d'autres termes, soit $\gamma_I : A \to I$ l'intervalle de concrétisation défini dans l'équation (3.13), pour $* \in \{+, -, \times, \div, \ldots\}$, nous voulons

$$\gamma_I(\hat{x} * \hat{y}) \subseteq \gamma_I(x^\sharp * y^\sharp) \tag{8.13}$$

où $\gamma_I(\hat{x})$ est l'intervalle de concrétisation de la forme affine $\hat{x}$ introduite dans l'équation 8.2. Nous notons $\mathcal{F}(\mathcal{I})$ l'ensemble des formes affines dont les indices appartiennent à l'ensemble fini $\mathcal{I}$. Les concrétisations $\gamma_I(\hat{x})$ et $\gamma_I(\hat{y})$ sont données dans le tableau 8.1. Nous ne pouvons pas fusionner directement les deux symboles ε_1 et ε_2 en un nouveau symbole ε_{12} en écrivant

$$x^\sharp = 4 + 0.1\varepsilon_{12} + 0.3\varepsilon_{12} + 0.2\varepsilon_3 = 4 + 0.4\varepsilon_{12} + 0.2\varepsilon_3 \tag{8.14}$$

et

$$y^\sharp = 1 + 0.2\varepsilon_{12} + 0.2\varepsilon_{12} + 0.3\varepsilon_4 = 1 + 0.4\varepsilon_{12} + 0.3\varepsilon_4 \ . \tag{8.15}$$

Par exemple, si nous procédons ainsi, nous aurions

$$\hat{x} - \hat{y} = 3 - 0.1\varepsilon_1 + 0.1\varepsilon_2 + 0.2\varepsilon_3 - 0.3\varepsilon_4 \tag{8.16}$$

et

$$x^\sharp - y^\sharp = 3 - 0.0\varepsilon_{12} + 0.2\varepsilon_3 - 0.3\varepsilon_4. \tag{8.17}$$

Dans ce cas, $[2.3, 3.7] = \gamma_I(\hat{x} - \hat{y}) \not\subseteq \gamma_I(x^\sharp - y^\sharp) = [2.5, 3.5]$ ce qui n'est pas ce que nous

Expr.	Valure	Intervalle
x	$4 + 0.1\varepsilon_1 + 0.3\varepsilon_2 + 0.2\varepsilon_3$	$[3.4, 4.6]$
y	$1 + 0.2\varepsilon_1 + 0.2\varepsilon_2 + 0.3\varepsilon_4$	$[0.3, 1.7]$
$x^\sharp$	$4 + 0.2\varepsilon_3 + 0.4\varepsilon_5$	$[3.4, 4.6]$
$y^\sharp$	$1 + 0.3\varepsilon_4 + 0.4\varepsilon_6$	$[0.3, 1.7]$
$x + y$	$5 + 0.3\varepsilon_1 + 0.5\varepsilon_2 + 0.2\varepsilon_3 + 0.3\varepsilon_4$	$[3.7, 6.3]$
$x - y$	$3 - 0.1\varepsilon_1 + 0.1\varepsilon_2 + 0.2\varepsilon_3 - 0.3\varepsilon_4$	$[2.3, 3.7]$
$x \times y$	$4 + 0.9\varepsilon_1 + 1.1\varepsilon_2 + 0.2\varepsilon_3 + 1.2\varepsilon_4 + 0.42\varepsilon_h$	$[0.18, 7.82]$
$x^\sharp + y^\sharp$	$5 + 0.2\varepsilon_3 + 0.3\varepsilon_4 + 0.4\varepsilon_5 + 0.4\varepsilon_6$	$[3.7, 6.3]$
$x^\sharp - y^\sharp$	$3 + 0.2\varepsilon_3 - 0.3\varepsilon_4 + 0.4\varepsilon_5 - 0.4\varepsilon_6$	$[1.7, 4.3]$
$x^\sharp \times y^\sharp$	$4 + 0.2\varepsilon_3 + 1.2\varepsilon_4 + 0.4\varepsilon_5 + 1.6\varepsilon_6 + 0.42\varepsilon_h$	$[0.18, 7.82]$

Table 8.1: Opérations élémentaires entre formes affines standard et formes affines compressées et leurs concrétisations.

attendons. Au lieu de cela, nous allons utiliser des formes affines compressées. Considérons m affines formes qui ont n symboles de bruits

$$\begin{gathered} x_{1,0} + x_{1,1}\varepsilon_1 + \cdots + x_{1,n}\varepsilon_n \\ \vdots \\ x_{m,0} + x_{m,1}\varepsilon_1 + \cdots + x_{m,n}\varepsilon_n \end{gathered} \tag{8.18}$$

Nous pouvons réduire le nombre de symboles de bruit en fusionnant plusieurs symboles de bruit. Soit S un ensemble d'indices de symboles de bruit que nous souhaitons regrouper, nous pouvons les regroupes en procédant comme suit:

$$\begin{gathered} \alpha_S(\sum_{i \in S} x_{1,i}\varepsilon_i) = (\sum_{i \in S} |x_{1,i}|)\varepsilon_{n+1} \\ \vdots \\ \alpha_S(\sum_{i \in S} x_{m,i}\varepsilon_i) = (\sum_{i \in S} |x_{m,i}|)\varepsilon_{n+m} \end{gathered} \tag{8.19}$$

Ici, m nouveaux symboles de bruits sont ajoutés afin de représenter les intervalles nouvellement générés. Dans notre exemple, nous allons remplacer $\hat{x}$ et $\hat{y}$ par les formes affines compressées.

$$x^\sharp = 4 + 0.2\varepsilon_3 + 0.4\varepsilon_5 \quad \text{et} \quad y^\sharp = 1 + 0.3\varepsilon_4 + 0.4\varepsilon_6 \ . \tag{8.20}$$

Nous pouvons observer dans le tableau 8.1 que l'intervalle de concrétisation des formes affines compressées contient l'intervalle de concrétisation des formes affines standard, c'est-à-dire que pour tout opérateur $* \in \{+, -, \times\}$, on a $\gamma_I(\hat{x} * \hat{y}) \subseteq \gamma_I(x^\sharp * y^\sharp)$.

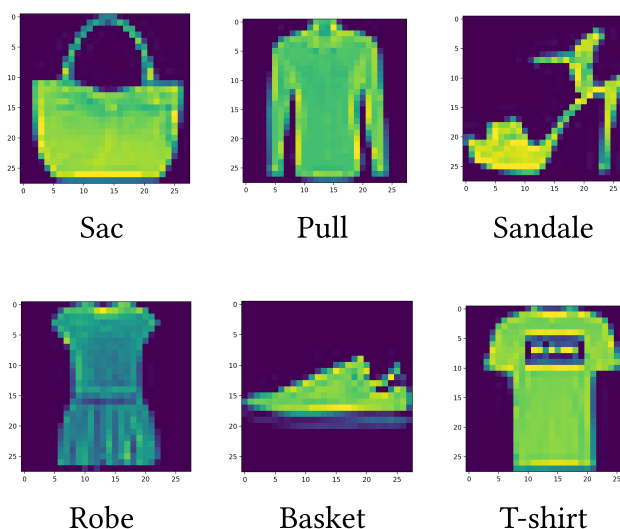


Figure 8.2: Images utilisées pour tester l'efficacité des formes affines.

8.4.2 Résultats expérimentaux

Dans cette section, nous décrivons notre méthodologie pour évaluer l'impact en termes de précision et de temps d'exécution de la fusion des symboles de bruit dans la validation des réseaux de neurones. Nous utilisons deux types de réseaux de neurones: entièrement connectés et convolutionnels qui prend en entrée des images de taille $(n \times n)$, les valeurs des pixels étant comprises entre -1 et 1. Chaque couche du réseau contient $(n \times n)$ neurones.

Lorsque on travaille avec des formes affines standard, l'entrée x_0 du réseau est un vecteur de $n \times n$ formes affines, chacune ayant également $n \times n$ symboles de bruit (un symbole de bruit par pixel). Soit c_{ij} la couleur du pixel (i, j) et ν l'intensité de son bruit. Ce pixel correspond à la composante $i \times n + j$ du vecteur d'entrée x_0 des formes affines, dont la valeur est :

$$x_0[i \times n + j] = c_{ij} + \nu c_{ij} \varepsilon_{i \times n + j} \quad . \tag{8.21}$$

Notons que nous associons la même intensité de bruit à tous les pixels et nous utilisons $\nu = 0.2$.

Dans nos expériences, nous considérons les cas $k = 81, 162, 324$ pour des images de taille 18×18 et $k = 196, 392, 784$ pour des images de taille 28×28. Intuitivement, la compression choisie pour nos expériences consiste à regrouper chaque β symboles de bruit consécutifs de la même ligne dans un seul nouveu symbol de bruit. nous avons considéré les cas $\beta = 2$ et $\beta = 4$.

Dans nos expériences, nous avons mesuré deux quantités: le temps nécessaire pour exécuter le réseau de neurones avec des formes affines compressées et la largeur moyenne w des intervalles correspondant aux concrétisations des formes affines compressées. Puisqu'un

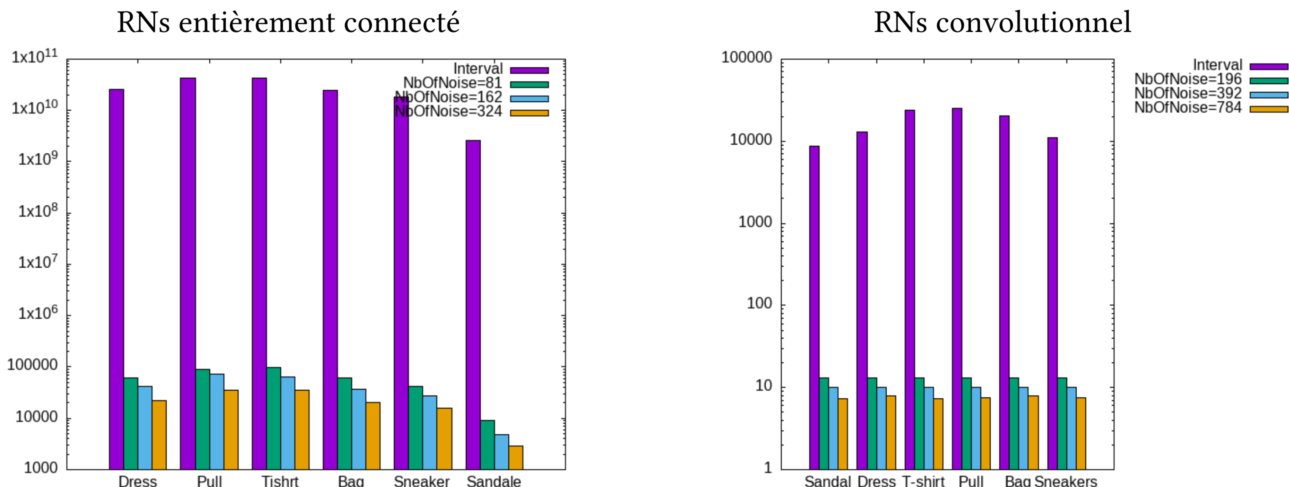


Figure 8.3: Propagation des formes affines correspondant à notre ensemble d'images.

vecteur de sortie x_L est composé de $N = n \times n$ formes affines, nous avons

$$w = \frac{\sum_{i=0}^{N} \text{width}\big(\gamma_I(x_L[i])\big)}{N} \quad . \tag{8.22}$$

Les résultats sont donnés dans les Figures 8.3 et 8.4 pour le cas des réseaux entièrement connectés et convolutionnels. Concernant la précision des calculs, on peut observer que la largeur de l'intervalle de concrétisation des formes affines augmente lorsque le nombre de symboles de bruit diminue. Notons que, pour nos exemples, l'arithmétique d'intervalle habituelle renvoie un intervalle dont la largeur est beaucoup plus grande. Ces résultats sont représentatifs de ce que nous obtenons pour d'autres images et paramètres.Cela montre que les formes affines compressées préservent la plupart de la précision des formes affines

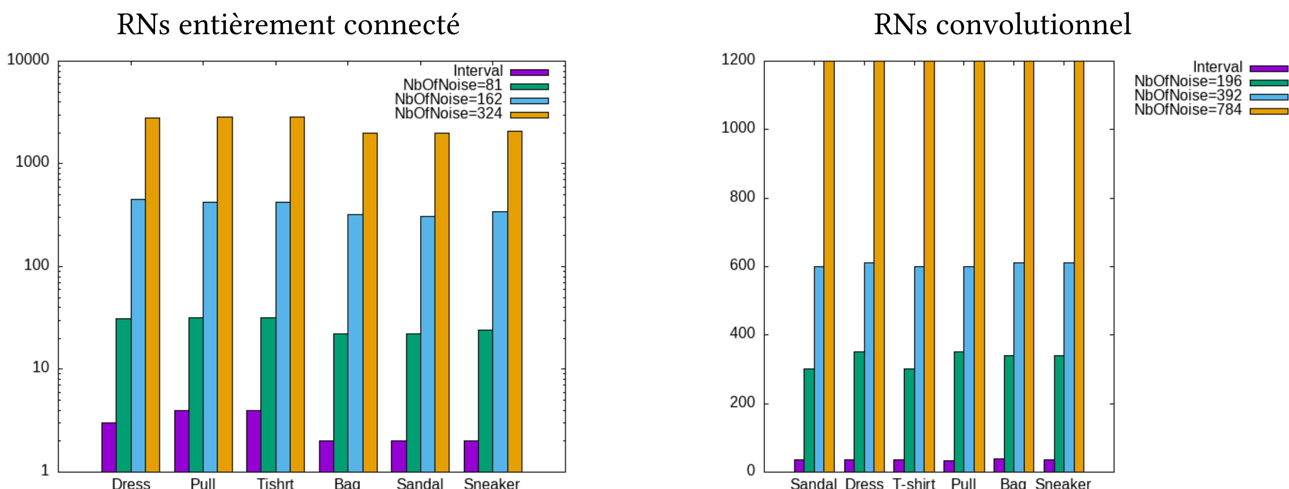


Figure 8.4: Temps d'exécution en fonction du nombre de symboles de bruit.

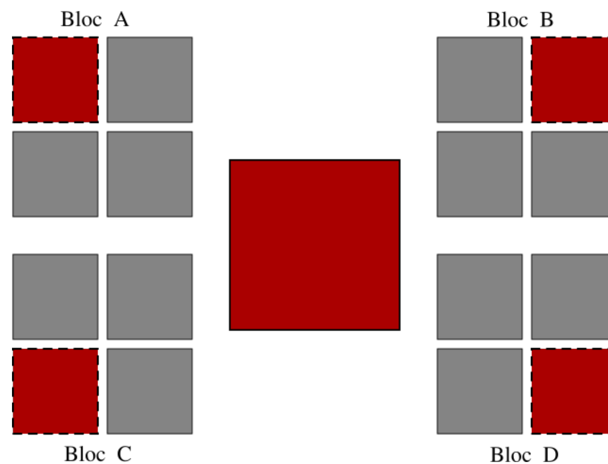


Figure 8.5: Diviser l'entrée en 4 blocs.

comparant à l'arithmétique d'intervalle.

La perte de précision due à la compression semble tout à fait acceptable au regard des gains en termes de temps d'exécution, où nous remarquons que le temps d'exécution est approximativement divisé par 2 à 4 lorsque nous utilisons des formes affines compressées (le temps d'exécution augmente lorsque le nombre de symboles de bruit augmente). Ces résultats sont en adéquation avec la complexité théorique: les additions et les produits à vecteur constant sont linéaires pour les formes affines et les formes affines compressées. Notons que les accélérations observées sont importantes et montrent que les formes affines compressées sont d'un grand intérêt pour la vérification des réseaux de neurones.

8.5 Bruit par bloc

Le bruit est couramment utilisé dans la validation des réseaux de neurones pour évaluer la robustesse de ces modèles. En pratique, ce bruit peut correspondre à des éléments comme une goutte d'eau, du brouillard ou des rayons de soleil qui viennent se superposer à une entrée donnée, soit de manière naturelle, soit de manière malveillante. Prenons l'exemple d'une goutte de pluie qui tombe sur un panneau de signalisation grâce à ce bruit, on peut tester la capacité d'un réseau à maintenir ses performances face à ces perturbations. Dans ce cas, on veut que le réseau soit capable de détecter le panneau de signalisation malgré les gouttes d'eau qui s'y ajoutent. la gestion de ces perturbations visuelles est souvent évaluée à l'aide de transformations abstraites, telles que les transformations affines. Cependant, ces transformations peuvent perdre en précision et être coûteuses en termes de calcul (temps et mémoire). Nous proposons dans cette section notre deuxième contribution appelée *Bruit par bloc* pour surmonter ces limitations. Le bruit par bloc simule des situations réelles dans lesquelles des parties spécifiques d'une image sont perturbées par l'insertion de symboles de bruit non nuls uniquement à l'intérieur d'une section donnée de l'image. Utilisons l'exemple de la Figure 8.5 pour clarifier cette approche. Nous divisons notre entrée en n ré-

gions (dans notre exemple, nous prenons $(n = 4)$. Ensuite, nous ajoutons des symboles de bruit non nuls à une seule région (parties rouges) et des symboles de bruit nuls à toutes les autres régions (parties grises). Nous répétons ce processus pour toutes les régions de notre entrée. À la fin du calcul, pour les couches linéaires (complètement connectées ou convolutionnelles sans ReLU), nous constatons que la somme de toutes les régions est exactement la même que les résultats obtenus avec l'entrée originale utilisant les formes affines (un seul bloc). Grâce à cette méthode, il est possible d'évaluer la résistance des réseaux de neurones à ces perturbations tout en préservant leur évolutivité et leur précision.

8.5.1 Principe de bruit par bloc

Dans cette section, nous présentons notre méthode de bruit par bloc de manière détaillée et expliquons la procédure de calcul associée. Pour illustrer notre approche, nous utilisons un exemple concret donné dans la figure 8.6, où nous appliquons notre méthode de bruit par bloc à une matrice 2×2 de transformations affines. Par exemple, $x_{00} = 2 + \varepsilon_1$. De

x

$2+\varepsilon_1$	$3+2\varepsilon_2$
$1-\varepsilon_3$	$4+2\varepsilon_4$

W

1	0	2	1
1	−1	0	1

b

−1
0

(a) Entrées données à une couche entièrement connectée.

$$\left(\overset{W}{\begin{bmatrix} 1 & 0 & 2 & 1 \end{bmatrix}} \times \overset{x}{\begin{bmatrix} 2+\varepsilon_1 \\ 3+2\varepsilon_2 \\ 1-\varepsilon_3 \\ 4+2\varepsilon_4 \end{bmatrix}} + \overset{b}{\begin{bmatrix} -1 \end{bmatrix}} \right)$$

(b) La première opération de la couche entièrement connectée.

$$\left(\overset{W}{\begin{bmatrix} 1 & -1 & 0 & 1 \end{bmatrix}} \times \overset{x}{\begin{bmatrix} 2+\varepsilon_1 \\ 3+2\varepsilon_2 \\ 1-\varepsilon_3 \\ 4+2\varepsilon_4 \end{bmatrix}} + \overset{b}{\begin{bmatrix} 0 \end{bmatrix}} \right)$$

(c) La deuxième opération de la couche entièrement connectée.

Figure 8.6: Opérations de la couche entièrement connectée.

plus, nous employons une matrice de poids W de taille 2×4 ainsi qu'un vecteur de biais b.

Cette technique est spécifiquement conçue pour l'évaluation des réseaux de neurones. Le bruit est incorporé à l'entrée x, tandis que W et b représentent respectivement les poids et le biais du réseau de neurones, contenant des coefficients scalaires.

on veut calculer des couches entièrement connectées définies comme ceci $y_j = \sum_{i=1}^{n} W_{ij}.x_i + b_j$. Pour notre exemple, nous avons :

$$y_1 = 7 + \varepsilon_1 - 2\varepsilon_3 + 2\varepsilon_4, \quad y_2 = 3 + \varepsilon_1 - 2\varepsilon_2 + 2\varepsilon_4. \tag{8.23}$$

Bloc 1

$2 + \varepsilon_1$	$3 + 0\varepsilon_2$
$1 - \varepsilon_3$	$4 + 0\varepsilon_4$

Bloc 2

$2 + 0\varepsilon_1$	$3 + 2\varepsilon_2$
$1 + 0\varepsilon_3$	$4 + 2\varepsilon_4$

Figure 8.9: La divison de x en 2 blocs.

Pour des raisons d'efficacité, nous souhaitons utiliser notre méthode de bruit par bloc à l'intérieur de x. Ainsi, nous divisons x en deux blocs. Par exemple, pour le bloc 1, nous ajoutons des symboles de bruit non nuls dans la partie rouge et des symboles de bruit nuls dans la partie bleu. À l'inverse, pour le bloc 2, nous introduisons des symboles de bruit dans les parties rouges et laissons les parties bleues sans bruit. Ensuite, nous calculons les couches entièrement connectées pour les blocs 1 et 2. Pour le bloc 1, nous avons :

$$b_{11} = 7 + \varepsilon_1 - 2\varepsilon_3, \quad b_{12} = 3 + \varepsilon_1, \tag{8.24}$$

et pour le bloc 2, nous avons

$$b_{21} = 7 + 2\varepsilon_4, \quad b_{22} = 3 - 2\varepsilon_2 + 2\varepsilon_4. \tag{8.25}$$

Nous obtenons que

$$b_{11} + b_{21} = 14 + \varepsilon_1 - 2\varepsilon_3 + 2\varepsilon_4 = y_1 - 7, \tag{8.26}$$

et

$$b_{12} + b_{22} = 6 + \varepsilon_1 - 2\varepsilon_2 + 2\varepsilon_4 = y_2 - 3. \tag{8.27}$$

Les constantes 7 et 3 soustraites de y_1 et y_2 sont présentes en raison de l'addition redondante des biais dans la formule et doivent être exclues.

De cette manière, nous réduisons le temps d'exécution et la mémoire nécessaire pour calculer une couche entièrement connectée sans perte de précision. On note que cette approche est également valable pour d'autres types de couches, en particulier les couches convolutions.

8.5.2 Fonction d'activation

Les formes affines constituent une approche efficace pour la vérification des réseaux de neurones, car elles sont rapides et exactes pour les transformations affines [106]. Toutefois, lorsqu'il s'agit de modéliser des fonctions d'activation non linéaires telles que Relu, l'abstraction du zonotope [27] n'est pas exacte. Par conséquent, une technique d'approximation [38, 62] qui crée un compromis entre le coût de calcul et la précision est nécessaire. Suivant l'approche développée dans [104], nous présentons dans cette section une méthode d'approximation de Relu qui établit un équilibre entre le coût de calcul et la précision.

Soit $\hat{x}$ une forme affine $\hat{x} = x_0 + \sum_{i=1}^{n} x_i \varepsilon_i$ donnée à une fonction ReLU. Soit

$$l_x = x_0 + \left(\sum_{i=1}^{n} |x_i|\right) \times -1 \quad \text{et} \quad u_x = x_0 + \left(\sum_{i=1}^{n} |x_i|\right) \times 1 \tag{8.28}$$

les bornes inférieure et supérieure respectivement, pour l'entrée $\hat{x}$. Le transformateur abstrait de la fonction d'activation ReLU est donné par :

$$ReLU(\hat{x}) = \begin{cases} \hat{x} & \text{pour } l_x \geq 0 \ , \\ 0 & \text{pour } u_x \leq 0 \ , \\ \lambda x + \nu + \nu\varepsilon & \text{sinon} \ . \end{cases}$$

où

$$\lambda = \frac{u_x}{u_x - l_x} \quad \text{et} \quad \nu = \frac{u_x(1-\lambda)}{2} \tag{8.29}$$

représentent respectivement le minimum de l'aire du parallélogramme dans le plan xy et le centre du zonotope dans l'axe vertical.

8.5.3 Résultats expérimentaux

Dans cette section, nous présentons certains résultats expérimentaux obtenus en utilisant cinq réseaux de neurones entraînés. Tous ces réseaux sont des classificateurs. Ils sont décrits dans le tableau 8.2. La première colonne du tableau indique le modèle et son entrée, la deuxième colonne le nombre de couches, la troisième colonne le nombre de neurones et la quatrième colonne le nombre de paramètres.

Nous utilisons les trois ensembles de données les plus couramment utilisés pour nos expériences: MNIST, Fashion-MNIST [33] et CIFAR-10 [1]. Nous avons entraînés deux types de réseaux de neurones: les réseaux entièrement connectés et les réseaux convolutionnels.

La figure 8.3 montre les résultats de notre mesure du temps nécessaire à l'exécution de chaque réseau de neurones avec différents nombres de blocs.

Nous remarquons que pour tous nos réseaux entièrement connectés, utilisant différents types de données (MNIST, Fashion-MNIST et CIFAR), le temps d'exécution diminue considérablement lorsque le nombre de blocs augmente. Par exemple, dans le cas d'un réseau

Modèle	entrée	couches	Neurones	Paramètres
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	$111,146$

Table 8.2: Configurations des modèles utilisées dans nos expériences.

Modèle	entrée	1 bloc	4 bloc	8 bloc
FashionMNIST				
CNN	(28×28)	1 h 35 min	1 h 6 min	1 h 15 min
FC	(28×28)	3 h 37 min	1 h 35 min	1 h 25 min
CIFAR				
CNN	(32×32)	5 h 33 min	4 h 10 min	4 h 27 min
MNIST				
CNN	(28×28)	1 h 33 min	1 h 12 min	1 h 28 min
FC	(28×28)	18 min	8 min	7 min

Table 8.3: Temps d'exécution comparés entre réseaux entièrement connectés et convolutionnels à différentes profondeurs.

entièrement connecté utilisant le jeu de données Fashion-MNIST, les temps d'exécution sont les suivants: 3 h 37 min, 1 h 35 min and 1 h 25 min min pour des blocs de 1, 4 et 8, respectivement. Il est démontré que lorsque nous utilisons notre méthode de bruits par blocs avec 4 bloc, le temps d'exécution est approximativement divisé par deux.

Nous pouvons observer que dans le cas des réseaux convolutionnels, les temps d'exécution diminuent au fur et à mesure que le nombre de blocs d'entrée augmente, mais ils commencent à augmenter dès que le nombre de blocs d'entrée dépasse 4. Par exemple, pour le CNN qui utilise le jeu de données Fashion-MNIST, les temps d'exécution sont les suivants: 1 h 35 min, 1 h 6 min and 1 h 15 min min pour des blocs de 1, 4 et 8, respectivement. Cela peut s'expliquer par le fait que, lorsque le nombre de blocs augmente, les chevauchements entre les blocs deviennent plus fréquents, ce qui entraîne des calculs redondants. En d'autres termes, les opérations de convolution sont répétées pour les pixels qui sont communs à différents blocs, ce qui entraîne une augmentation du temps d'exécution par rapport

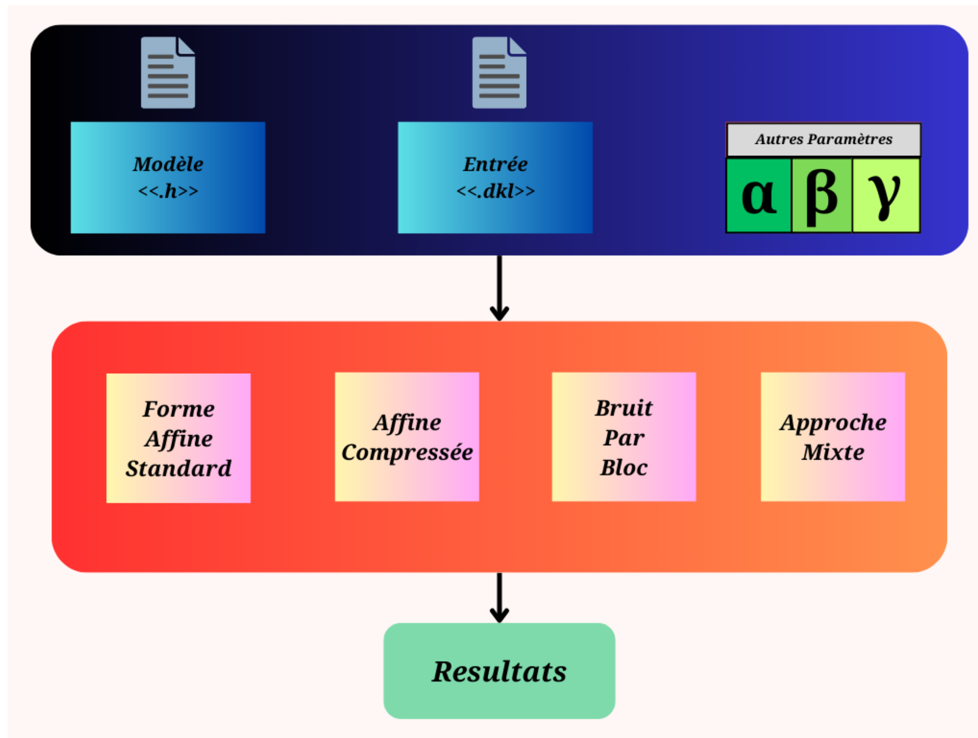


Figure 8.10: L'architecture de notre outil NNaff.

aux couches entièrement connectées.

8.6 L'outil NNAFF

Dans cette section nous décrivons notre outil NNAFF pour la vérification des réseaux de neurones. Ce dernier est implémenté en python. Un aperçu de son architecture est donné dans la figure 8.10. Elle est définie comme suit :

Paramétres

Les paramètres utilisés dans notre outil sont les suivants

- **Modèle :** renvoie à un fichier portant l'extension *.h5*, qui représente les paramètres de notre réseau de neurones entraîné (poids, biais, fonctions d'activation, etc.).
- **Entrée :** un fichier avec l'extension *.pkl* contenant les valeurs de pixels de nos entrées, ainsi que leur taille. Par défaut, nous considérons les images en niveaux de gris et normalisons les valeurs des pixels entre 0 et 1.
- **Autres paramètres :** représentent les paramètres que nous utiliserons dans les approches affines standard, affines compressées et de bruits par blocs. Les principaux paramètres sont les suivants :

1. α : l'intensité du bruit, représente le niveau de bruit associé à chaque pixel,
2. β : le facteur de compression, représente le nombre de symboles de bruit que nous souhaitons fusionner,
3. γ : le nombre de blocs, représente le nombre de blocs par lesquels nous souhaitons diviser notre entrée.

Modules

NNaff est composé de plusieurs modules, chacun implémentant différentes approches que nous avons développées : les formes affines standard, les formes affines compressées et le bruit par blocs.

- **Affine standard :** implémente la propagation avant et arrière du modèle en utilisant des formes affines standard.
- **Affine compressée :** implémente la propagation vers l'avant et vers l'arrière du modèle à l'aide de formes affines compressées.
- **Bruit par bloc :** implémente la propagation vers l'avant et vers l'arrière du modèle en utilisant l'approche du bruit par bloc.
- **Approche mixte :** implémente la propagation vers l'avant et vers l'arrière du modèle à l'aide d'une approche mixte, combinant les méthodes de affine compressées et bruit par blocs.

8.6.1 L'utilisation de l'outil

Dans cette section, nous allons montrer comment notre outil, NNaff, est utilisé. La commande d'exécution est la suivante :

```
./nnaff --model_path './file.h5' --data_path './file.pkl' --noise α
```

Ici, *model_path* représente le réseau de neurone que nous souhaitons analyser, *data_path* représente l'image que nous allons utiliser dans l'analyse et *noise* représente la valeur de bruit associée à chaque pixel. Par exemple, si nous voulons ajouter $0,04$ de bruit à chaque pixel, il nous suffit de définir $\alpha = 0,04$. En exécutant la commande ci-dessus, on obtient une analyse du modèle à l'aide de formes affines standard.

Si nous souhaitions utiliser l'approche forme affine compressée, il suffit d'ajouter le paramètre *–factor_comp* à la commande précédente :

```
./nnaff --model_path './file.h5' --data_path './file.pkl'\ --noise α
    --factor_comp β
```

Le paramètre *factor_compressed* représente le facteur de compression appliqué aux symboles de bruit dans l'approche de forme affine compressée. Il indique combien de symboles de bruit sont combinés ensemble pour réduire la complexité de calcule tout en préservant la précision de l'analyse du modèle.

Pour appliquer l'approche de bruit par bloc, il suffisait simplement inclure le paramètre *−block* :

```
./nnaff --model_path './file.h5' --data_path './file.pkl' \-- noiseα
    --bloc γ
```

Ici, *block* fait référence au nombre de blocs dans lesquels nous divisons notre entrée (par exemple, 4, 8, etc.).

Si nous voulons combiner les deux méthodes, c'est-à-dire la forme affine compressée et le bruit par bloc, nous devons mentionner simultanément ces deux paramètres *bloc* et *factor_comp* dans la ligne de commande :

```
./nnaff --model_path './file.h5' --data_path './file.pkl' \--noise α
    --factor_comp β --bloc γ
```

Il convient de noter que notre outil prend en charge les couches dense, flatten, conv2d et maxpool. En ce qui concerne les fonctions d'activation, il ne considère que la fonction ReLU. L'utilisateur a la liberté de choisir la méthode qui lui convient. La section suivante présente les avantages et les inconvénients de chaque méthode lorsqu'elles sont appliquées à différents modèles.

8.6.2 Résultats expérimentaux

Dans cette section, nous allons évaluer les performances de NNaff sur différents types de réseaux de neurones entraînés. L'architecture de ces différents modèles est donnée dans le tableau 8.4.

La première colonne du tableau 8.4 indique les ensembles de données, la deuxième colonne indique le nom du modèle, la troisième colonne indique le type de modèle, la quatrième colonne indique le nombre de couches et la dernière colonne indique le nombre de paramètres.

Notez que nous travaillons avec des images de taille $(28 \times 28 \times 1)$ extraites de l'ensemble de données MNIST, Fashion MNIST [33] et des images de taille $(32 \times 32 \times 1)$ extraites de l'ensemble de données CIFAR [1].

Pour l'approche sur forme affine compressée, nous avons décidé de regrouper les symboles de bruit par 2 (β = 2) et pour l'approche du bruit par blocs, nous divisons les images par 4 (γ = 4). Dans le cas de l'approche mixte, nous optons pour une combinaison où (β = 2) et (γ = 4).

Dataset	Modèles	Types	Couches	Paramètres
MNIST	**Conv1**	convolutional	11	136, 906
	Fc1	fully connected	8	593322
FASHION_MNIST	**Conv2**	convolutional	7	431, 242
	Fc2	fully connected	6	575, 050
CIFAR 10	**LeNet**	convolutional	8	44, 426
	AlexNet	convolutional	12	2, 910, 730
	Vgg16	convolutional	18	30, 613, 572

Table 8.4: Architectures de réseaux de neurones utilisées pour évaluer la performance de NNaff.

Dans notre analyse des résultats, nous observons une amélioration significative des temps d'exécution pour différents types de réseaux de neurones. En particulier, dans le cas des réseaux de neurones entièrement connectés, le temps d'exécution est fortement divisé par 2. Par exemple, pour le modèle FC2 avec Fashion MNIST, les temps d'exécution sont les suivants: 1 h, 45 min, 35 min, 49 h et 22 min pour les approches affine standard, affine compressée, bruit par bloc et mixte, ce qui est particulièrement remarquable.

En outre, pour les réseaux de neurones convolutionnels (CNN), nous constatons également une amélioration en termes de temps d'exécution, ce qui nous permet d'analyser de grands réseaux de neurones tels que VGG16 et AlexNet, qui ne pouvaient même pas être analysés à l'aide d'approche affine standard sur la même machine. Par exemple, pour le modèle AlexNet, les temps d'exécution sont les suivants: 34 h, 2 jours, et 32 h pour les approches affine compressée, bruit par bloc et mixte. Il est à noter qu'en raison du temps d'exécution prolongé, nous ne sommes pas en mesure d'effectuer une analyse à l'aide de l'approche affine standard.

En procédant à une analyse plus détaillée et en comparant les quatre techniques disponibles utilisées dans cette étude, il apparaît clairement que parmi les quatre, l'approche mixte est beaucoup plus rapide, en particulier lorsqu'il s'agit de grands réseaux de neurones comme VGG16, que même l'approche affine compressée ne parvient pas à analyser. Nous pouvons conclure que l'approche mixte offre un équilibre entre la précision et le temps d'exécution, ce qui en fait une solution efficace dans certaines situations.

Mnist				
Modèles	**Affine**	**Compressée**	**Bruit par bloc**	**Mixte**
Conv1	52 min	34 min	49 min	28 min
Fc1	1 h 50 min	38 min	53 min	24 min
Fashion Mnist				
Modèles	**Affine**	**Compressée**	**Bruit par bloc**	**Mixte**
Conv2	47 min	22 min	35 min	20 min
Fc2	1 h45 min	35 min	49min	22 min
Cifar				
Modelés	**Affine**	**Compressée**	**Bruit par bloc**	**Mixte**
LeNet	32 min	10 min	16 min	9 min
AlexNet	×	34 h	2 jours 12 h	32 h
Vgg16	×	×	20 jours	9jours

Table 8.5: Temps d'exécution de différents modèles.

8.7 Conclusion

Les travaux présentés dans cette thèse ont abordé le problème de la vérification des réseaux de neurones profonds (DNN) à l'aide de l'arithmétique affine. Les formes affines sont largement utilisées dans ce domaine car elles permettent d'obtenir des résultats plus précis en capturant les relations linéaires entre les variables, réduisant ainsi les sur-approximations. Cependant, leur principal inconvénient réside dans leur forte consommation de temps et de mémoire, ce qui complique la vérification de réseaux de grande taille. Pour surmonter ces limitations, nous avons proposé deux approches pour vérifier les réseaux de neurones profonds, appelées *formes affines compressée* et *bruit par blocs*. Les deux approches reposent sur les mêmes principes que les formes affines standard, mais intègrent des mécanismes visant à réduire le nombre de symboles de bruit, ce qui constitue un défi majeur dans la vérification des grands réseaux de neurones avec les formes affines standard.

L'idée principale derrière les formes affines compressées est de regrouper plusieurs sym-

boles de bruit en un seul symbole de bruit de manière à garantir la validité des calculs. Les formes affines compressées, étant une abstraction des formes affines standard, sont donc moins précises. Cependant, en utilisant moins de symboles de bruit, les calculs sont significativement plus rapides et consomment moins de mémoire que les formes affines standard. Les expériences menées sur des réseaux de neurones ont montré que notre approche affine compressée est plus rapide et consomme moins de mémoire comparant au formes affines standard. En d'autres termes, les formes affines compressées représentent un compromis entre la précision d'une part et le temps d'exécution ainsi que la consommation de mémoire d'autre part.

Notre deuxième approche, le bruit par blocs, consiste à ajouter des symboles de bruit non nuls uniquement dans une région définie d'une image. Intuitivement, nous avons divisé notre entrée en n blocs. Ensuite, nous ajoutons des symboles de bruit non nuls uniquement dans une région de chaque bloc, tout en laissant toutes les autres régions avec des symboles de bruit nuls. Nous répétons cette technique pour tous les blocs de notre entrée. Une fois le calcul terminé, nous constatons que la somme de tous les blocs correspond exactement aux résultats produits par l'entrée originale avec les formes affines standard. Le principal avantage de cette approche est que les calculs sont considérablement plus rapides par rapport aux formes affines standard. En moyenne, le temps d'exécution est réduit de 50% dans la majorité des cas. De plus, le bruit par blocs parallélisé a également été mis en œuvre, et les résultats montrent qu'il peut accélérer le calcul de $3,1$ à 4 fois.

L'ensemble de ces méthodes a été intégré dans un outil nommé NNaff, développé en Python. NNaff prend en entrée des réseaux de neurones et retourne le temps d'exécution nécessaire pour analyser ces réseaux en utilisant une méthode spécifique (affine compressée, bruit par bloc ou mixte). Un point clé de nos résultats est que, grâce à NNaff, nous sommes désormais capables d'analyser des réseaux de neurones à grande échelle tels qu'AlexNet [64], VGG16 [86] et ResNet [48], qui possèdent des millions de paramètres. Les méthodes traditionnelles basées sur les formes affines standard ne peuvent pas gérer de tels réseaux en raison de l'augmentation exponentielle du nombre de paramètres.

De nombreuses perspectives peuvent être envisagées pour ces travaux, nous détaillons ci-dessous quelques-unes.

Formes affines compressée améliorées

Notre méthode actuelle pour les formes affines compressées consiste à fusionner k symboles de bruit consécutifs d'une même ligne en un seul nouveau symbole. Dans une perspective d'analyse, il serait bénéfique d'explorer d'autres stratégies de regroupement.

- Nous pourrions commencer par fusionner les plus petits symboles de bruit. En pratique, cette méthode consiste à identifier les symboles de bruit dont les coefficients sont inférieurs à un certain seuil, puis à fusionner tous ces symboles de bruit en un

nouveau symbole. De cette manière, nous pouvons minimiser l'impact sur la précision et réduire le risque de sur-approximation. Cependant, nous devons faire attention au choix du seuil car si nous regroupons trop de symboles de bruit, nous risquons de perdre en précision, tandis que si nous en regroupons trop peu, nous risquons de ne pas obtenir de gain de performance. Par conséquent, nous devrions trouver un moyen de déterminer le seuil optimal. Une idée serait de commencer avec une valeur de seuil très élevée. Si, en raison de la sur-approximation, des fausses alertes surviennent, nous pourrons alors diminuer la valeur du seuil. Évidemment, ce processus peut être appliqué de manière répétée jusqu'à ce que nous trouvions le seuil optimal. Nous pouvons utiliser des techniques telles qu'un algorithme de bissection, similaire à ce qui est fait dans la bissection d'intervalles.

En expérimentant différentes stratégies de fusion des symboles de bruit, nous espérons améliorer l'efficacité et la précision de nos formes affines compressées.

Scalabilité : Extension aux réseaux plus complexes et aux autres fonctions d'activation

En ce qui concerne la scalabilité, notre approche actuelle se concentre sur les fonctions d'activation ReLU et utilise principalement des couches convolutionnelles et entièrement connectées. Pour relever le défi de la scalabilité, nous proposons au moins deux directions pour les travaux futurs.

La première direction consiste à étendre les types de fonctions d'activation utilisées dans notre analyse. En incorporant d'autres types de fonctions d'activation telles que $sigmoid$, $tanh$ et $softmax$ et en explorant les transformations abstraites pour ces fonctions, cela nous permettrait d'élargir notre compréhension de l'applicabilité de nos approches. Dans la même idée, nous prévoyons d'étendre notre analyse au-delà des couches convolutionnelles et entièrement connectées pour inclure d'autres types de couches, telles que les couches de normalisation et les couches résiduelles. Cette extension améliorera la scalabilité de notre outil, lui permettant de gérer des modèles plus complexes, y compris ceux similaires à GPT-4 [2] et PaLM [23].

La deuxième direction se concentre sur l'extension des types de réseaux de neurones analysés. Notre travail actuel se concentre principalement sur les réseaux convolutionnels, mais il serait intéressant d'explorer d'autres types de réseaux de neurones, tels que les réseaux de neurones récurrents (RNN) [75, 98]. Comme ils sont de plus en plus utilisés dans diverses tâches, telles que le traitement du langage naturel, il serait pertinent d'étudier comment nos méthodes se comportent dans de tels modèles.

En abordant ces aspects, nous visons à améliorer la capacité de NNaff à analyser une grande variété de réseaux de neurones.

Parallélisation et utilisation des GPU

De nos jours, les outils de calcul pour valider rigoureusement les performances des modèles de réseaux de neurones à grande échelle ont considérablement progressé. Les solveurs les plus efficaces utilisent des accélérateurs GPU hautement spécialisés. Ces outils sont essentiels pour le déploiement réussi des réseaux de neurones profonds dans les systèmes critiques pour la sécurité.

Dans notre travail actuel, le bruit par blocs utilise la parallélisation CPU en répartissant le calcul de chaque bloc sur différents processeurs. Pour améliorer encore les performances, nous cherchons à personnaliser l'algorithme avec une gestion explicite de la mémoire afin de paralléliser efficacement nos approches (bruit par blocs, affine compressée et mixte) sur les GPU. Cela pourrait entraîner des améliorations significatives des temps d'exécution puisque nous savons que l'utilisation des GPU peut fournir plus de puissance de calcul que les CPU.

De plus, nous prévoyons d'explorer la combinaison de deux méthodes de parallélisation: le parallélisme de modèle [58, 101] et le parallélisme de données [70]. Nous pouvons nous appuyer sur des bibliothèques GPU existantes telles que cuDNN [21] développée par NVIDIA, qui offre des implémentations hautement optimisées des opérations de réseaux de neurones. Cette approche de double parallélisation pourrait encore réduire les frais généraux de calcul et améliorer l'efficacité de notre outil.

À la fin de cette thèse, nous croyons que notre outil NNaff constitue une étape prometteuse pour relever le défi de la vérification des réseaux de neurones en termes de scalabilité et de précision. Il ouvre de nombreuses perspectives de recherche pour son amélioration et son application futures.

Bibliography

[1] Yehya Abouelnaga, Ola S Ali, Hager Rady, and Mohamed Moustafa. “Cifar-10: Knn-based ensemble of classifiers”. In: *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE. 2016, pp. 1192–1195.

[2] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. “Gpt-4 technical report”. In: *arXiv preprint arXiv:2303.08774* (2023).

[3] Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. “Face description with local binary patterns: Application to face recognition”. In: *IEEE transactions on pattern analysis and machine intelligence* 28.12 (2006), pp. 2037–2041.

[4] Aws Albarghouthi. “Introduction to neural network verification”. In: *Foundations and Trends® in Programming Languages* 7.1–2 (2021), pp. 1–157.

[5] Saad Albawi, Oguz Bayat, Saad Al-Azawi, and Osman N Ucan. “Social touch gesture recognition using convolutional neural network”. In: *Computational Intelligence and Neuroscience* 2018.1 (2018), p. 6973103.

[6] Ethem Alpaydin. *Machine learning*. MIT press, 2021.

[7] Julia Angwin, Jeff Larson, Surya Mattu, and Lauren Kirchner. “Machine bias”. In: *Ethics of data and analytics*. Auerbach Publications, 2022, pp. 254–264.

[8] SH Shabbeer Basha, Shiv Ram Dubey, Viswanath Pulabaigari, and Snehasis Mukherjee. “Impact of fully connected layers on performance of convolutional neural networks for image classification”. In: *Neurocomputing* 378 (2020), pp. 112–119.

[9] George Bebis and Michael Georgiopoulos. “Feed-forward neural networks”. In: *Ieee Potentials* 13.4 (1994), pp. 27–31.

[10] Barry Becker and Ronny Kohavi. *Adult*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20. 1996.

[11] George A Bekey and Kenneth Y Goldberg. *Neural networks in robotics*. Vol. 202. Springer Science & Business Media, 2012.

[12] Michel Bénichou, Jean-Michel Gauthier, Paul Girodet, Gerard Hentges, Gerard Ribière, and Olivier Vincent. “Experiments in mixed-integer linear programming”. In: *Mathematical programming* 1 (1971), pp. 76–94.

[13] Liane Bernstein, Alexander Sludds, Ryan Hamerly, Vivienne Sze, Joel Emer, and Dirk Englund. “Freely scalable and reconfigurable optical hardware for deep learning”. In: *Scientific reports* 11.1 (2021), p. 3144.

[14] Akhilan Boopathy, Tsui-Wei Weng, Pin-Yu Chen, Sijia Liu, and Luca Daniel. “Cnn-cert: An efficient framework for certifying robustness of convolutional neural networks”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 3240–3247.

[15] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. "Deep neural networks and tabular data: A survey". In: *IEEE transactions on neural networks and learning systems* (2022).

[16] Robin A Brown, Edward Schmerling, Navid Azizan, and Marco Pavone. "A unified view of SDP-based neural network verification through completely positive programming". In: *International conference on artificial intelligence and statistics*. PMLR. 2022, pp. 9334–9355.

[17] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. "A unified view of piecewise linear neural network verification". In: *Advances in Neural Information Processing Systems* 31 (2018).

[18] Rudy R Bunel, Ilker Turkaslan, Philip Torr, Pushmeet Kohli, and Pawan K Mudigonda. "A unified view of piecewise linear neural network verification". In: *Advances in Neural Information Processing Systems* 31 (2018).

[19] Gano B Chatterji and Banavar Sridhar. "Neural network based air traffic controller workload prediction". In: *Proceedings of the 1999 American control conference (Cat. No. 99CH36251)*. Vol. 4. IEEE. 1999, pp. 2620–2624.

[20] Jieqiu Chen and Samuel Burer. "Globally solving nonconvex quadratic programming problems via completely positive programming". In: *Mathematical Programming Computation* 4.1 (2012), pp. 33–52.

[21] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759* (2014).

[22] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. "Rigorous floating-point mixed-precision tuning". In: *ACM SIGPLAN Notices* 52.1 (2017), pp. 300–315.

[23] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. "Palm: Scaling language modeling with pathways". In: *Journal of Machine Learning Research* 24.240 (2023), pp. 1–113.

[24] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555* (2014).

[25] João Luiz Dihl Comba and Jorge Stol. "A ne arithmetic and its applications to computer graphics". In: *Proceedings of VI SIBGRAPI (Brazilian Symposium on Computer Graphics and Image Processing)*. 1993, pp. 9–18.

[26] Patrick Cousot. "Abstract interpretation based formal methods and future challenges". In: *Informatics: 10 Years Back, 10 Years Ahead*. Springer, 2001, pp. 138–156.

[27] Patrick Cousot and Radhia Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1977, pp. 238–252.

[28] Patrick Cousot and Radhia Cousot. “Static determination of dynamic properties of programs”. In: *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod. 1976, pp. 106–130.

[29] Patrick Cousot and Nicolas Halbwachs. “Automatic discovery of linear restraints among variables of a program”. In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1978, pp. 84–96.

[30] Navneet Dalal and Bill Triggs. “Histograms of oriented gradients for human detection”. In: *2005 IEEE computer society conference on computer vision and pattern recognition (CVPR’05)*. Vol. 1. Ieee. 2005, pp. 886–893.

[31] Luiz Henrique De Figueiredo and Jorge Stolfi. “Affine arithmetic: concepts and applications”. In: *Numerical Algorithms* 37 (2004), pp. 147–158.

[32] Leonardo De Moura and Nikolaj Bjørner. “Satisfiability modulo theories: An appetizer”. In: *Brazilian Symposium on Formal Methods*. Springer. 2009, pp. 23–36.

[33] Li Deng. “The mnist database of handwritten digit images for machine learning research [best of the web]”. In: *IEEE signal processing magazine* 29.6 (2012), pp. 141–142.

[34] Truong-Dong Do, Minh-Thien Duong, Quoc-Vu Dang, and My-Ha Le. “Real-time self-driving car navigation using deep neural network”. In: *2018 4th International Conference on Green Technology and Sustainable Development (GTSD)*. IEEE. 2018, pp. 7–12.

[35] Souradeep Dutta, Susmit Jha, Sriram Sankaranarayanan, and Ashish Tiwari. “Output range analysis for deep feedforward neural networks”. In: *NASA Formal Methods Symposium*. Springer. 2018, pp. 121–138.

[36] Krishnamurthy Dvijotham, Robert Stanforth, Sven Gowal, Timothy A Mann, and Pushmeet Kohli. “A Dual Approach to Scalable Verification of Deep Networks.” In: *UAI*. Vol. 1. 2. 2018, p. 3.

[37] Ruediger Ehlers. “Formal verification of piece-wise linear feed-forward neural networks”. In: *Automated Technology for Verification and Analysis: 15th International Symposium, ATVA 2017, Pune, India, October 3–6, 2017, Proceedings 15*. Springer. 2017, pp. 269–286.

[38] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. “Ai2: Safety and robustness certification of neural networks with abstract interpretation”. In: *2018 IEEE symposium on security and privacy (SP)*. IEEE. 2018, pp. 3–18.

[39] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics.* JMLR Workshop and Conference Proceedings. 2011, pp. 315–323.

[40] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. "An overview on the static code analysis approach in software development". In: *Faculdade de Engenharia da Universidade do Porto, Portugal* 16 (2009).

[41] Divya Gopinath, Hayes Converse, Corina Pasareanu, and Ankur Taly. "Property inference for deep neural networks". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE. 2019, pp. 797–809.

[42] Eric Goubault. "Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT". In: *International Static Analysis Symposium.* Springer. 2013, pp. 1–3.

[43] Eric Goubault and Sylvie Putot. "A zonotopic framework for functional abstractions". In: *Formal Methods in System Design* 47 (2015), pp. 302–360.

[44] Eric Goubault and Sylvie Putot. "Static analysis of numerical algorithms". In: *International Static Analysis Symposium.* Springer. 2006, pp. 18–34.

[45] Eric Goubault, Sylvie Putot, and Franck Védrine. "Modular static analysis with zonotopes". In: *Static Analysis: 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings 19.* Springer. 2012, pp. 24–40.

[46] Kai Han, Yunhe Wang, Qi Tian, Jianyuan Guo, Chunjing Xu, and Chang Xu. "Ghostnet: More features from cheap operations". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 2020, pp. 1580–1589.

[47] Eldon R Hansen. "A generalized interval arithmetic". In: *Interval Mathematics: Proceedings of the International Symposium Karlsruhe, West Germany, May 20–24, 1975.* Springer. 1975, pp. 7–18.

[48] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2016, pp. 770–778.

[49] Timothy Hickey, Qun Ju, and Maarten H Van Emden. "Interval arithmetic: From principles to implementation". In: *Journal of the ACM (JACM)* 48.5 (2001), pp. 1038–1068.

[50] Thibault Hilaire. *AffaPy.* `https://gitlab.lip6.fr/hilaire/affapy`. 2020.

[51] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *IEEE Signal processing magazine* 29.6 (2012), pp. 82–97.

[52] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[53] Hans Hofmann. *Statlog (German Credit Data).* UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5NC77. 1994.

[54] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint:1704.04861* (2017).

[55] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. "Safety verification of deep neural networks". In: *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30.* Springer. 2017, pp. 3–29.

[56] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized neural networks". In: *Advances in neural information processing systems* 29 (2016).

[57] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International conference on machine learning*. pmlr. 2015, pp. 448–456.

[58] Zhihao Jia, Matei Zaharia, and Alex Aiken. "Beyond Data and Model Parallelism for Deep Neural Networks." In: *Proceedings of Machine Learning and Systems* 1 (2019), pp. 1–13.

[59] Yiming Jiang, Chenguang Yang, Jing Na, Guang Li, Yanan Li, and Junpei Zhong. "A brief review of neural networks based learning and control and their applications for robots". In: *Complexity* 2017.1 (2017), p. 1895897.

[60] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. "Policy compression for aircraft collision avoidance systems". In: *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC).* IEEE. 2016, pp. 1–10.

[61] Ioannis Kanellopoulos and Graeme G Wilkinson. "Strategies and best practice for neural network image classification". In: *International Journal of Remote Sensing* 18.4 (1997), pp. 711–725.

[62] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. "Reluplex: An efficient SMT solver for verifying deep neural networks". In: *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I 30*, pp. 97–117.

[63] Guy Katz, Derek A Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, et al. "The marabou framework for verification and analysis of deep neural networks". In: *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I 31.* Springer. 2019, pp. 443–452.

[64] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[65] Anders Krogh. "What are artificial neural networks?" In: *Nature biotechnology* 26.2 (2008), pp. 195–197.

[66] Simon B Laughlin and Terrence J Sejnowski. "Communication in neuronal networks". In: *Science* 301.5641 (2003), pp. 1870–1874.

[67] COMBA Joao LD. "Affine arithmetic and its applications to computer graphics". In: *SIBGRAPI'93, Recife, PE (Brazil), October* (1993).

[68] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[69] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. "Putting static analysis to work for verification: A case study". In: *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis.* 2000, pp. 26–38.

[70] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. "Pytorch distributed: Experiences on accelerating data parallel training". In: *arXiv preprint arXiv:2006.15704* (2020).

[71] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. "A survey of convolutional neural networks: analysis, applications, and prospects". In: *IEEE transactions on neural networks and learning systems* 33.12 (2021), pp. 6999–7019.

[72] Tony Lindeberg. "Scale invariant feature transform". In: (2012).

[73] Nam Mai-Duy and Thanh Tran-Cong. "Numerical solution of differential equations using multiquadric radial basis function networks". In: *Neural networks* 14.2 (2001), pp. 185–199.

[74] Paul D McNelis. *Neural networks in finance: gaining predictive edge in the market.* Elsevier, 2005.

[75] Larry R Medsker and Lakhmi Jain. "Recurrent neural networks". In: *Design and Applications* 5.64-67 (2001), p. 2.

[76] Mark Huasong Meng, Guangdong Bai, Sin Gee Teo, Zhe Hou, Yan Xiao, Yun Lin, and Jin Song Dong. "Adversarial robustness of deep neural networks: A survey from a formal verification perspective". In: *IEEE Transactions on Dependable and Secure Computing* (2022).

[77] Frédéric Messine. "Extentions of Affine Arithmetic: Application to Unconstrained Global Optimization." In: *J. Univers. Comput. Sci.* 8.11 (2002), pp. 992–1015.

[78] Frédéric Messine and Ahmed Touhami. "A general reliable quadratic form: An extension of affine arithmetic". In: *Reliable Computing* 12.3 (2006), pp. 171–192.

[79] Antoine Miné. "The octagon abstract domain". In: *Higher-order and symbolic computation* 19 (2006), pp. 31–100.

[80] Ramon E Moore and CT Yang. "Interval analysis I". In: *Technical Document LMSD-285875, Lockheed Missiles and Space Division, Sunnyvale, CA, USA* (1959).

[81] Prakash M Nadkarni, Lucila Ohno-Machado, and Wendy W Chapman. "Natural language processing: an introduction". In: *Journal of the American Medical Informatics Association* 18.5 (2011), pp. 544–551.

[82] Vinod Nair and Geoffrey E Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[83] Nina Narodytska, Shiva Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. "Verifying properties of binarized deep neural networks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[84] Mesut Ozdag. "Adversarial attacks and defenses against deep neural networks: a survey". In: *Procedia Computer Science* 140 (2018), pp. 152–161.

[85] Luca Pulina and Armando Tacchella. "An abstraction-refinement approach to verification of artificial neural networks". In: *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer. 2010, pp. 243–257.

[86] Hussam Qassim, Abhishek Verma, and David Feinzimer. "Compressed residual-VGG16 CNN model for big data places image recognition". In: *2018 IEEE 8th annual computing and communication workshop and conference (CCWC)*. IEEE. 2018, pp. 169–175.

[87] R Regin, S Suman Rajest, and T Shynu. "A review of secure neural networks and big data mining applications in financial risk assessment". In: *Central Asian Journal of Innovations on Tourism Management and Finance* 4.2 (2023), pp. 73–90.

[88] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[89] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[90] Federico Rossi, Cinzia Bernardeschi, and Marco Cococcioni. "Neural networks in closed-loop systems: verification using interval arithmetic and formal prover". In: *Engineering Applications of Artificial Intelligence* 137 (2024), p. 109238.

[91] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. "Reachability analysis of deep neural networks with provable guarantees". In: *arXiv preprint arXiv:1805.02242* (2018).

[92] Wenjie Ruan, Min Wu, Youcheng Sun, Xiaowei Huang, Daniel Kroening, and Marta Kwiatkowska. "Global robustness evaluation of deep neural networks with provable guarantees for the hamming distance". In: *IJCAI-19* (2019).

[93] Siegfried M Rump and Masahide Kashiwagi. “Implementation and improvements of affine arithmetic”. In: *Nonlinear Theory and Its Applications, IEICE* 6.3 (2015), pp. 341–359.

[94] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. “Imagenet large scale visual recognition challenge”. In: *International journal of computer vision* 115 (2015), pp. 211–252.

[95] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. “Mobilenetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 4510–4520.

[96] Chiharu Sano. *Japanese Credit Screening*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5259N. 1992.

[97] Dominik Scherer, Andreas Müller, and Sven Behnke. “Evaluation of pooling operations in convolutional architectures for object recognition”. In: *International conference on artificial neural networks*. Springer. 2010, pp. 92–101.

[98] Robin M Schmidt. “Recurrent neural networks (rnns): A gentle introduction and overview”. In: *arXiv preprint arXiv:1912.05911* (2019).

[99] Randall S Sexton, Bahram Alidaee, Robert E Dorsey, and John D Johnson. “Global optimization for artificial neural networks: A tabu search application”. In: *European Journal of Operational Research* 106.2-3 (1998), pp. 570–584.

[100] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. “Activation functions in neural networks”. In: *Towards Data Sci* 6.12 (2017), pp. 310–316.

[101] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. “Megatron-lm: Training multi-billion parameter language models using model parallelism”. In: *arXiv preprint arXiv:1909.08053* (2019).

[102] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[103] Gagandeep Singh, Rupanshu Ganvir, Markus Püschel, and Martin Vechev. “Beyond the single neuron convex barrier for neural network certification”. In: *Advances in Neural Information Processing Systems* 32 (2019).

[104] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. “Fast and effective robustness certification”. In: *Advances in neural information processing systems* 31 (2018).

[105] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. “An abstract domain for certifying neural networks”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–30.

[106] Asma Soualah and Matthieu Martel. "Efficient Neural Network Validation with Affine Forms". In: *2022 6th International Conference on System Reliability and Safety (ICSRS)*. IEEE. 2022, pp. 179–185.

[107] Asma Soualah, Matthieu Martel, and Stéphane Abide. "Scaling-up the Analysis of Neural Networks by Affine Forms: A Block-Wise Noising Approach". In: (2023).

[108] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. "Going deeper with convolutions". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[109] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[110] Ryo Takahashi, Takashi Matsubara, and Kuniaki Uehara. "Scale-invariant recognition by weight-shared cnns in parallel". In: *Asian Conference on Machine Learning*. PMLR. 2017, pp. 295–310.

[111] Vincent Tjeng, Kai Xiao, and Russ Tedrake. "Evaluating robustness of neural networks with mixed integer programming". In: *arXiv preprint arXiv:1711.07356* (2017).

[112] Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. "Perfectly parallel fairness certification of neural networks". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–30.

[113] Caterina Urban and Antoine Miné. "A review of formal methods applied to machine learning". In: *arXiv preprint arXiv:2104.02466* (2021).

[114] Eduru Harindra Venkatesh, Yelleti Vivek, Vadlamani Ravi, and Orsu Shiva Shankar. "Parallel and Streaming Wavelet Neural Networks for Classification and Regression under Apache Spark". In: *arXiv preprint arXiv:2209.03056* (2022).

[115] Alex Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. "Phoneme recognition using time-delay neural networks". In: *IEEE transactions on acoustics, speech, and signal processing* 37.3 (1989), pp. 328–339.

[116] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. "Efficient formal safety analysis of neural networks". In: *Advances in neural information processing systems* 31 (2018).

[117] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. "Formal security analysis of neural networks using symbolic intervals". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 1599–1614.

[118] Lily Weng, Huan Zhang, Hongge Chen, Zhao Song, Cho-Jui Hsieh, Luca Daniel, Duane Boning, and Inderjit Dhillon. "Towards fast computation of certified robustness for relu networks". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 5276–5285.

[119] Eric Wong and Zico Kolter. “Provable defenses against adversarial examples via the convex outer adversarial polytope”. In: *International conference on machine learning*. PMLR. 2018, pp. 5286–5295.

[120] Jianxin Wu. “Introduction to convolutional neural networks”. In: *National Key Lab for Novel Software Technology. Nanjing University. China* 5.23 (2017), p. 495.

[121] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. “Deep image: Scaling up image recognition”. In: *arXiv preprint arXiv:1501.02876* (2015).

[122] Hyeon-Joong Yoo. “Deep convolution neural networks in computer vision: a review”. In: *IEIE Transactions on Smart Processing and Computing* 4.1 (2015), pp. 35–43.

[123] Hongce Zhang, Maxwell Shinn, Aarti Gupta, Arie Gurfinkel, Nham Le, and Nina Narodytska. “Verification of recurrent neural networks for cognitive tasks via reachability analysis”. In: *ECAI 2020*. IOS Press, 2020, pp. 1690–1697.

[124] Huan Zhang, Tsui-Wei Weng, Pin-Yu Chen, Cho-Jui Hsieh, and Luca Daniel. “Efficient neural network robustness certification with general activation functions”. In: *Advances in neural information processing systems* 31 (2018).

[125] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. “Shufflenet: An extremely efficient convolutional neural network for mobile devices”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 6848–6856.

[126] Rui Zhao, Ruqiang Yan, Zhenghua Chen, Kezhi Mao, Peng Wang, and Robert X Gao. “Deep learning and its applications to machine health monitoring”. In: *Mechanical Systems and Signal Processing* 115 (2019), pp. 213–237.

[127] Daniel Zügner, Oliver Borchert, Amir Akbarnejad, and Stephan Günnemann. “Adversarial attacks on graph neural networks: Perturbations and their patterns”. In: *ACM Transactions on Knowledge Discovery from Data (TKDD)* 14.5 (2020), pp. 1–31.

Abstract

Affine forms, which are an extension of interval arithmetic, have been successfully used to assess the robustness of neural networks as well as to explain their decisions. They are well-suited to this application domain since they capture affine relations between variables and since neural networks mostly perform affine computations. However, a drawback is that affine forms are time and memory consuming, making the verification of industrial-size neural network difficult or impossible. In this thesis, we introduce two new approaches to overcome these limitations. Affine compressed, merged serval noise symbols into a single noise symbol in a way that ensures the soundness of the computations. 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 methods, it is possible to assess neural networks resilience to these disturbances while preserving scalability and accuracy. Both methods have been implemented in a tool named, NNAFF. The experimental results have demonstrated the effectiveness of these methods, striking a balance between precision and execution time.

Keywords: *Artificial Intelligence, Static Analysis, Abstract Interpretaion, Scalability, Analysis Time Optimization,*

Résumé

Les formes affines, qui sont une extension de l'arithmétique des intervalles, ont été utilisées avec succès pour évaluer la robustesse des réseaux de neurones et expliquer leurs décisions. Elles sont bien adaptées à ce domaine car elles capturent les relations linéaires entre les variables étant donné que les réseaux de neurones effectuent en majorité des calculs affines. Cependant, un inconvénient majeur de ces formes utilisées dans le contexte des réseaux de neurones, c'est qu'elles consomment beaucoup de temps et de mémoire, ce qui rend la vérification des réseaux de taille industrielle difficile voire impossible. Dans cette thèse, nous introduisons deux nouvelles approches pour surmonter ces limitations: affine compressée, qui fusionne plusieurs symboles de bruit en un seul tout en garantissant la solidité des calculs. Le bruit par bloc, qui simule des situations réelles où des parties particulières d'une image sont perturbées en insérant des symboles de bruit non nuls uniquement dans une section donnée de l'image. Ces méthodes permettent d'évaluer la résilience des réseaux de neurones à ces perturbations tout en préservant leur évolutivité et leur précision. Les deux méthodes ont été implémentées dans un outil nommé, NNAFF, et les résultats expérimentaux ont démontré leur efficacité, trouvant un équilibre entre précision et temps d'exécution.

Mots-clés : *Intelligence Artificielle, Analyse statique, Interprétation abstraite, Scalabilité, Optimisation du Temps d'Analyse,*

A.1 Neural Networks Evaluated

We test with five trained networks which are described below:

Fashion-MNIST

- CNN
  Input$(28 \times 28 \times 1) \rightarrow$ Conv$(32 \times 3 \times 3) \rightarrow$ Relu $\rightarrow$ MaxPool$(2 \times 2) \rightarrow$ Conv$(64 \times 3 \times 3) \rightarrow$ Relu $\rightarrow$ MaxPool$(2 \times 2) \rightarrow$ Flatten $\rightarrow$ Fc$(256) \rightarrow$ Relu $\rightarrow$ Fc(10)

```
from tensorflow import keras

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.
    fashion_mnist.load_data()

# Preprocess the data
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_train = x_train.astype('float32') / 255.0
y_train = keras.utils.to_categorical(y_train)

# Define the model architecture
model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu',
   input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Flatten(),
    keras.layers.Dense(256, activation='relu'),
    keras.layers.Dense(10, activation='softmax'),
])
model.summary()

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5)
# Save the model weights and biases to an HDF5 file
model.save('nn_conv_fashion.h5')
```

Listing 1: TensorFlow code for CNN on Fashion MNIST

- FC
  Input$(28 \times 28 \times 1) \rightarrow$ Flatten $\rightarrow$ Fc$(512) \rightarrow$ Relu $\rightarrow$ Fc$(256) \rightarrow$ Relu $\rightarrow$ Fc$(128) \rightarrow$

Relu → Fc(64) → Relu → Fc(10) → y

```
from tensorflow import keras

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.
    fashion_mnist.load_data()

# Preprocess the data
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_train = x_train.astype('float32') / 255.0
y_train = keras.utils.to_categorical(y_train)

# Define the model architecture
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28, 1)),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dense(256, activation='relu'),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(10, activation='softmax'),
])
model.summary()

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5)

# Save the model weights and biases to an HDF5 file
model.save('nn_fc_fashion.h5')
```

Listing 2: TensorFlow code for FC Model on Fashion MNIST

MNIST

- CNN
  Input$(28 \times 28 \times 1)$ → Conv$(64 \times 3 \times 3)$ → Relu → MaxPool(2×2) → Conv$(64 \times 3 \times 3)$ → Relu → MaxPool(2×2) → Conv$(64 \times 3 \times 3)$ → Relu → Flatten → Fc(128) → Relu → Fc(64) → Relu → Fc(10) → y

```
from tensorflow import keras

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.
    load_data()

```

```
# Preprocess the data
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_train = x_train.astype('float32') / 255.0
y_train = keras.utils.to_categorical(y_train)

# Define the model architecture
model = keras.Sequential([
    keras.layers.Conv2D(64, (3, 3), activation='relu',
   input_shape=(28, 28, 1)),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.Flatten(),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(10, activation='softmax'),
])
model.summary()

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5)

# Save the model weights and biases to an HDF5 file
model.save('nn_conv_mnist.h5')
```

Listing 3: TensorFlow code for CNN Model on MNIST

- FC
  Input$(28 \times 28 \times 1) \rightarrow$ Flatten $\rightarrow$ Fc$(128) \rightarrow$ Relu $\rightarrow$ Fc$(64) \rightarrow$ Relu $\rightarrow$ Fc$(32) \rightarrow$ Relu $\rightarrow$ Fc$(10) \rightarrow$ y

```
from tensorflow import keras

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.
   load_data()

# Preprocess the data
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1))
x_train = x_train.astype('float32') / 255.0
y_train = keras.utils.to_categorical(y_train)

# Define the model architecture
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28, 1)),
```

```
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(32, activation='relu'),
    keras.layers.Dense(10, activation='softmax'),
])
model.summary()

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=5)

# Save the model weights and biases to an HDF5 file
model.save('nn_fc_mnist.h5')
```

Listing 4: TensorFlow code for FC Model on MNIST

Cifar-10

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

```
from tensorflow import keras

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.
    load_data()

# Preprocess the data
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
y_train = keras.utils.to_categorical(y_train)
y_test = keras.utils.to_categorical(y_test)
print(x_train.shape)

# Define the model architecture
model = keras.Sequential([
    keras.layers.Conv2D(32, (3, 3), activation='relu',
    input_shape=x_train.shape[1:]),
    keras.layers.Conv2D(32, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
    keras.layers.Dropout(0.25),
    keras.layers.Conv2D(32, (3, 3), activation='relu'),
    keras.layers.Conv2D(32, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D((2, 2)),
```

```
    keras.layers.Dropout(0.25),
    keras.layers.Flatten(),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(10, activation='softmax')
])
model.summary()

# Compile model
model.compile(loss='categorical_crossentropy', optimizer='adam',
    metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=20, batch_size=128,
   validation_data=(x_test, y_test))

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)

# Save the model weights and biases to an HDF5 file
model.save('cifar.h5')
```

Listing 5: TensorFlow code for CNN Model on CIFAR-10

A.2 Loading and saving Images with Pickle library

In this section, we present a Python code that demonstrates how to utilize the pickle library for saving and loading image data. The purpose of this code is to illustrate the process of storing a dataset of images, such as those from the Fashion MNIST dataset, into a file and then retrieving it for use within our NNaff tool.

```
import tensorflow as tf
import pickle
import os

def save_data(a):
    # Load the Fashion MNIST data from the server
    fashion_mnist = tf.keras.datasets.fashion_mnist
    (images, targets), (images_test, targets_test) = fashion_mnist.
   load_data()

    # Obtain only a portion of the dataset
    images = images[:10000]
    targets = targets[:10000]
    images = images.astype('float32') / 255
    images_test = images_test.astype('float32') / 255
    images = images.reshape(10000, 28, 28, 1)
    images = images[a]
```

```
17
18     # Store images and their shape information in a dictionary
19     data = {
20         'image_shape': images.shape,
21         'images': images
22     }
23
24     # Save the data dictionary to a file using pickle
25     if not os.path.isfile('data.pkl'):
26         with open('data.pkl', 'wb') as file:
27             pickle.dump(data, file)
28
29     return images
30
31 # Example usage
32 a = 2  # Example index
33 save_images = save_data(a)
34
35 # Load the data from the file using pickle
36 with open('data.pkl', 'rb') as file:
37     data_loaded = pickle.load(file)
```

Listing 6: Loading and Saving Fashion MNIST Images with pickle