CENTRALE LYON

N° d'ordre NNT: 2025ECDL0036

THESE DE DOCTORAT DE L'ECOLE CENTRALE DE LYON
membre de l'Université de Lyon

Ecole Doctorale N°160
EEA - Electronique, Electrotechnique, Automatique de Lyon

Spécialité de doctorat : Electronique, Micro et Nanoélectronique, optique et laser

Soutenue publiquement le 1 décembre 2025, par :

Salvatore Pappalardo

Investigating Hardware for Resilient Artificial Intelligence

Devant le jury composé de:

Andrea PINNA, Professeur des universités, Sorbonne Université LIP6	Rapporteur
Leticia Maria BOLZANI PÖHLS, Directrice de recherche, Leibniz IHP	Rapporteure
Angeliki KRITIKAKOU, Directrice de recherche, Université de Rennes	Examinatrice
Alberto BOSIO, Professeur des universités, École Centrle de Lyon, INL	Directeur de thèse
Bastien DEVEAUTOUR, Maître de conférences, Nantes Université	Co-encadrant de thèse

Unité de recherche : INL - Institut des Nanotechnologies de Lyon

Table of Contents

Abstract i
List of Publications iii
How to Read this Manuscript v

1 Introduction **1**
1.1 (Artificial) Intelligence 1
1.1.1 Intelligent Machines 3
1.2 The Cost of Artificial Intelligence (AI) 5
1.3 The Walls 6
1.3.1 Trends in Architectures 9
1.4 Reliability and Resilience of Neural Networks (NNs) 10
1.4.1 Safety Critical Systems 11
1.5 Faults in Hardware 12
1.6 Conclusions 13

2 Background **19**
2.1 Neural Networks 19
2.1.1 Feed Forward NN 20
2.1.2 Training Algorithm 22
2.1.3 Convolutional Neural Network 23
2.1.4 Popular Neural Architectures 25
2.2 Faults and Reliability 28
2.2.1 Fault models 28
2.2.2 Fault Injection 30
2.2.3 Functional Assessment through Fault Injection (FI) 30
2.2.4 Statistical Fault Injection 31
2.3 Systolic Array 32
2.3.1 A Matrix-Multiplier Systolic Array 33
2.3.2 Systolic Array Dataflow and advantages 34
2.4 Conclusions 36

3 Fault Injection on Systolic Array **39**
3.1 Motivation 39
3.2 Systolic Arrays in AI and their formalization 41
3.2.1 LoLif 41
3.2.2 Uniform Recurrent Equations 43

3.2.3 Regular Iterative Algorithms . . . 44
3.2.4 Matrix Multiplication . . . 44
3.2.5 Processing Element instantiation . . . 45
3.2.6 The Physical Parameters . . . 46
3.3 Flauers . . . 47
3.3.1 Fault injection mechanism . . . 47
3.3.2 Additional details . . . 49
3.3.3 Software considerations . . . 53
3.3.4 Projection Matrices Examples . . . 54
3.4 Effect of the Dataflow on Resilience . . . 59
3.4.1 Convolutional Neural Networks and Datasets . . . 60
3.4.2 Dataflows . . . 61
3.4.3 Fault Injection setup . . . 61
3.4.4 Fault Classification . . . 63
3.4.5 Results . . . 64
3.5 Conclusions and Future Perspectives . . . 67

4 Fault Injection Optimization **71**
4.1 Motivation . . . 71
4.2 Preliminaries . . . 73
4.3 Error resilience in DNNs . . . 74
4.3.1 Metric-based Test Samples Ranking Method . . . 75
4.3.2 Experimental Setup . . . 77
4.4 Selecting adequate metrics . . . 78
4.5 Hardware Independent Experiments . . . 89
4.6 Hardware-aware Experiments . . . 90
4.7 Conclusions . . . 93

5 Approximation **99**
5.1 Motivation . . . 99
5.2 Approximate Computing . . . 101
5.2.1 Metrics and Multipliers . . . 102
5.2.2 Functional approximation . . . 102
5.3 A Case Study using a Systolic Array . . . 102
5.3.1 Neural network . . . 102
5.3.2 Approximate multipliers . . . 103
5.3.3 Experimental setup . . . 104
5.3.4 Intrinsic NN robustness . . . 105
5.3.5 General resilience . . . 106
5.3.6 Correlation between injected bit and resilience . . . 106
5.3.7 Resilient approximate multiplier . . . 108
5.4 The Effect of Approximate Multipliers on NNs . . . 110
5.4.1 Neural Networks . . . 110
5.4.2 Approximate Multipliers . . . 110
5.4.3 Fault Injection . . . 113
5.4.4 General Resilience . . . 114
5.4.5 Layer-specific resilience . . . 114
5.4.6 Discussion . . . 114
5.5 Conclusions . . . 117

6 Fault Propagation Prevention through Monitors **121**
6.1 Preventing Fault Propagation . . . 121
6.2 Preliminaries . . . 122
6.3 Proposed Approach . . . 122
6.4 Experimental results . . . 123
6.5 Computational Costs . . . 126
6.6 Conclusions . . . 127

7 Conclusions and Future Perspectives **129**
7.1 Main Contributions . . . 129
7.2 Perspectives and future directions . . . 130

A Synthèse en Français **133**
A.1 Résumé (Abstract) . . . 133
A.2 Contexte et enjeux . . . 134
A.3 Contributions clés . . . 135
A.4 Perspectives futures . . . 135
A.5 Publications associées . . . 135
A.6 Résumé par Chapitre . . . 136
A.6.1 Chapitre 1 : Introduction . . . 136
A.6.2 Chapitre 2 : Contexte Théorique . . . 136
A.6.3 Chapitre 3 : Injection de Fautes dans les Systolic Arrays . . . 138
A.6.4 Chapitre 4 : Optimisation de l'Injection de Fautes . . . 139
A.6.5 Chapitre 5 : Approximation et Résilience . . . 140
A.6.6 Chapitre 6 : Prévention de la Propagation des Fautes via des Moniteurs . . . 140
A.6.7 Chapitre 7 : Conclusions et Perspectives . . . 141
A.7 Conclusion . . . 142

Abstract

It was 1943 when the idea of Neural Networks (NNs) was first introduced in the academic community as a computational paradigm. At that time, the pair composed of Warren McCulloch and Walter Pitts published a paper by the name "The Logical Calculus of the Ideas Immanent in Nervous Activity". In that paper, the authors formalize the concept of *nervous net* through a novel theory, stating that "to psychology, however defined, specification of the net would contribute all that could be achieved in that field", highlighting the hope of the authors of reproducing *general* Artificial Intelligence (AI), i.e. an artificial intelligence *comparable to human intelligence*. It wasn't until 2012 that this theoretical view of computation really grasped on. In fact, in 2012, a novel architecture called AlexNet was proposed for image classification. That was the year that renewed interest in NNs. Thirteen years later, in 2025, we have come a long way. Today, our systems are sophisticated enough to easily pass the Turing test. The advent of Large Language Models (LLMs) changed completely the scenario of AI.

New advancements in this technology are impressive, and it seems that this trend of improving AI is no way near to end. Nevertheless, these results come with a cost: not only the power required to run present systems is much more than any previous computational system of the past, but also it has been documented that bigger AI models have better capabilities, forecasting an even higher power requirements.

Today, we have the most powerful machines of all time. Yet, looking at the recent advancements it seems that soon we will saturate all the power we have. It is undeniable that most of these advancements are due to Moore's law, which states that the number of transistors in an integrated circuit doubles every two years. However, Moore's law is approaching its limits: quantum effects manifest at the present scale ($\sim$ 3nm) and manufacturing defects are becoming more relevant. On top of that, the potential improvements in AI seem to be limited by the computational limits due to the power and memory walls.

Although these issues have been known since early 2000s, the community moved toward novel architectures (with moderate success) to address the most common operation in modern AI: matrix multiplication.

To this extent, the most popular architecture is the Systolic Array (SA), which was introduced in the 60s, offers almost ten times the performance while using about half the power. The success of this architecture has driven many companies into proposing their own version. However, with a sufficiently large datacenter, a new challenge is arising: Silent Data Corruptions (SDCs). It has been reported by big companies like Intel, Google and Meta that quasi-random errors impact the computations of their servers. The problem becomes especially important when similar systems are introduced in safety-critical environments.

Given this context, this manuscript explores the problem of evaluating AI systems' reliability on four fronts:

- First of all, improving simulation time. Register Transfer Level (RTL) simulation is quite expensive,

and it would require a significant amount of time to perform the simulation of a complex system comprised of SA. For this reason, we propose a more efficient Fault Injection (FI) framework to study the reliability of AI when the SA is faulty and errors are introduced in the computation.

- Second, FI optimization. As FI is an expensive process, we propose a novel technique to reduce the workload of the FI through data selection: by taking only *challenging examples*, it is possible to obtain a pessimistic estimation of the resilience of the network. This methodology is applicable to any kind of FI, as it focuses on the dataset to be used for the FI campaign.
- Third, we study the effect of noise in the NNs through the employment of approximate multipliers in the network.
- Lastly, we developed a simple method to prevent the propagation of faults in Deep Neural Networks (DNNs).

List Of Publications

Journals

[1] A. Ruospo, S. Pappalardo, V. Turco, A. Bosio, and E. Sanchez, "Reliability of deep neural networks: Impact and open issues," *IEEE Design & Test*, vol. 42, no. 3, pp. 14–25, 2025. DOI: 10.1109/MDAT.2025.3544125

[2] C. Bolchini et al., "Benchmark suite for resilience assessment of deep learning models," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025. DOI: 10.1109/TCAD.2025.3578297

[3] N. Bellarmino, S. Barone, S. Pappalardo, A. Bosio, and R. Cantoro, "OpRA: Optimizing resiliency assessment for deep neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2025, ISSN: 1937-4151. DOI: 10.1109/TCAD.2025.3610062 Accessed: Oct. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11164517

[4] S. Pappalardo et al., "Saffira a framework for assessing the reliability of systolic-array dnn accelerators," *Journal of Circuits, Systems and Computers*, vol. 34, no. 18, p. 2 543 001, 2025. DOI: 10.1142/S0218126625430017 [Online]. Available: https://doi.org/10.1142/S0218126625430017

Conferences

[5] S. Pappalardo, A. Ruospo, I. O'Connor, B. Deveautour, E. Sanchez, and A. Bosio, "A fault injection framework for ai hardware accelerators," in *2023 IEEE 24th Latin American Test Symposium (LATS)*, 2023, pp. 1–6. DOI: 10.1109/LATS58125.2023.10154505

[6] S. Pappalardo, A. Ruospo, I. O'Connor, B. Deveautour, E. Sanchez, and A. Bosio, "Resilience-performance tradeoff analysis of a deep neural network accelerator," in *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 2023, pp. 181–186. DOI: 10.1109/DDECS57882.2023.10139704

[7] A. Piri et al., "Input-aware accuracy characterization for approximate circuits," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2023, pp. 179–182. DOI: 10.1109/DSN-W58399.2023.00050

[8] S. Pappalardo et al., "Investigating the effect of approximate multipliers on the resilience of a systolic array dnn accelerator," in *2023 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2023, pp. 1–6. DOI: 10.1109/DFT59622.2023.10313535

[9] M. Taheri et al., "Saffira: A framework for assessing the reliability of systolic-array-based dnn accelerators," in *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, 2024, pp. 19–24. DOI: 10.1109/DDECS60919.2024.10508925

[10] M. Traiola et al., "Approximate fault-tolerant neural network systems," in *2024 IEEE European Test Symposium (ETS)*, 2024, pp. 1–10. DOI: 10.1109/ETS61313.2024.10567290

[11] S. Pappalardo, B. Deveautour, and A. Bosio, "How systolic array dataflow impacts on the resilience to hardware faults," in *2024 IEEE International Conference on Design, Test and Technology of Integrated Systems (DTTIS)*, 2024, pp. 1–6. DOI: 10.1109/DTTIS62212.2024.10780346

[12] S. Pappalardo, A. Piri, A. Ruospo, B. Deveautour, E. Sanchez, and A. Bosio, "Reliable and efficient hardware for trustworthy deep neural networks," in *2024 IEEE International Conference on Design, Test and Technology of Integrated Systems (DTTIS)*, 2024, pp. 1–5. DOI: 10.1109/DTTIS62212.2024.10780183

[13] M. H. Ahmadilivani et al., "Special session: Reliability assessment recipes for dnn accelerators," in *2024 IEEE 42nd VLSI Test Symposium (VTS)*, 2024, pp. 1–11. DOI: 10.1109/VTS60656.2024.10538707

[14] N. Cherezova et al., "Heterogeneous approximation of dnn hw accelerators based on channels vulnerability," in *2024 IFIP/IEEE 32nd International Conference on Very Large Scale Integration (VLSI-SoC)*, 2024, pp. 1–4. DOI: 10.1109/VLSI-SoC62099.2024.10767798

[15] A. Rafiq, A. Bosio, S. Pappalardo, and M. Jenihhin, "Axenmult: Design of an efficient and reliable approximate encoding-based multiplier," in *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, vol. 1, 2025, pp. 1–6. DOI: 10.1109/ISVLSI65124.2025.11130248

How to Read this Manuscript

In order to make import concepts more visible and easily findable, this manuscript makes use of marginnotes: paragraphs on the outer margin of the page. Like the one on the side of this page, marginnotes are used mainly to highlight definitions, to make them pop up, with respect of the main body.

Definition: marginnote.

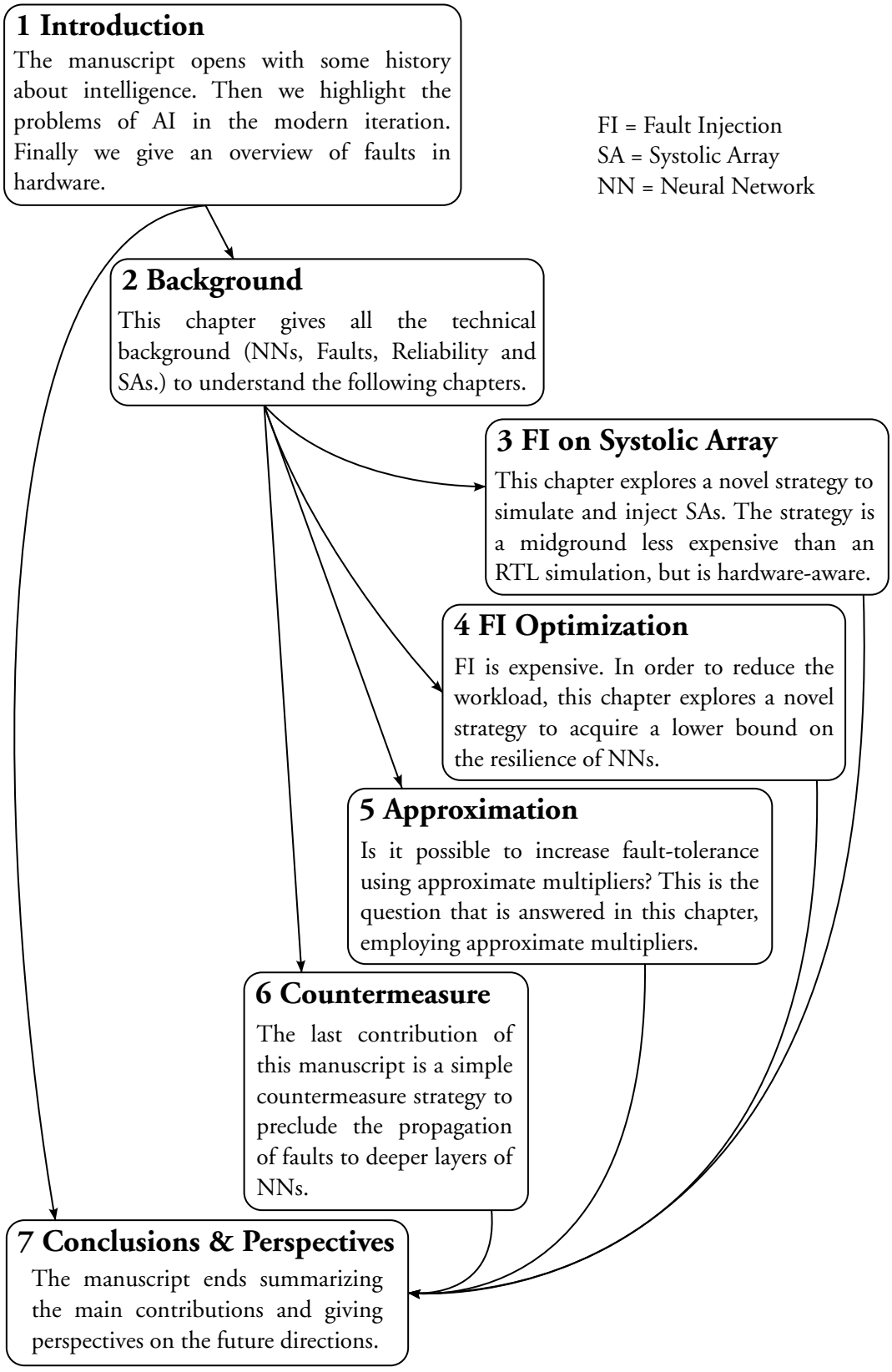
On top of that, sidenotes are used throughout the text[1].

[1] The difference between a sidenote and a marginnote is that sidenotes are numbered and referred to a specific sentence in the main body.

Furthermore, all the vectors are indicated with lowercase bold text, e.g. $\mathbf{x}$, while matrices are indicated with uppercase bold text, e.g. $\mathbf{A}$. All the other values are scalars, e.g. w.

What is in this manuscript

The manuscript is organized as in the figure that follows. We suggest reading the first two chapters completely, as they provide the background to read the rest of the manuscript. The central chapters are independent of one another. Finally, the manuscript ends with conclusions on the work of all the chapters, giving perspectives and future directions.


1 Introduction
The manuscript opens with some history about intelligence. Then we highlight the problems of AI in the modern iteration. Finally we give an overview of faults in hardware.
FI = Fault Injection
SA = Systolic Array
NN = Neural Network
2 Background
This chapter gives all the technical background (NNs, Faults, Reliability and SAs.) to understand the following chapters.
3 FI on Systolic Array
This chapter explores a novel strategy to simulate and inject SAs. The strategy is a midground less expensive than an RTL simulation, but is hardware-aware.
4 FI Optimization
FI is expensive. In order to reduce the workload, this chapter explores a novel strategy to acquire a lower bound on the resilience of NNs.
5 Approximation
Is it possible to increase fault-tolerance using approximate multipliers? This is the question that is answered in this chapter, employing approximate multipliers.
6 Countermeasure
The last contribution of this manuscript is a simple countermeasure strategy to preclude the propagation of faults to deeper layers of NNs.
7 Conclusions & Perspectives
The manuscript ends summarizing the main contributions and giving perspectives on the future directions.

CHAPTER 1

Introduction

Let us introduce this manuscript by discussing the origin of the words *artificial intelligence*. The word "artificial", today has the meaning of "made by man, contrived by skill and labor". Nevertheless, the meaning has shifted around the 15th century [1]. The etymology of "artificial" is the Latin *"artificialis"*, a combination between *"artificium"* and the suffix "-alis" [2]. The suffix is added to create an adjective. Not very interesting. The juicy part is the other half: *artificium* is translated as "art", "craft", "skill", "talent" or "craftsmanship". The original meaning of artificialis designated someone (or something) that possess "art", "craft", "skill" or "talent".

"Intelligence" comes from the Latin *"intelligentia"*, composed by *"intelligens"* and the suffix *"-ia"*. This is less interesting, than the other word, as the meaning is essentially the same: an abstract noun with the meaning of "understanding", "discerning".

The original meaning of *Artificial Intelligence* designate an *understanding of the art*. And is central to this manuscript, although we will mainly talk about something slightly different but related:

The art of understanding.

Artificial Intelligence, as we mean it today, has the same meaning: the art of understanding. More explicitly, the art of teaching machines to understand. Or more ambitiously: the art of creating understanding by essentially replicating the function of the human brain. This is one goal of modern research, and has been around for a while.

Unfortunately, the models we have today are not capable of *understanding*, but rather they are complex statistical models which are very good at guessing what word (or token, shall I say) comes next. Yet, as we will see, this technology is impressive. But let us not get ahead of ourselves.

This chapter serves as an introduction for the rest of the manuscript, and gives sufficient background to understand the context we live in and the reason of this research.

Today, Artificial Intelligence (AI) models are evolving rapidly, achieving unexpected levels of accuracy and human-like capabilities. For sure, people kept trying, but real progress did not come until the modern days. And, while it is true that current machines do not think, they surely performed amazingly in any test we give them (e.g. the Turing test) and keep getting better [3].

1.1 (Artificial) Intelligence

The concept of intelligence is somewhat elusive. As of today, 29th May 2025, the English Wikipedia page for "Intelligence" reads:

> Intelligence has been defined in many ways: the capacity for abstraction, logic understanding, self-awareness, learning, emotional knowledge, reasoning, planning, creativity, critical thinking and problem-solving. It can be described as the ability to perceive or infer information and to retain it as knowledge to be applied to adaptive behaviors within an environment or context.

The issue of defining *intelligence* is one that has taunted humans for quite some time now. The idea of *measuring* intelligence dates back to the early 20th century. Back in 1904, the French Ministry for Public Instruction asked a psychologist by the name of Alfred Binet to design a procedure to identify children with special needs [4]. The objective was to ensure these children received the attention they required. Although a clear definition of intelligence was not given, Binet measured a variety of cognitive abilities like memory, reasoning and comprehension. A survey about *intelligence testing* from 1980 [5], reports the various attempts at measuring intelligence. In the article, there is an interesting section which reports the various theories on which the tests are based. Surprisingly, there is no strong theoretical bases on this concept. Rather, the authors say that (i) these tests are driven by pragmatism, rather than theory and (ii) there is different view on the concept itself, some consider it unidimensional, while some speak about a multidimensional construct.

In the following years, a book was published by the psychologist Richard J. Herrnstein and the political scientist Charles Murray [6]. Released in 1994, the book sparked heated discussions. In fact, its content might actually be quite controversial. To give the reader an idea, here is how the second chapter of part I begins:

> People in different jobs have different average IQs. Lawyers, for example, have higher IQs on the average than bus drivers. Whether they must have higher IQs than bus drivers is a topic we take up in detail in the next chapter. Here we start by noting simply that people from different ranges on the IQ scale end up in different jobs.

Given the controversial nature of a work like this, in 1995 a task-force was put together by the Board of Scientific Affairs of the American Psychological Association in order to clarify all the scientific misunderstanding generated by that text [7]. For the scope of this manuscript, this paper is probably the most recent and comprehensive overview on the topic of intelligence.

The authors clearly explain that «concepts of intelligence are attempts to clarify and organize this complex set of phenomena», referring to many abilities [7] like:

- understand complex ideas,
- adapt to environments,
- learn from experience.

The authors also highlight how, although these abilities are substantial, they are never entirely consistent: intellectual performance vary on different occasions, in different domains, as judged by different criteria. Another interesting finding, and we finally come to the topic of *artificial* intelligence, is the recognition that individuals possess different types of intelligence. In the paper, three main theories are cited, but all of them agree on one aspect: there are different types of intelligence. I found the distinction between analytical problems and practical problems particularly intriguing I will not waste your time explaining the difference, but I will report an example: «women shoppers in California who had no difficulty in comparing product values at the supermarket were unable to carry out the same mathematical operations in paper and pencil tests»[8]. Other authors speak of emotional intelligence [9], spatial intelligence [10], and more.

Recognizing intelligence as a multifaceted problem is what eventually gave rise to modern AI. To understand that, let us introduce some more history. The way we achieve AI is by breaking down complex tasks into smaller, more manageable tasks that AI can actually solve. By the time all the individual tasks are solved independently, the overall complex problem is solved.

1.1.1 Intelligent Machines

The fascination with somewhat human-like machines dates back a long time, when the first automatic machines became reality. The earliest reference that I could find on the topic is the play R.U.R. [11], written in 1920.

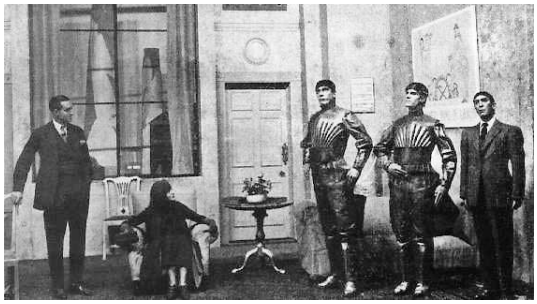


Figure 1.1: A scene from the play, showing three robots

In the play, the main character Rossum invents a substance similar to organic matter, which is later used to assemble robots. With advancements in technology, robots become cheap and essential for industry, replacing figures like secretaries, field workers, miners, and so on. Eventually the robots revolt and destroy the human race. This play is also commonly known as the earliest reference to the word "robot"; nevertheless, the way Capek describes his robots differs from what we would consider today as a robot. Capek speaks of artificial biological organisms that may be mistaken for humans. On top of that, they are biological beings, assembled in factories, as opposed to grown or born.

Although Capek's robots are not "machines" in the strict sense[2], the essence is the same: a mechanical person that performs mechanical tasks. The meaning of the word robot itself in 1923 was "a person whose work or activities are entirely mechanical" [12]. On top of that, the concept of *intelligent machines* was already well established back then, as demonstrated by the fact that these *robots* revolted and destroyed the humans, a clear sign of free will.

[2] The Merriam-Webster dictionary defines machine as "a mechanically, electrically, or electronically operated device performing a task".

Fast-forward to the '40s, and the first attempts at modeling *intelligence* were conducted. At that time, the pair composed of Warren McCulloch and Walter Pitts published a paper by the name "The Logical Calculus of the Ideas Immanent in Nervous Activity" [13]. In that paper, the authors formalize the concept of *nervous net* through a novel theory, stating that "to psychology, however defined, specification of the net would contribute all that could be achieved in that field", underlying how, in the vision of the authors, a working model for the nervous system would revolutionize the field of neuroscience. For the first time, in 1943, the idea of Neural Networks (NNs) was introduced in the academic community as a computational paradigm, effectively trying to emulate the way organic brains work.

Later, in 1958, the novel concept of perceptrons was introduced by Rosenblatt, showing how an electronic device, modeled after biological neurons, could *learn*. Rosenblatt described what we would recognize today as a typical training of modern NN. In his technical report [14], he describes a device that has three parts: (i) a sensory system, which senses (measures) physical quantities, such as the intensity of light; (ii) an association system, which receives its inputs from the sensory system and produces an output for the response system, based on a threshold θ; and (iii) a response system, consisting «typically of a relatively small number of units», which prints or displays an output signal. Additionally, this system needs to guarantee that only one response is triggered at a time.

Rosenblatt discovered that his device could associate specific responses to specific stimuli consistently, with a probability that he described as "better-than-chance". He outlines a learning procedure in which the device is forced to give a specific response when exposed to a specific stimulus. In his words [14]:

> For example, we might require the perceptron to learn the concepts "square" and "circle", and to turn on Signal Light 1 for "square", and Signal Light 2 for "circle". We would then proceed to show the system a large set of squares of different sizes, in different locations, while holding Light No. 1 on, thus "forcing" the response. We would then show a similar set of circles, while holding Light No. 2 on. If we then show the perceptron any square or

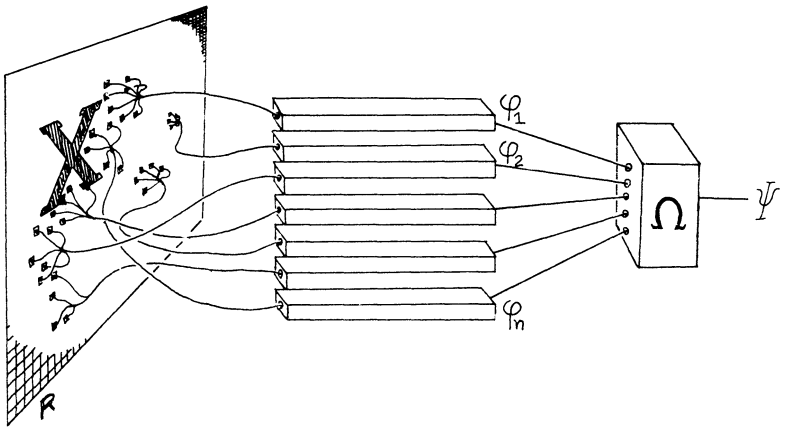


Figure 1.2: Perceptron as described by Rosenblatt. Taken from [15].

> any circle, we would expect it to turn on the appropriate light, with a high probability of being correct.

Rosenblatt later published a paper [16] in which he mentions many concepts that are trendy today. For example, he talks about reinforcement learning, zero-shot learning and feature learning, predicting overfitting and network capacity. Research about perceptrons went ahead, but it became clear that perceptrons had various limitations [15].

In the following years, MultiLayer Perceptrons (MLP) became popular. MLPs are also known as "feed-forward neural networks". The idea is an extension of Rosenblatt's original idea, by means of introducing multiple layers. While a perceptron has a single hidden layer, referred to as the *association system*, MLPs have multiple association systems.

Research stalled for almost 20 years, when finally, in 1989, Hornik discovered that MLPs are universal approximators [17]. Although the paper is very technical, it clearly shows that a feed-forward NN can approximate virtually any function to any desired degree of accuracy. The paper actually addresses a more general type of network, defined as $\Sigma\Pi$, which also have a productory component, but this is outside the scope of this text. Nevertheless, we might see a surge of this concept in the following years.

To reach modern NN, we are missing a final piece of technology: back-propagation learning. Although the idea was first introduced in 1962 by Rosenblatt [18] himself, the concept was not developed for MLP until 1967 [19]. The introduction of back-propagation is the true realization of Machine Learning (ML). With this new procedure, it is finally possible to perpetrate automatic learning. The idea of back-propagation is based on computing the error committed by an MLP (or more generally, by a parametric function) and then adjusting its parameters based on the derivative. We will see these concepts in detail in the next chapter, section 2.1. The history of neural networks is very interesting, but this is the extent of the detail I will provide within this text. For those who are interested, reference [20] is a good starting point.

In 2025, we have come a long way, while maintaining the basic principle largely unchanged. Today, our systems are sophisticated enough to easily pass the Turing test [21]. The advent of Large Language Models (LLMs) has completely transformed the AI landscape. Today we have the most powerful AI technology ever seen on the planet: we can talk to our computers [22], cars drive autonomously [23], [24], robots can be controlled for general purposes [25], automatic translators are comparable to interpreters [26], professional go players [27].

Today, we define AI as a general property of a digital system. Figure 1.3 shows the hierarchy of AI.

NNs are part of ML, as they operate in a way that mimics "learning". The innermost category, Deep Learning, refers to the practice of *going deeper* in the context of NNs. This historical aspect will be discussed in subsection 2.1.4. Just as a side note: when we speak about AI today (especially enthusiastic companies) we actually refer to deep NNs, as this is the basic technology behind modern LLMs.

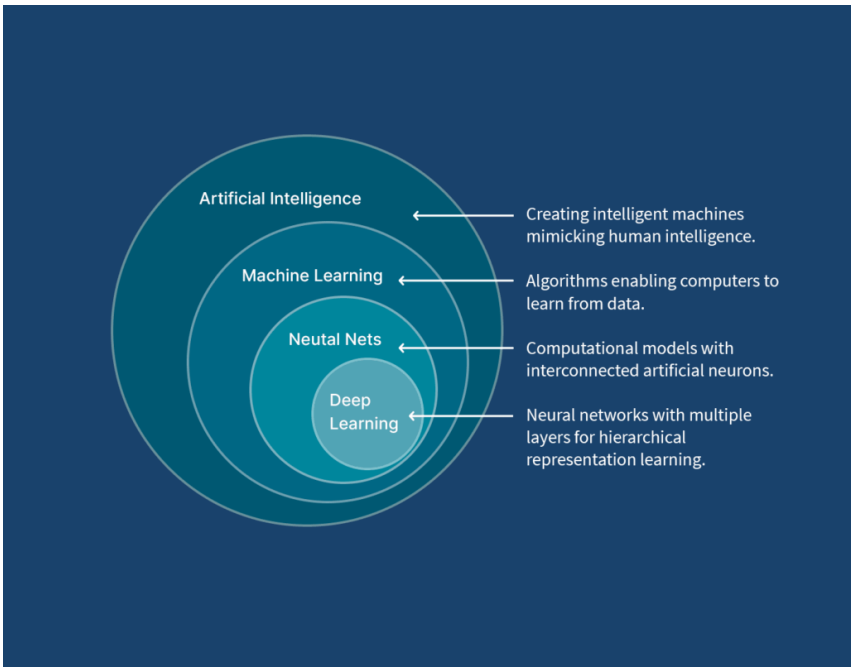


Figure 1.3: AI hierarchy. Picture taken from [28].

1.2 The Cost of AI

There is a catch with this technology. The cost for this so called AI revolution is paid in terms of computational requirements, training costs, and power budget. A recent paper [29] computes the cost of training frontier AI models. The authors estimate the "amortized hardware cost plus energy" as a measure of the total cost of a given model. Since the advent of Moore's law (which we will talk about later in this chapter), the cost of hardware has been going down every year: even though the cost of manufacturing increased with more advanced technology nodes, the same technology nodes (being smaller) allowed to cram exponentially more transistor in the same area, resulting in a net cost decrease [30]. As result, the hardware produced today is fundamentally more capable than hardware produced a year ago. This trend affects the price of acquired hardware: its value depreciates just because it was produced earlier, while technology keeps advancing. The authors consider such depreciation and compute the amortized cost of hardware as the portion of value that the hardware loses during training, based on the growth rate of the price-performance ratio [31]. On top of that, they also consider the energy used for training. Figure 1.4 shows the amortized cost of hardware plus energy for frontier models. The figure illustrates that the cost increases by a factor of 2.4 every year. The authors estimate that by 2027, the most expensive publicly available model will cost about 1 billion (10^9) $. Additionally, the authors estimate that the cost of acquisition of the hardware is almost 50% of the amortized cost. Interestingly, the upfront acquisition cost is also rising at a similar rate, by 2.5 times per year, although such costs are one to two orders of

magnitude higher.

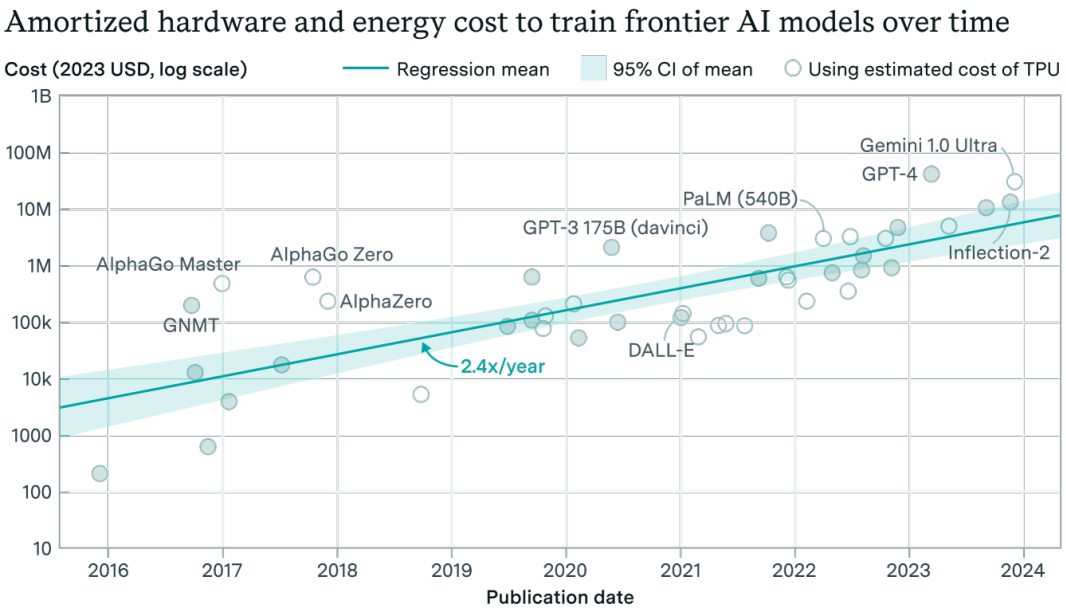


Figure 1.4: Amortized hardware cost plus energy for the final training run of frontier models. Taken from [29].

In addition to the overall cost of AI, we cannot forget to mention the *computational cost* of the training itself. A post published by EPOCH AI in 2024 [32] shows that computational demands has grown by a factor of about 4.2 every year in the past 15 years. We can measure the computational demands in terms of FLoating-points Operations Per Second (FLOPS) required by a given model to perform an inference. Figure 1.5 shows the general trends of the computational demands of ML.

It is undeniable that the performance achieved by these models is impressive [3]. The last frontier of models is called LLM and has been shown to exhibit some behavior similar to problem-solving. These models are getting better and better at reasoning tasks. Nevertheless, the current trend cannot continue indefinitely.

1.3 The Walls

To this day, we have the most powerful machines ever created. A little quirk arising from this fact is that we can now solve problems that, 50 years ago, would have been unthinkable due to the computational power required For example, array languages like APL [33], are implementable thanks to modern technology. Another example is hyper-dimensional computing [34], which employs high-dimensional vectors to generalize concepts. Nevertheless, despite our very powerful machines, modern AI still needs amounts of computational power that we cannot yet provide.

These amazing features require lots of computations. In 1965 Gordon Moore, co-founder of Intel, predicted that the number of transistors in integrated circuits would double every two years [35]. For the past 60 years, that claim proved true [36]. Figure 1.6 shows the general trend of transistor count in popular chips from 1970 until 2020. There is a very clear exponential trend, highlighting the resilience of Moore's law. Nevertheless, different challenges had to be overcome to reach this result, as it proved

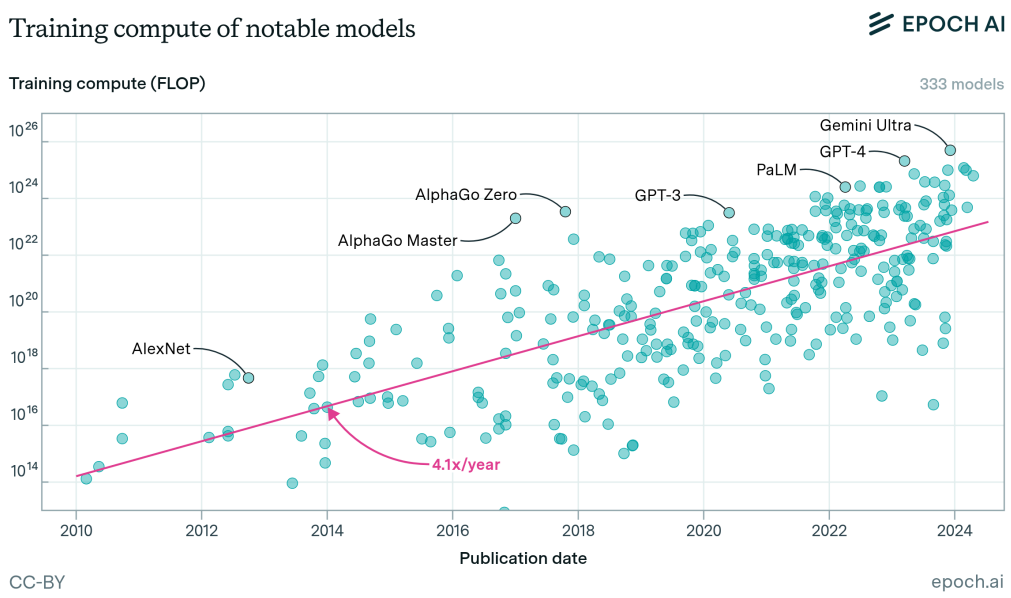


Figure 1.5: Computational demands of ML models. The ordinate shows the FLOPS required to run the model. The abscissa shows the publication year of the model. Figure taken from [32].

challenging (sometimes impossible) to just plainly scale down the same technology. As Moore himself said: "no exponential can continue forever" [37], and we delayed it for quite some time.

On one side, there are the device technology walls which are related to efficiency: as transistors scaled down in size, the heat dissipated scaled down with it due to Dennard's scaling law [38]. Ideally, we would need some device that is more efficient than transistors. However, new phenomena arose, which we were ignorant about at first, which made us rethink the way transistors are manufactured. Scaling down the physical size of the transistors, a new current, called leakage current [39], started flowing in the transistor, despite it being powered off. Furthermore, the smaller the transistors, the more power is wasted through leakage [40]. The FLOPS per unit of energy has been essentially the same since the first transistor was born. For these reasons, the community came up with new designs to reduce the problem.

On the other side, there are the architecture walls: memory, power and instruction level parallelism. Although all of them are interesting and worth exploring [41], [42], the memory wall is probably the most relevant when focusing on AI. In a paper from 2024 this problem is highlighted very clearly [43], with respect to the current AI technology. The paper shows three main points, which we quickly summarize:

Growing rate of bandwidth and FLOPS - Figure 1.7 compares the growth rates of memory bandwidth and FLOPS in hardware. While the latter has been increasing at a rate of about 3x/2 years, improvements in the bandwidth are much smaller, at about 1.5x/2 years.

The effect is visible - The discrepancy is measurable. The authors conducted experiments with different models, showing that it is possible to have smaller latency for bigger models if those models perform fewer accesses to memory.

Rethinking AI - Many of the recent advancements come from scaling up the same technology, without really improving the algorithms or the architectures substantially. This is true for both neural architectures and accelerators for AI.

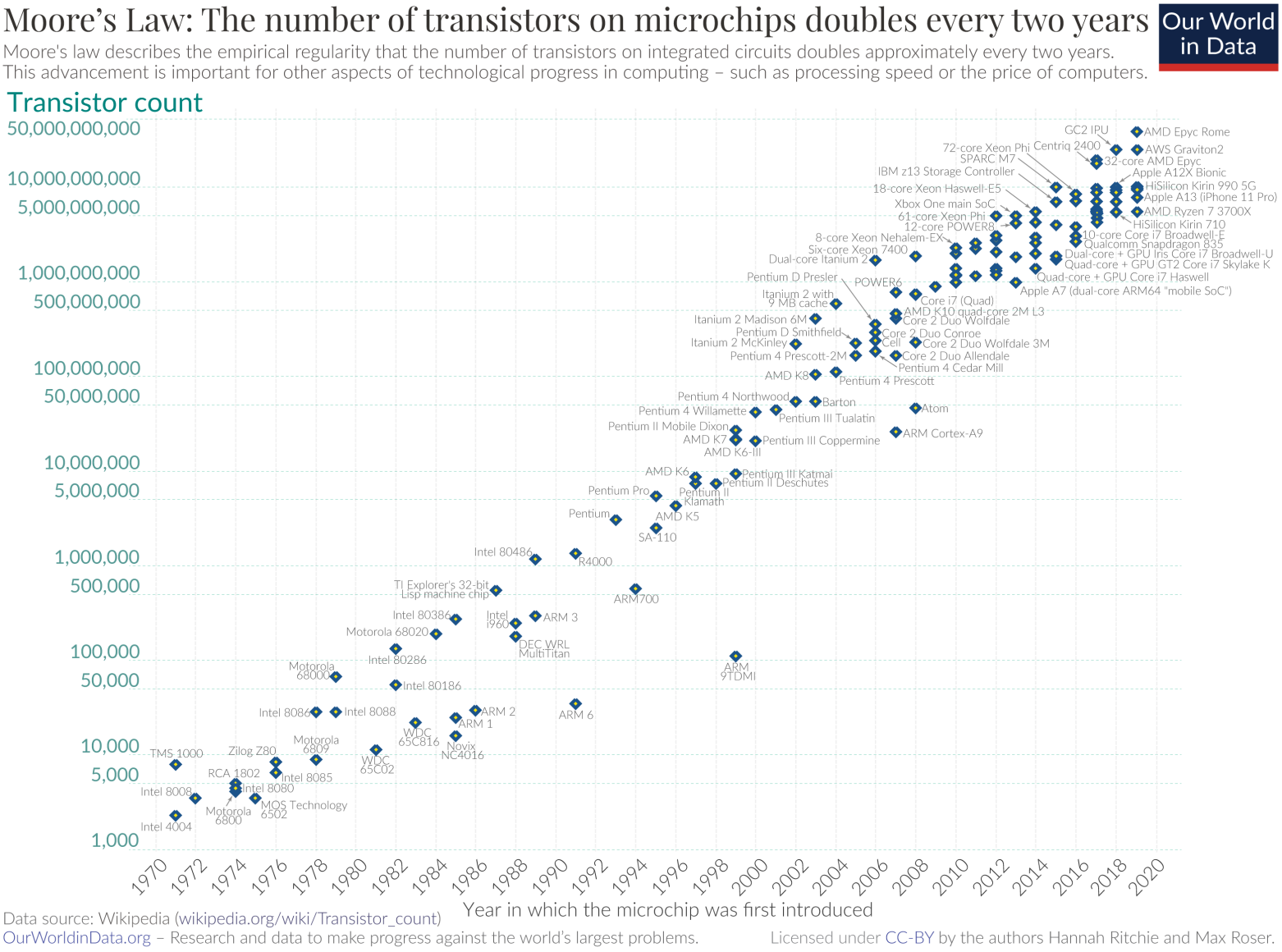


Figure 1.6: Moore's Law predicting blah blah blah. Picture taken from [36].

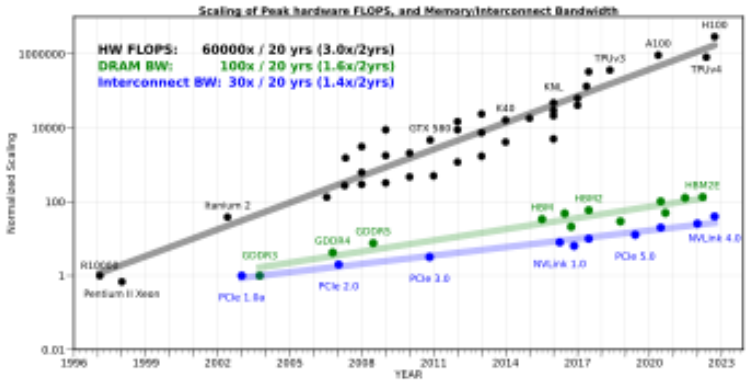


Figure 1.7: Growing rate of bandwidth of interconnect and memory, compared to growing rate of FLOPS of hardware. Figure taken from [43].

1.3.1 Trends in Architectures

As said before, Moore's law continued for so long because of many factors at play. In this section we will discuss briefly the major reasons for this trend, while putting an accent on architectural developments.

Since CPUs are very efficient but not very fast (at least for bulk data), many new architectures started to appear. Today the most popular is surely the GPU and it dominates all kind of fields, from professional fields to academics. Nevertheless, a GPU is still a superscalar processor and has limitations due to the memory wall. To address this problem and given that the most expensive operation to perform is matrix-multiplication [44] another architecture became more and more popular: the Systolic Array (SA). Matrix multiplication is so relevant because it is used for the most popular NNs in use today. LLMs implement the attention mechanisms [45], which is constructed with it; the fully-connected layers (used to build MLP) are also called linear, as they actually perform a linear transformation, i.e. another matrix multiplication; finally Convolutional Neural Networks (CNNs), especially the most performant as we will see later in subsection 2.1.4, can be implemented via matrix multiplication.

At the beginning of the '70s, the architectures were simple and had only a few capabilities. With Dennard's Scaling, the processors became smaller and smaller. There were also a lot of Domain Specific Architectures (DSA). Starting around 1985 pipelining became popular, to the point that also embedded space application used it [46]. Pipelining is a technique that allows to process multiple instructions at the same time, overlapping their execution: to execute each instruction there are a few actions to take which can be parallelized. Usually, a pipelined processor has five stages:

instruction fetch the next instruction is fetched from the memory and copied in the CPU;

instruction decode this action is needed to understand (i) which registers will be read written during execution and (ii) verify whether any branch has to be taken;

execution during this action, the actual computation is done, for example by using the ALU;

memory access this action allows accessing the main memory when the executing instruction is a load or a store;

writeback this action has the responsibility to update the register file.

This technique is useful because, at regime when the pipeline is full, 5 instructions are executed all together, without the need to wait for the previous instruction to finish, which would take additional time [47].

Interestingly, the improvements of performances came in the form of frequency increase, since the power needed by the transistors was roughly proportional to the area. This led some manufacturers to push in this direction for several years.

Around the years 2004 Dennard's scaling ended. This news pushed the architecture in different directions. One of the main novelties introduced around that time, was speculative execution. A paradigm that exploits instruction-level parallelism. The basic idea is to execute instructions out of order, in such a way to reduce stalls of the pipeline, while taking care of potential data hazards. On a related note, other novelties were introduced: Data-level and Task-level parallelism. [48] These two variables were essential to push the computing capabilities forward.

Data-level parallelism Data-level parallelism is essential achieved through SIMD: Sing Instruction Multiple Data. Exploiting a single instruction to perform the same operation on more than one datum. As an example, we might think of multiplying a vector by a scalar: all the components of the vector have to be scaled by the same amount and there is no dependence among those operations. Pushing on this concept, we arrive to Graphical Processing Units (GPUs). Although the story of traditional CPUs is unrelated to that of GPUs, the things converge, as graphics is a strong example of parallelization[3].

[3] The GPU was born to do one thing very well: graphic processing. Due to its nature, as virtually all operators are applied to singular pixels or subset of pixels at a time, parallelization is a must on graphic processing.

Task-level parallelism Another way to perform parallelization is inherent to unrelated tasks being executed on the same hardware. Making this observation, multiprocessing and multi-threading were born. The former was firstly implemented by Intel with their Core 2 Duo. The CPU had two processing units which could operate independently, gaining high processing speeds.

Although all of these improvements made computer very much faster, this is not enough for modern NNs. As explained above, the amount of computations to be executed is so vast, that any classical architecture could not perform efficiently enough. The best option, as of today, is using GPUs, as they offer huge performance gains, due to their regular structure. Yet, GPUs are power-hungry and are still limited by the core's capabilities: GPUs have less capable cores (although similar in structure) than Central Processing Unit (CPU)'s cores. For this reason, GPUs are bottlenecked by the memory bandwidth. To this extent, today we see the rebirth of DSAs: simple architectures built to solve a specific problem, as opposed to a general architecture for every single imaginable task. On this note, the SA architecture has become extremely popular. Many companies have invested in the development of such architectures. Notably, Google made their own Tensor Processing Unit (TPU) starting already in 2017 [49]. NVIDIA, the same manufacturer of the most popular GPUs in the planet, developed a similar architecture they called NVIDIA's Deep Learning Accelerator (NVDLA) [50]. The problem is explained in [49]: Google noticed that people using voice recognition would double in a matter of a few years. As said, the cost of voice recognition is not negligible (it is implemented using a NN). They argued that, if this was true, then they needed to double the computational demand of their datacenters: a very expensive plan. To solve the issue, they designed a coprocessor which was able to perform up to 92'000 billion operations a second (92 Tera-OPS) [51].

SAs are usually composed by a grid of Processing Elements (PEs), where each PE performs a Multiply and Accumulate (MAC) operation, meanwhile forwarding the data to the neighbors PEs. These PE grids are able to process a matrix multiplication with $O(n \times m)$ time complexity (supposing a matrix multiplication between two matrices with sizes $n \times n$ and $m \times m$). Due to their low energy consumption requirements, they can be used in embedded systems [52].

1.4 Reliability and Resilience of NNs

Until now, we have discussed AI. As explained previously the general term of AI includes all kinds of computer algorithms. In reality, today, 99% of the "AI talk" is related to NNs, which is the driving force

of AI. For this reason, the rest of this manuscript is written using only the concept of NN. Even in the occasion of encountering AI, it will be meant as NN.

In this section, we will briefly discuss the problem of Reliability, what it is, what is resilience and why it is important.

1.4.1 Safety Critical Systems

Safety Critical Systems are:

> Those systems whose failure could result in loss of life, significant property damage, or damage to the environment. There are many well known examples in application areas such as medical devices, aircraft flight control, weapons, and nuclear systems [53].

Deep Neural Networks (DNNs) have revolutionized the field of artificial intelligence and machine learning, due to their capacity to learn from vast amounts of data, making very complex predictions. Nowadays, the industry is employing them in many different fields including safety-critical ones, such as autonomous driving systems [52].

For example, [54] used a neural network to determine the flight regime of an aircraft using temporal segmentation, achieving more than 90% accuracy. Another example can be found in [52], in which the authors proposed a novel approach for lane detection in the context of autonomous drive.

In recent years, the possibility of using NNs in such system became very attractive, as the potential unlocked became endless. Most notably, autonomous driving received a lot of attention in recent years. Experimentation with autonomous driving has been going on for a while. In Stanford, the *Stanford Self-Driving Car Team* was founded already in 2004. The team went on until it was renamed Waymo in 2016 and became a proper company. Today Waymo has a robotaxi service active in the USA. Already in 2013 the UK government approved an experimental program to perform experimentation with self-driving cars, aiming to reduce traffic on the road [55]. In the same year the CEO of Tesla, an electrical vehicles company, announced they would work on autonomous driving [56]. Regulations arrived soon after.

Today self-driving cars drive hundreds of passengers around. The problem with these systems is safety: many accidents were reported involving fatalities. Given that autonomous driving is viewed as *the future*, autopilot crashes received great attention. For example, regarding Tesla, there are both a Wikipedia page [57] and even a dedicated website [58]. Waymo published a paper [59] that contains crash data, including every minor contact, "what if"s simulations and collision typology and severity.

Similarly, the aviation community has interest in NN and AI in general. Reference [60] shows, in their *related works* section, at least three domains that can benefit from the advent of AI, e.g. Air Traffic Management and Aviation Safety.

In the summer of this year (2025) the future of *life institute* published the annual *AI Safety Index* [61]. They analyzed the risk associated with modern AI. Their analysis is extensive and evaluates the models (and the companies behind them) on 6 axes. Although their analysis is broad, it gives us an idea of the context of today's world. Very briefly, the report states that current companies and models are not ready to be given excessive power.

All these domains have pretty much the same problem: dependability and safety. To this extent, we could define five main concepts all in view of NN:

Dependability As a paper from 2001 states: «Dependability of a computing system is the ability to deliver service that can justifiably be trusted. »[62]. In our context, dependability has to be intended as a general characteristic of a NN working in a safety-critical environment, thus including the other concepts.

Reliability Reliability is defined as the continuity of correct service. In our case, that means that we need to ensure that our NN correctly performs its task, whether it is classification, segmentation or detection (or whatever else) at any given moment in time [62].

Safety Safety is the absence of catastrophic consequences. Generally speaking, this requirement does not belong to NN itself, but rather it is part of the system in which NN is implemented, as there should be some kind of failsafe. Nevertheless, in the context of this work, we talk about safety as if the decision of the NN has a direct impact on the system without considering failsafes [62].

Resilience Resilience is defined in many ways depending on the author [63]. Given this ephemeral definition, we give to ourselves the liberty of defining it once again. We define resilience as the ability to withstand noise without breaking. The meaning of "breaking" is tied to the specific task as e.g. "breaking" would be misclassifying for a classification task.

Robustness Robustness is, in the ML community, the capability of withstanding adversarial attacks. It is, in a way, similar to resilience, with the difference that in this case the noise is carefully crafted to cause troubles [64].

These are the attributes that are mandatory for deploying NNs in safety-critical systems. Nevertheless, enforcing them is not a simple task: modern NNs are often treated as black-box systems, as we cannot really explain why some decisions are taken rather than others[4][68]. To this extent, one of the objectives of this manuscript is to give a methodology to assess these attributes, while treating the models as black-boxes. To be more precise, we will study these attributes at hardware level: the efficiency of novel architectures like SAs makes the execution of NNs for both edge and cloud applications; nevertheless, silicon is known to malfunction due to hardware faults, making the applications not dependable. In this regard, the next section gives an overview of the meaning of hardware faults.

[4] It would be unfair to just state that "we cannot explain why some decisions are taker rather than others" at leave it there. Although it is true that AI explainability is an unresolved problem today, much progress has been done and is being done [65], [66], [67].

1.5 Faults in Hardware

A fault manifest itself as an imperfection in the behavior of a circuit. Faults can originate in different ways. For example, they can be part of the manufacturing process: integrated circuits manufacturing is a very delicate operation, even particle of dust can affect the functionality. Although most of the faults are caught during testing, there is a minority which manages to make it through and reach an end-user application. Fault might be also caused by aging: stress on the circuit exaggerates the small imperfections which eventually lead to misbehavior. Another common cause of faults lies on radiations. Cosmic radiation travel in space and eventually reach earth. Oftentimes this high-energy particles clash with the circuit causing transient faults. In the following subsection we briefly explain the three categories just mentioned.

Manufacturing Defects Manufacturing defects originate during the fabrication or assembly phases of electronic components. Such defects may remain dormant until activated by operational stress or environmental conditions, leading to premature failures. Common causes include contamination, improper material deposition, lithography errors, or mechanical stress during packaging [69]. For example, we could have process variability: inconsistencies in doping, etching, or oxidation steps, resulting in non-uniform transistor performance. Another example could be contamination: Particulate or chemical residues (e.g., metallic ions) that alter electrical properties or create short circuits.

Aging Related Faults Some defects in the material are due to aging phenomena. For example, Bias temperature instability is the phenomenon for which the threshold voltage of the device increases (in absolute value) causing mis-behaviors that manifest as faults [70]. Another example is electromigration: atomic displacement in metal interconnects due to high current density, leading to opens or shorts [71]. Aging faults develop gradually due to material degradation under prolonged operational or environmental stress.

Radiation-Induced Faults Radiation-induced faults occur when energetic particles (e.g., cosmic rays, alpha particles, or neutrons) interact with electronic materials, disrupting normal operation. These effects are critical in aerospace, nuclear, and high-altitude applications but also affect terrestrial systems at advanced technology nodes (i.e. when the implemented technology is smaller than 28nm) [72]. Although these kinds of faults are most common in space, their effect is visible also on earth in large scale [73].

Definition: Defect, Fault, Error and Failure

A *Defect* represents any physical imperfection or external perturbation that might compromise the correct behavior of the system and can exhibit while subjected to anomalies. A *Fault* is a representation of a defect at the abstracted function level. An *Error* is a wrong output signal produced by a defective system. The error can be intended in both a physical (when it affects an electrical signal) and abstract (when it propagates to higher levels like the software) way. If the error is not corrected, or it affects an important part of the computation, then it becomes a *Failure*.

1.6 Conclusions

In this chapter we explored the etymological analysis of artificial and intelligence, not only clarifying their historical usage but also establishing a foundational understanding of artificial intelligence as a discipline rooted in human ingenuity and technical ambition. The exploration of these linguistic origins reveals more than just the evolution of words; it underscores the profound connection between human creativity and the concept of intelligence. By understanding that "artificial" once denoted skill and craftsmanship, we recognize that artificial intelligence is not merely a product of modern technology but an extension of humanity's enduring quest to replicate and amplify its own capabilities. This conception drove the development of AI through the centuries, to reach us in its modern form. Other than the philosophical perspective of having human-like like intelligence, we highlighted the real cost needed to sustain AI. We have seen that recent developments are pushing what is physically feasible. We are at the end of Moore's Law and for this reason new architectures like the SA have been developed, to (hopefully) succumb on the computational requirements of AI. Finally, we have anticipated the concepts of reliability and faults in the hardware, underscoring how critical it is to produce safe systems, i.e. systems that, while failing, do not threaten human lives.

The next chapter will give the necessary technical background to understand the core chapters of the manuscript.

References

[1] "Search 'artificial' on etymonline," etymonline, Accessed: Sep. 11, 2025. [Online]. Available: https://www.etymonline.com/search?q=artificial

[2] *Artificialis*, in *Wiktionary, the free dictionary*, Page Version ID: 78738452, Apr. 2, 2024. Accessed: Sep. 11, 2025. [Online]. Available: https://en.wiktionary.org/w/index.php?title=artificialis&oldid=78738452

[3] N. Maslej, "Artificial intelligence index report 2025," *Artificial Intelligence*, 2025.

[4] "Méthodes nouvelles pour le diagnostic du niveau intellectuel des anormaux," *L'année psychologique*, vol. 11, pp. 191–244, 1905. [Online]. Available: %5Curl%7Bhttps://gallica.bnf.fr/ark:/12148/bpt6k9647203g/f207.image%7D

[5] M. B. White and A. E. Hall, "An overview of intelligence testing," *Educational Horizons*, vol. 58, no. 4, pp. 210–216, 1980, Publisher: [Sage Publications, Ltd., Phi Delta Kappa International], ISSN: 0013-175X. Accessed: Sep. 12, 2025. [Online]. Available: https://www.jstor.org/stable/42924403

[6] R. J. Herrnstein and C. A. Murray, *The bell curve: intelligence and class structure in American life*. New York: Free Press, 1994, 845 pp., ISBN: 978-0-02-914673-6.

[7] U. Neisser et al., "Intelligence: Knowns and unknowns," Aug. 7, 1995.

[8] J. Lave, *Cognition in practice: Mind, mathematics and culture in everyday life*. Cambridge University Press, 1988.

[9] R. R. Sharma, "Emotional intelligence from 17th century to 21st century: Perspectives and directions for future research," *Vision*, vol. 12, no. 1, pp. 59–66, Jan. 1, 2008, Publisher: SAGE Publications India, ISSN: 0972-2629. DOI: 10.1177/097226290801200108 Accessed: Sep. 12, 2025. [Online]. Available: https://doi.org/10.1177/097226290801200108

[10] M. Hegarty, "Chapter 7 - components of spatial intelligence," in *Psychology of Learning and Motivation*, ser. The Psychology of Learning and Motivation, vol. 52, Academic Press, Jan. 1, 2010, pp. 265–297. DOI: 10.1016/S0079-7421(10)52007-3 Accessed: Sep. 15, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0079742110520073

[11] *R.U.R.* In *Wikipedia*, Page Version ID: 1298470572, Jul. 2, 2025. Accessed: Sep. 15, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=R.U.R.&oldid=1298470572

[12] "Robot - etymology, origin & meaning," etymonline, Accessed: Sep. 15, 2025. [Online]. Available: https://www.etymonline.com/word/robot

[13] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1, 1943, ISSN: 1522-9602. DOI: 10.1007/BF02478259 Accessed: May 4, 2025. [Online]. Available: https://doi.org/10.1007/BF02478259

[14] "The perceptron — a perceiving and recognizing automaton – brain wars," Accessed: Sep. 15, 2025. [Online]. Available: https://websites.umass.edu/brain-wars/1957-the-birth-of-cognitive-science/the-perceptron-a-perceiving-and-recognizing-automaton/

[15] M. Minsky and S. A. Papert, *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 1969, ISBN: 978-0-262-34393-0. DOI: 10.7551/mitpress/11301.001.0001 Accessed: Sep. 15, 2025. [Online]. Available: https://direct.mit.edu/books/monograph/3132/PerceptronsAn-Introduction-to-Computational

[16] Frank Rosenblatt, "The perceptron: A probabilistic model for information storage and organization (1958)," in *Ideas That Created the Future: Classic Papers of Computer Science*, MIT Press, 1958, pp. 183–190, ISBN: 978-0-262-36317-4. Accessed: Sep. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9357585

[17] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, Jan. 1, 1989, ISSN: 0893-6080. DOI: 10.1016/0893-6080(89)90020-8 Accessed: Sep. 15, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0893608089900208

[18] N. Y. Spartan, Ed., *Principles of Neurodynamics*. 1962, pp. 287–298.

[19] S. Amari, "A theory of adaptive pattern classifiers," *IEEE Transactions on Electronic Computers*, vol. EC-16, no. 3, pp. 299–307, Jun. 1967, ISSN: 0367-7508. DOI: 10.1109/PGEC.1967.264666 Accessed: Sep. 17, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/4039068/

[20] B. Macukow, "Neural networks – state of art, brief history, basic models and architecture," in *Computer Information Systems and Industrial Management*, ISSN: 1611-3349, Springer, Cham, 2016, pp. 3–14, ISBN: 978-3-319-45378-1. DOI: 10.1007/978-3-319-45378-1_1 Accessed: Apr. 6, 2025. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-45378-1_1

[21] A. M. TURING, "I.—computing machinery and intelligence," *Mind*, vol. LIX, no. 236, pp. 433–460, Oct. 1950, ISSN: 0026-4423. DOI: 10.1093/mind/LIX.236.433 eprint: https://academic.oup.com/mind/article-pdf/LIX/236/433/61209000/mind_lix_236_433.pdf. [Online]. Available: https://doi.org/10.1093/mind/LIX.236.433

[22] S. K. Singh, S. Kumar, and P. S. Mehra, "Chat GPT & google bard AI: A review," in *2023 International Conference on IoT, Communication and Automation Technology (ICICAT)*, Jun. 2023, pp. 1–6. DOI: 10.1109/ICICAT57735.2023.10263706 Accessed: Sep. 17, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10263706

[23] S. Ingle and M. Phute, "Tesla autopilot : Semi autonomous driving, an uptick for future autonomy," vol. 03, no. 9,

[24] P. Sun et al., "Scalability in perception for autonomous driving: Waymo open dataset," presented at the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 2446–2454. Accessed: Sep. 17, 2025. [Online]. Available: https://openaccess.thecvf.com/content_CVPR_2020/html/Sun_Scalability_in_Perception_for_Autonomous_Driving_Waymo_Open_Dataset_CVPR_2020_paper.html

[25] K. Black et al., *$ _0$: A vision-language-action flow model for general robot control*, Nov. 13, 2024. DOI: 10.48550/arXiv.2410.24164 arXiv: 2410.24164[cs]. Accessed: Sep. 17, 2025. [Online]. Available: http://arxiv.org/abs/2410.24164

[26] J. Liu, C. Liu, B. Shan, and Ö. S. Ganiyusufoglu, "A computer-assisted interpreting system for multilingual conferences based on automatic speech recognition," *IEEE Access*, vol. 12, pp. 67 498–67 511, 2024, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3400014 Accessed: Sep. 17, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10529505

[27] D. Silver et al., *Mastering chess and shogi by self-play with a general reinforcement learning algorithm - alphago*, Dec. 5, 2017. DOI: 10.48550/arXiv.1712.01815 arXiv: 1712.01815[cs]. Accessed: Oct. 7, 2025. [Online]. Available: http://arxiv.org/abs/1712.01815

[28] SilicoGene. "AI in genomics," SilicoGene, Accessed: Oct. 9, 2025. [Online]. Available: https://silicogene.com/blog/ai-in-genomics/

[29] B. Cottier, R. Rahman, L. Fattorini, N. Maslej, T. Besiroglu, and D. Owen, *The rising costs of training frontier AI models*, Feb. 7, 2025. DOI: 10.48550/arXiv.2405.21015 arXiv: 2405.21015[cs]. Accessed: Sep. 17, 2025. [Online]. Available: http://arxiv.org/abs/2405.21015

[30] C. Corrado, J. Haskel, J. Miranda, and D. Sichel, *Measuring and Accounting for Innovation in the Twenty-First Century*. University of Chicago Press, 2021, ISBN: 978-0-226-72817-9 978-0-226-72820-9. DOI: 10.7208/chicago/9780226728209.001.0001 Accessed: Feb. 23, 2026. [Online]. Available: https://www.bibliovault.org/BV.landing.epl?ISBN=9780226728209

[31] M. Hobbhahn. "Trends in machine learning hardware," Epoch AI, Accessed: Sep. 18, 2025. [Online]. Available: https://epoch.ai/blog/trends-in-machine-learning-hardware

[32] J. Sevilla. "Training compute of frontier AI models grows by 4-5x per year," Epoch AI, Accessed: Sep. 18, 2025. [Online]. Available: https://epoch.ai/blog/training-compute-of-frontier-ai-models-grows-by-4-5x-per-year

[33] K. E. Iverson, "A programming language," 1962. DOI: 10.1145/1460833.1460872

[34] *A new approach to computation reimagines arstificial intelligence*. [Online]. Available: https://www.quantamagazine.org/a-new-approach-to-computation-reimagines-artificial-intelligence-20230413/?mc_cid=ad9a93c472

[35] G. E. Moore, "Cramming more components onto integrated circuits," vol. 38, no. 8, 1965.

[36] M. Roser, H. Ritchie, and E. Mathieu, "What is moore's law?" *Our World in Data*, Mar. 28, 2023. Accessed: Sep. 18, 2025. [Online]. Available: https://ourworldindata.org/moores-law

[37] G. E. Moore, "1.1 no exponential is forever: But "forever" can be delayed!," 2003.

[38] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted mosfet's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974. DOI: 10.1109/JSSC.1974.1050511

[39] N. Kim et al., "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003, ISSN: 1558-0814. DOI: 10.1109/MC.2003.1250885 Accessed: Oct. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1250885

[40] M. Bohr, "A 30 year retrospective on dennard's MOSFET scaling paper," *IEEE Solid-State Circuits Society Newsletter*, vol. 12, no. 1, pp. 11–13, 2007, ISSN: 1098-4232. DOI: 10.1109/N-SSC.2007.4785534 Accessed: Oct. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/4785534

[41] H. Corporaal, "TTAs: Missing the ILP complexity wall," *Journal of Systems Architecture*, vol. 45, no. 12, pp. 949–973, Jun. 1999, ISSN: 13837621. DOI: 10.1016/S1383-7621(98)00046-0 Accessed: Feb. 23, 2026. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1383762198000460

[42] O. Villa et al., "Scaling the power wall: A path to exascale," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ISSN: 2167-4337, Nov. 2014, pp. 830–841. DOI: 10.1109/SC.2014.73 Accessed: Feb. 23, 2026. [Online]. Available: https://ieeexplore.ieee.org/document/7013055/

[43] A. Gholami, Z. Yao, S. Kim, C. Hooper, M. W. Mahoney, and K. Keutzer, *AI and memory wall*, Mar. 21, 2024. DOI: 10.48550/arXiv.2403.14123 arXiv: 2403.14123[cs]. Accessed: May 12, 2025. [Online]. Available: http://arxiv.org/abs/2403.14123

[44] C. Patel et al., "DBGC: Dimension-based generic convolution block for object recognition," *Sensors*, vol. 22, no. 5, p. 1780, Jan. 2022, Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 1424-8220. DOI: 10.3390/s22051780 Accessed: Oct. 14, 2025. [Online]. Available: https://www.mdpi.com/1424-8220/22/5/1780

[45] A. Vaswani et al., *Attention is all you need*, Aug. 1, 2023. DOI: 10.48550/arXiv.1706.03762 arXiv: 1706.03762[cs]. Accessed: May 29, 2024. [Online]. Available: http://arxiv.org/abs/1706.03762

[46] J. L. Hennessy, D. A. Patterson, D. Goldberg, and K. Asanovi , *Computer architecture: a quantitative approach*, 3. ed., [Nachdr.] Amsterdam: Morgan Kaufmann, 2004, ch. 1, ISBN: 978-1-55860-596-1 978-1-55860-724-8.

[47] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, 4. ed., [repr.] Amsterdam Heidelberg: Elsevier [u.a.], 2008, 423 pp., ISBN: 978-0-12-370490-0.

[48] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, Sixth edition. Cambridge, Mass: Morgan Kaufmann Publishers, an imprint of Elsevier, 2019, 617 pp., ISBN: 978-0-12-811905-1.

[49] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, Toronto ON Canada: ACM, Jun. 24, 2017, pp. 1–12, ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246 Accessed: Sep. 2, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3079856.3080246

[50] nvidia, *Nvdla*, 2021. [Online]. Available: https://nvdla.org/

[51] P. Kennedy. "Case study on the google TPU and GDDR5 from hot chips 29," ServeTheHome, Accessed: Sep. 23, 2025. [Online]. Available: https://www.servethehome.com/case-study-google-tpu-gddr5-hot-chips-29/

[52] S. Ghanem, P. Kanungo, G. Panda, and P. Parwekar, "An improved and low-complexity neural network model for curved lane detection of autonomous driving system," *Soft Computing*, vol. 27, no. 1, pp. 493–504, 2023.

[53] J. Knight, "Safety critical systems: Challenges and directions," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, May 2002, pp. 547–550. Accessed: Sep. 21, 2025. [Online]. Available: https://ieeexplore.iee.org/document/1007998

[54] J. Wu et al., "Aircraft flight regime recognition with deep temporal segmentation neural network," *Engineering Applications of Artificial Intelligence*, vol. 120, p. 105 840, 2023.

[55] "Driverless cars to be tested on UK roads by end of 2013," *BBC News*, Jul. 16, 2013. Accessed: Sep. 22, 2025. [Online]. Available: https://www.bbc.com/news/technology-23330681

[56] Available throug th wayback machine. [Online]. Available: https://www.bloomberg.com/news/articles/2013-05-07/tesla-ceo-talking-with-google-about-autopilot-systems

[57] *List of tesla autopilot crashes*, in *Wikipedia*, Page Version ID: 1315103568, Oct. 4, 2025. Accessed: Oct. 9, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_Tesla_Autopilot_crashes&oldid=1315103568

[58] "Tesla deaths: Digital record of tesla crashes resulting in death," Tesla Deaths, Accessed: Oct. 9, 2025. [Online]. Available: https://www.tesladeaths.com

[59] M. Schwall, T. Daniel, T. Victor, F. Favarò, and H. Hohnhold, "Waymo public road safety performance data,"

[60] I. Kabashkin, B. Misnevs, and O. Zervina, "Artificial intelligence in aviation: New professionals for new technologies," *Applied Sciences*, vol. 13, no. 21, p. 11 660, Jan. 2023, Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2076-3417. DOI: 10.3390/app132111660 Accessed: Oct. 9, 2025. [Online]. Available: https://www.mdpi.com/2076-3417/13/21/11660

[61] "2025 AI safety index," Future of Life Institute, 2025. Accessed: Sep. 18, 2025. [Online]. Available: https://futureoflife.org/ai-safety-index-summer-2025/

[62] A. Avi˘zienis, J.-C. Laprie, and B. Randell, "Fundamental concepts of dependability," 2001.

[63] V. Moskalenko, V. Kharchenko, A. Moskalenko, and B. Kuzikov, "Resilience and resilient systems of artificial intelligence: Taxonomy , models and methods," *Algorithms*, vol. 16, no. 3, p. 165, Mar. 2023, Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 1999-4893. DOI: 10.3390/a16030165 Accessed: Sep. 23, 2025. [Online]. Available: https://www.mdpi.com/1999-4893/16/3/165

[64] S. Webb, T. Rainforth, Y. W. Teh, and M. P. Kumar, *A statistical approach to assessing neural network robustness*, Feb. 21, 2019. DOI: 10.48550/arXiv.1811.07209 arXiv: 1811.07209[cs, stat]. Accessed: May 1, 2023. [Online]. Available: http://arxiv.org/abs/1811.07209

[65] L. V. Haar, T. Elvira, and O. Ochoa, "An analysis of explainability methods for convolutional neural networks," *Engineering Applications of Artificial Intelligence*, vol. 117, p. 105 606, Jan. 1, 2023, ISSN: 0952-1976. DOI: 10.1016/j.engappai.2022.105606 Accessed: Nov. 20, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0952197622005966

[66] P.-J. Kindermans, K. Schütt, K.-R. Müller, and S. Dähne, *Investigating the influence of noise and distractors on the interpretation of neural networks*, Nov. 22, 2016. DOI: 10.48550/arXiv.1611.07270 arXiv: 1611.07270[cs, stat]. Accessed: May 1, 2023. [Online]. Available: http://arxiv.org/abs/1611.07270

[67] P. Tamagnini, J. Krause, A. Dasgupta, and E. Bertini, "Interpreting black-box classifiers using instance-level visual explanations," in *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, Chicago IL USA: ACM, May 14, 2017, pp. 1–6, ISBN: 978-1-4503-5029-7. DOI: 10.1145/3077257.3077260 Accessed: Mar. 28, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3077257.3077260

[68] F. K. Dosilovic, M. Brcic, and N. Hlupic, "Explainable artificial intelligence: A survey," in *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija: IEEE, May 2018, pp. 0210–0215, ISBN: 978-953-233-095-3. DOI: 10.23919/MIPRO.2018.8400040 Accessed: Mar. 27, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/8400040/

[69] M. Quirk and J. Serda, *Semiconductor manufacturing technology*. Upper Saddle River, NJ: Prentice Hall, 2001, 666 pp., ISBN: 978-0-13-081520-0.

[70] D. K. Schroder, "Negative bias temperature instability: What do we understand?" *Microelectronics Reliability*, vol. 47, no. 6, pp. 841–852, 2007, Modelling the Negative Bias Temperature Instability, ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2006.10.006 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002627140600374X

[71] D. Pierce and P. Brusius, "Electromigration: A review," *Microelectronics Reliability*, vol. 37, no. 7, pp. 1053–1072, 1997, Reliability Physics of Advanced Electron Devices, ISSN: 0026-2714. DOI: https://doi.org/10.1016/S0026-2714(96)00268-5 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026271496002685

[72] F. Wang and V. D. Agrawal, "Single event upset: An embedded tutorial," in *21st International Conference on VLSI Design (VLSID 2008)*, ISSN: 2380-6923, Jan. 2008, pp. 429–434. DOI: 10.1109/VLSI.2008.28 Accessed: Nov. 1, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/4450538/?arnumber=4450538

[73] B. Bittel, M. Shamsa, B. Inkley, A. Gur, D. Lerner, and M. Adams, "Data center silent data errors: Implications to artificial intelligence workloads & mitigations," in *2024 IEEE International Reliability Physics Symposium (IRPS)*, ISSN: 1938-1891, Apr. 2024, pp. 1–5. DOI: 10.1109/IRPS48228.2024.10529375 Accessed: Oct. 29, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10529375/?arnumber=10529375

CHAPTER 2

Background

In this chapter we will introduce all the necessary background to read the rest of this manuscript. The first section, section 2.1, gives an overview on the technology of *modern Artificial Intelligence (AI)*, Neural Networks. This section introduces the most basic concepts and the most popular convolutional models, as they are the ones used in the rest of the manuscript. The second section, section 2.2 is dedicated to the definitions of reliability and fault tolerance, focusing on the context of Neural Networks (NNs), as it is the driving force of this manuscript. Finally, the last section, section 2.3 is dedicated to showing the last frontier of micro-architectural accelerators, introducing the systolic array as the main component of future hardware dedicated to AI.

2.1 Neural Networks

In the introduction we saw a bit of history of NN. Here we will introduce the technical background necessary to understand the rest of the manuscript. Before any other section though, it is needed to explain some terminology which might be confusing in the rest of the manuscript.

We already mentioned in section 1.4 that with the term AI we actually mean NN. Other than that, we need to do the distinction between NN, Convolutional Neural Network (CNN) and Deep Neural Network (DNN):

Neural Network (NN) this term is to be intended as the general class of neural networks. It comprises them all. Sometimes, it is used to specifically refer to feed-forward NN;

Deep Neural Network (DNN) this second class refers to all those NN that are *deep*, referring to the number of consecutive layers the network has. In the literature the meaning of "deep" is vague, so for the scope of this manuscript, we can identify all those networks that have more than one layer, rendering this term effectively a synonym of MultiLayer Perceptrons (MLP) and (de facto) NN.

Convolutional Neural Network (CNN) this term describes all those NN which have at least one convolutional layer (which will be explained in subsection 2.1.3), de facto making it a synonym of the other two, since in this manuscript all the experiments described are CNNs.

2.1.1 Feed Forward NN

A perceptron is made by many neurons. The model of a neuron is usually represented as follows, in Figure 2.1. The inputs x_i are multiplied by the weights and are summed together to compute y. Then, the y is fed to a function σ, which we call *the activation function*, and the output r is given. Mathematically, we have the following relationship:

$$r = \sigma(y) = \sigma(\mathbf{W}\mathbf{x}) \tag{2.1}$$

[5] With the term "pointwise", we describe a scalar function that is applied to every component of the input vector (or matrix).

Basically, it is a linear transformation followed by a nonlinear pointwise[5]. We should emphasize the nature of the nonlinear activation function: if σ was linear, then a NN would be just a combination of linear functions, and a combination of linear function is linear itself. The magic happens when introducing the non-linearity in the system.

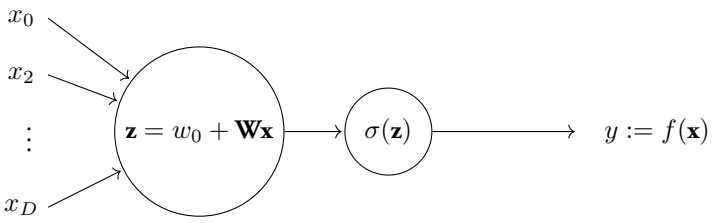


Figure 2.1: Single processing unit and its components. The input $\mathbf{x} = [x_1, \dots, x_D]$ is multiplied by a set of weights $\mathbf{W} = [w_1, \dots, w_D]$ and the bias w_0 is added to the multiplication, yielding $\mathbf{z}$. The activation function, denoted by σ, is then applied to $\mathbf{z}$ finally yielding y. The function $f = \sigma(w_0 + \mathbf{W}\mathbf{x})$ is then differentiable

Putting many neurons one after the other and connecting them together, we would obtain a perceptron, as shown in Figure 2.2. This model is mathematically equivalent to the one described by Rosenblatt (cf. subsection 1.1.1). The figure shows two layers: an input layer and an output layer. In the introduction we said how the evolution of NN was slowed down, as it was demonstrated that such a perceptron does not give enough generalization. The breakthrough came with the addition of multiple "hidden" layers, to be put and concatenated between the layers shown in the figure. A structure as described works effectively as a universal function approximator. It should be obvious that each layer is fundamentally a matrix multiplication followed by a pointwise activation. This means that an MLP is effectively a succession of matrix multiplications followed by a non-linear pointwise function. A stress should be put on the nature of σ; such function must be non-linear, as no "interesting" space transformations comes from a succession of linear functions, that is itself a linear function. For this reason, one of the most used functions is the sigmoid [1]. Thus, the general form of a feed-forward NN with n layers is the following:

$$f_i(x) = \begin{cases} \sigma(\theta_i x) & \text{if } i = 1 \\ \sigma(\theta_i f_{i-1}(x)) & \text{otherwise} \end{cases} \tag{2.2}$$

We can rewrite the function $f_\theta(x)$ in closed form:

$$f = \sigma(\theta_n \sigma(\theta_{n-1} \sigma(\theta_{n-3} \cdots (\theta_1 x)))) \tag{2.3}$$

Figure 2.2 shows only two layers: the input and the output. Nevertheless, it is possible to put other layers in between. In fact, this practice gave rise to the so-called Deep Learning. It refers to the depth of the network itself. As we will see in subsection 2.1.4, using deeper NNs achieves better results. The current understanding is that the layers near the inputs can capture simple features of an input. The

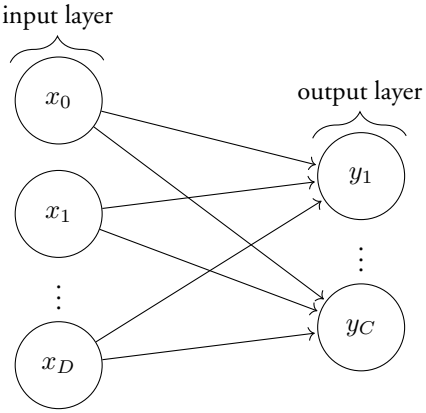


Figure 2.2: The perceptron consists of D input units and C output units. All units are labeled according to their output: $y_i = f(z_i)$ in the case of output units; x_i in the case of input units. The input values x_i are propagated to each output unit using the weighted sum propagation rule. The additional input value $x_0 := 1$ is used to include the biases as weights.

following layers then can combine these features into more complex features. Theoretically, this feature-building can continue indefinitely.

It is obvious that the more layers there are, the more parameters there are. If the beginning of the NN era came with about 60 thousands parameters (it is the case for LeNet-5 subsection 2.1.4), modern parameters have increased considerably: the popular ChatGPT v4 [2] has about 2 trillions parameters[3].

Classification Tasks

Although being one of the earliest task to have been explored, classification tasks are predominant all over the manuscript. The reason is simple: they are easy to understand, widely useful and especially small (in terms of parameters). Classification models have a useful component: the last layer employs an activation function whose role is to give probabilities in the output. To this extent, if there are n classes, the output layer will have n outputs, each one representing the probability that the input give belongs to that class. To do this, the activation function is a softmax. Its closed form is the following: Definition: softmax

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \quad \forall j = \{1, \dots, K\}. \tag{2.4}$$

It should be clear that it is a pointwise function, as expected, since the input is the vector $\mathbf{z}$ and it has K components. The softmax function is simply an exponential scaling of the values of the input $\mathbf{z}$ and, notably, as we interpret the output as a probability distribution, the following equation must hold:

$$\sum_j \sigma(\mathbf{z})_j = 1, \tag{2.5}$$

given the nature of this transformation, this is always the case.

To conclude this chapter, we will say that, in order to respect this convention about probability, the labels must be also vectors with n components, where the only component different from 0 is the correct class which has "probability" 1. In this way Equation 2.5 always holds. In the community, this transformation is called one-hot encoding. Figure 2.3 shows an example of a handwritten digit showing the value of 3. The label then should be the following, with 0-indexed arrays: Definition: one-hot encoding

$$Label_3 = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.6}$$

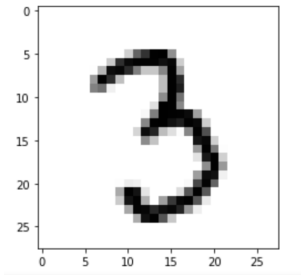


Figure 2.3: Example of a figure taken from the MNIST dataset[4].

2.1.2 Training Algorithm

Implementing this model on its own does not serve much purpose, as the trick lies in finding the matrices W for each layer. Using gradient-based algorithms proved to be very effective, as demonstrated already Amari [5] in 1967. Much similarly to what described by Rosenblatt in '58, the general idea is to show the NN some example, and then force it to produce the wanted outcome. More specifically, we use gradient descent to minimize a cost function J. Although there are more sophisticated strategies, in this section we will describe the classic Stochastic Gradinet Descent (SGD), which is the basis for all the other popular algorithms.

Let $f_\theta(x)$ be a function describing a feed-forward NN, like the one showed in Equation 2.3. SGD works by updating the parameters (or weights) θ of f by computing the gradient of the cost function J at the point of interest:

$$\theta' = \theta - \epsilon \nabla_x L(f(x), y) \tag{2.7}$$

where ϵ is called the "learning rate".

SGD is *stochastic* because usually using the whole dataset to train the algorithm is very expensive and might need too many resources. For this reason, only a sample of the subset is used to perform the training, making the algorithm stochastic. In order to do that, we can estimate the cost function J using only m samples from the dataset:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} L(x_i, y_i) \tag{2.8}$$

where L is the loss function of the single example. Using the estimation for cost function, we can perform the update of the parameters as in Equation 2.7. Most commonly, the loss function that is used for training NN is the negative log-likelihood of the estimation, so that to maximize the likelihood.

In conclusion, the general algorithm is described as follows:

- Compute the output $y_i = f_\theta(x_i)$ of a subset with m samples of the dataset,
- Compute the cost function $J(\theta)$,
- Update the parameters $\theta' = \theta - \epsilon J(\theta)$.
- Repeat from step 1 until desired.

As to when stopping the algorithm, it is basically up to the user to decide. A common strategy consists in having a performance metric to be evaluated during the computation of y_i and stop the algorithm upon reaching the desired performance. Another possibility is to use *early stopping*: if the cost function does not improve in the span of n iterations, stop the training.

The Dataset

The explosion of NNs came with In the context of classification tasks, the process typically involves two main phases: training and evaluation [6]. These rely on a labeled dataset $\boldsymbol{D} = \{(x_i, y_i)$ of cardinality N, where x_i represents the input features tensor and y_i denotes the corresponding class labels. Typically, $\boldsymbol{D}$ is divided into two subsets: the *training set* $\boldsymbol{D}_{\text{train}}$ and the *testing set* $\boldsymbol{D}_{\text{test}}$. Sometimes, the dataset is split in three: a training set, a validation set and a testing set. The validation set is used to tune the hyperparameters (e.g. the learning rate), while the other two have the same function.

During the training phase, the training set is used to update using an algorithm like the one seen in the previous section. At the end of a complete iteration over the training set, we use the testing set to evaluate the performances of the model, in an unbiased way: the weights are in fact tuned for the specific data of the training set, and the performances computed with the training set would be far from realistic. Doing too many passes on the training has the effect of over-fitting the weights: the model "learns" exactly how the training set is done and loses its generalization capabilities.

The testing phase involves evaluating the trained model f_θ on unseen samples from $\boldsymbol{D}_{\text{test}}$. This step provides an estimate of the model's generalization performance by calculating metrics like accuracy, precision, recall, or other relevant measures on $\boldsymbol{D}_{\text{test}}$. Let x_t be a sample belonging to the test set $\boldsymbol{D}_{\text{test}} = \{x_0, x_1, x_2, \dots\}$, and y_t be the corresponding class.

2.1.3 Convolutional Neural Network

In 1998, the famous paper from LeCun [1] introduced CNNs. There were mainly two features of importance:

Diminishing the number of parameters Images are large: several hundreds of pixels are to be processed. For a feed-forward NN that would increase the number of weights considerably. On the other hand, Convolutional Layers have s (relatively) small number of parameters, as the weights are shared among all the features of the inputs.

Make the algorithm invariant to translation Feed Forward NN are sensitive to the position of the features in the input vector. Learning the same features in different positions would require retraining of the NN. On the other hand, Convolutional Layers are invariant to translation: the same weights are used across all the image, so the feature are picked up in the same way, regardless of their position. Learning new features is a matter of increasing the number of channels that the layer has.

These features make CNN well suited for computer vision applications. In the literature we can find many examples of the effectiveness of such networks. The rest of this section will (i) describe in details the inner working of the convolutional layer, since it is effectively the focus of the manuscript and (ii) briefly show some of the most popular and relevant architecture which employ convolutional layers and then

Convolution

A 1D convolution is a fundamental operation in signal processing and deep learning that involves sliding a filter (or kernel) across a one-dimensional input array, such as a time-series signal or a sequence of data points. At each position, the filter performs an element-wise multiplication with the corresponding segment of the input, sums the results, and produces a single output value. This process effectively captures local patterns or features in the data, such as edges, trends, or motifs, by emphasizing certain characteristics based on the filter's weights. The output is a new 1D array where each value is influenced by the input values within the filter's receptive field, making it useful for tasks like noise reduction, feature extraction, and pattern recognition in sequential data. Mathematically, the convolution between two time-dependent signals $f(t)$ and $g(t)$ is described as follows:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) \cdot g(t - \tau)\, \mathrm{d}\tau \tag{2.9}$$

In the time-discrete domain, f_{k+n} representing the $(\mathrm{k} + \mathrm{n})^{\mathrm{th}}$ sample:

$$(f * g)_k = \sum_{n=-\infty}^{\infty} f_n \cdot g_{k-n} = \sum_{n=-\infty}^{\infty} f_{k-n} \cdot g_n \tag{2.10}$$

In practice, only the input signal f is infinite as it is often a stream of data output by a sensor. g is the filtering signal and of finite length: $g_k = 0$ for $k > l$ with l the length of the signal.

$$(f * g)_k = \sum_{n=1}^{l} f_{k-n} \cdot g_n = f_{k-1} \cdot g_1 + \cdots + f_{k-l} \cdot g_l \tag{2.11}$$

The convolutional layer reduces the number of parameters of the NN, other than introducing the features discussed above. The two facts are actually correlated: convolution is more general, as it introduces shift-independence, so fewer parameters are needed to learn the same amount of information. This is an advantage both in inference and in training: gradient-based optimization is done on the parameters and fewer parameters means less gradients, thus improving also the training efficiency.

Convolutional Layer

Feed forward NN are the basics of NN. Nevertheless, the amount of parameters to train is proportional to the size of the input of the NN. To make training more efficient, LeCun implemented a so called CNN in 1998 [1]. The difference with feed-forward is that the number of parameters is decided by the user and is independent of the size of the input. On top of that, the algorithm is forced to learn spatial relationships of the inputs. Finally, the convolution is independent to spatial shifts, making it more suitable to identify features in images. The operation of 2D convolution is illustrated in Figure 2.4.

LeCun obtained great results with his implementation, achieving more than 90% classification accuracy. Nevertheless, the dataset they used was small compared to today standard. The real revolution happened in 2013 when a computer scientist by the name of Alex Krizhevsky implemented a version of the neural network as described by Yann LeCun in 1998 which had a simple, yet powerful, difference: the size. Nevertheless, renewed interest has sparked only in 2016 with the advent of AlexNet [7].

A convolutional layer, as meant in modern CNNs is composed by $m \times n$ 2D convolutions. In the literature, the third dimension of an image is referred to as the *channel*. Images are usually 3D tensor, as each channel encodes a color among red, green and blue. The weights of a layer are 4-dimensional: two dimensions for the filters themselves, a third one to describe multiple filters for each channel of input and a fourth one for a different set of filters for each output channel. The rationale behind this design is that each set of filters is able to pick up different features of the input. The m channels of the input image (usually is the three colors of an image, but it could also be the third dimension of the output of another convolutional layer) are convoluted with each of the filter and then summed together to obtain the corresponding channel in the output. We can describe the operation mathematically assuming that W is a 4D tensor with shapes (n, m, kw, kh) describing the number of output channels m, the number of channels of the input n and the filter's width and height kw, kh respectively; I is a 3D tensor describing the input to the layer with shape n, w, h. Finally, R is a 3D tensor describing the output of the layer, with shape m, w, h. The convolution is thus implemented as follows:

$$R(c) = \sum_{k=1}^{n} W(k, c) * I(k) \quad 1 \leq c \leq m \tag{2.12}$$

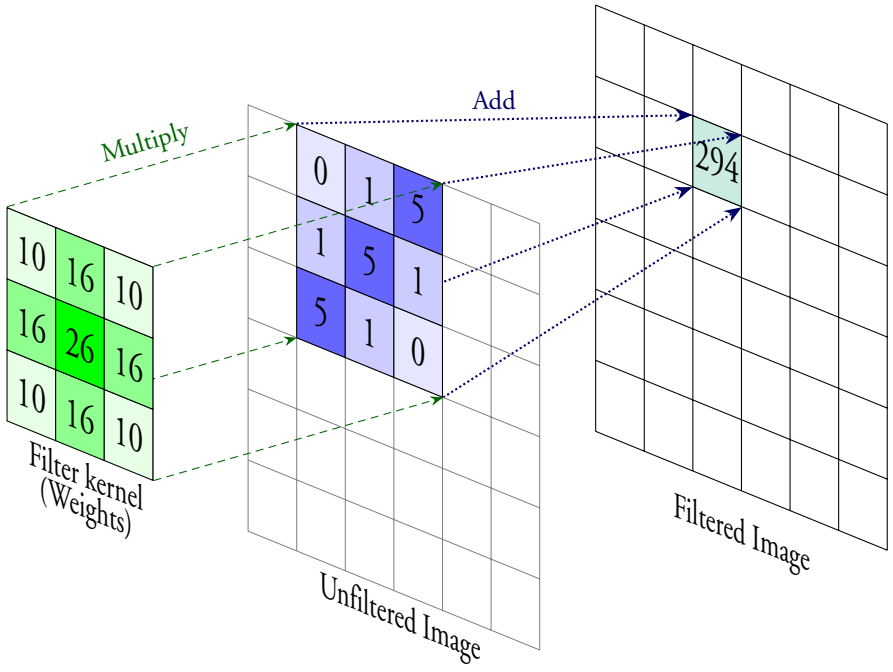


Figure 2.4: Illustration of 2D convolution. The filter kernel corresponds to the weights that are used in a convolutional layer. Specifically, this image describes only the last two dimensions of it. The Unfiltered Image is the input fed to the layer.

2.1.4 Popular Neural Architectures

In this section we will illustrate the most popular CNNs, highlighting the main innovations.

LeNet-5

Introduced in 1998, was the first CNN of them all. It is composed by 5 layers: 2 convolutional layers followed by average pooling and 3 fully connected layers. LeNet-5 introduced:

the convolutional layer which was already discussed in Equation 2.1.3;

the pooling layer this layer has the objective of performing subsampling in the spatial features of the previous layer. It is always placed after a convolutional one. The reasoning of the authors, in the original paper was that «once a feature has been located, its exact location becomes less important». For this reason, a simple way to reduce the precision of the position is to just reduce the spatial resolution of the feature map. To this extent, the feature map is often halved in size, while averaging the four region of the original feature map. Figure 2.6 shows an example of the operation.

stochastic gradient descent although this technique was known already in the 60's (cf. subsection 2.1.2), the authors highlight how such algorithm were never thought to be used for the optimization of non-linear functions, as it is LeNet-5. Using stochastic gradient descent to train NN was the real revolution. This way, it is almost as if the NN itself builds its own feature extractor, as opposed to carefully handcrafting one, as it was the case back then.

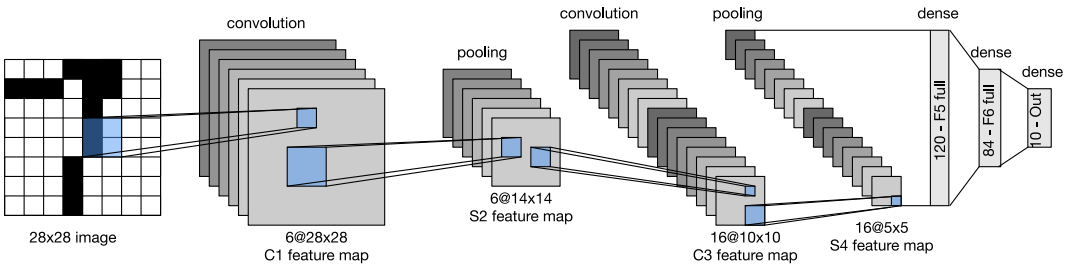


Figure 2.5: Original LeNet-5 architecture.

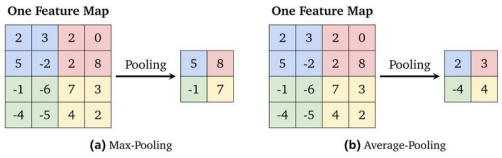


Figure 2.6: Pooling. (a) shows max pooling: on the four regions, only the maximum value is kept. (b) shows average pooling: the values of each region are averaged together.

AlexNet

Many years after LeNet-5, in 2012 the researcher Alex Krizhevsky introduced AlexNet [8]. It uses, for the first time, a much less expensive activation function: ReLU, or Rectified Linear Unit. The ReLU function is defined as follows:

$$\text{ReLU}(x) = max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.13}$$

and it is plotted in Figure 2.7.

The authors show how, using ReLU is convenient not only because it is so inexpensive, but it also achieves faster convergence due to its non-saturating nature: in contrast to the sigmoid, which saturates to -1 towards $-$ inf and $+1$ towards inf, ReLU is the same as the identity in the positive semi-plane. The use of ReLU has the effect of speeding up training, increasing the efficiency once more.

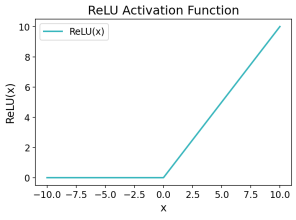


Figure 2.7: A graph depicting the ReLU function.

Other than this, AlexNet introduced the max-pooling, similar to pooling but only takes the maximum value in a region. Figure 2.6 shows an example of max-pooling. Finally, another "novelty" is: more parameters. AlexNet is, in a sense[6], a scaled up version of LeNet-5. With its 60 millions parameters, it is 3 orders of magnitude bigger than LeNet-5. The reasons are obvious: (i) on one side, more parameters just means that more information can be fit into the CNN, potentially providing better generalization, (ii) on the other side, by 2012 it was just possible; hardware improvements made it such a bigger network could be trained efficiently. To train AlexNet the authors used a set of two Graphical Processing Units (GPUs), with 3Gb of memory, technology that was not available in 1998.

[6] ignoring the differences in pooling type and activation function

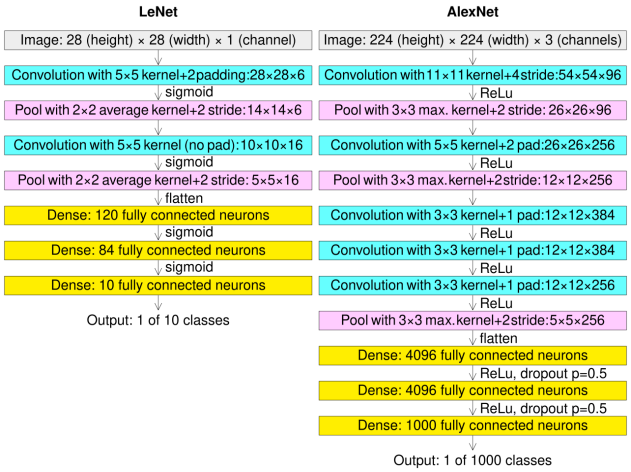


Figure 2.8: A comparison between the LeNet-5 architecture and AlexNet architecture.

The VGG Family

In 2015 this idea of *Deep Learning* was greatly pushed.

The VGG family [9] of architectures is probably the first real improvement after AlexNet. The idea of the authors was to study the depth of their architecture, introducing more convolutional layers. In order to do that, they used relatively small filters for their layers rather than big ones as it other works. This, as the authors say, allows for a more relaxed feature extraction: previously, the CNNs had large filters, in order to pick up more pixels and, hopefully, more features. This also meant that there was a relatively small limit to how deep the architecture could be, as the information available would be consumed more quickly with bigger filters. On top of that, not all the convolutions are followed by pooling, giving the possibility of localizing more complex features and effectively increasing the receptive field.

As a reminder: Deep Learning is the branch of Machine Learning (ML) involved with NNs which are *deep*, meaning that they have many layers.

The ResNet Family

Inception did not solve the vanishing gradient problem: for this reason in 2015 residual blocks were introduced by [10].

The ResNet (Residual Network) family of architectures, introduced in 2015 [10], marked yet another turning point in Deep Learning by addressing the long-standing challenge of training very deep neural networks. Before ResNet, as networks grew deeper, they often suffered from vanishing gradients. The vanishing gradients occur in NNs happens where the gradients used to update weights becomes so small that learning effectively stalled. This happens because gradients are computed as the product of derivatives from each layer, and repeated multiplication by values less than 1 (common with activation functions like sigmoid or tanh) causes the gradient to shrink exponentially. ResNet's breakthrough was the introduction of "skip connections" or "residual blocks," depicted in Figure 2.10, which allows the network to learn residual functions by bypassing one or more layers. This innovation enabled the training of networks with hundreds or even thousands of layers, such as ResNet-50, ResNet-101, and ResNet-152, without degrading performance. By allowing gradients to flow more easily through the network, ResNet architectures achieved state-of-the-art results in image classification, object detection, and other computer vision tasks,

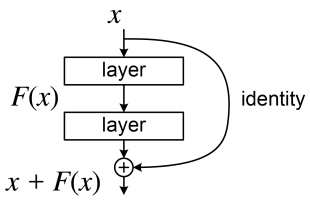


Figure 2.10: The residual block.

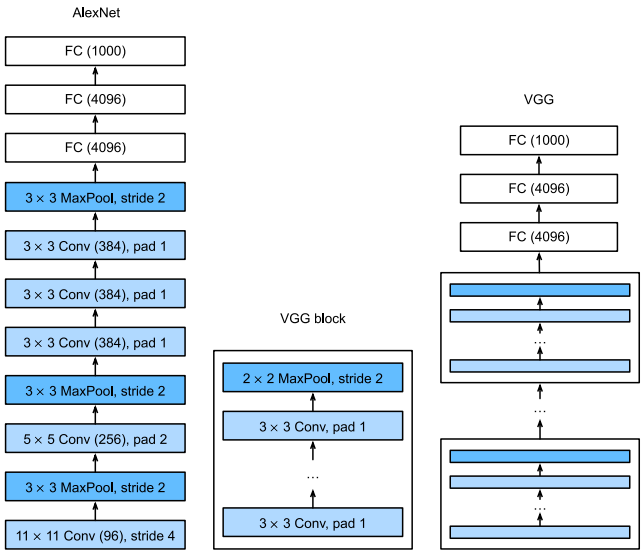


Figure 2.9: The architecture of the VGG family, compared to AlexNet. It is shown how the filters of the convolutions are small and follow each other, as opposed to AlexNet, where each convolution is followed by a pooling layer.

setting a new standard for deep learning models.

2.2 Faults and Reliability

As seen in 1.5 hardware faults refer to physical malfunctions or degradations in the components of a computing system. When hardware faults occur, they can disrupt the normal operation of an application, leading to incorrect computations, system crashes, or data corruption. In fact, it has been proven that hardware faults (generally affecting hardware accelerators) can significantly affect the inference, leading to DNN prediction failures that are likely to lead to a detrimental effect on the application [12], [13], [14]. Such faults are inevitable in real-world systems, especially as hardware ages or operates in harsh environments, making them a critical concern for engineers and architects designing robust applications.

To this extent, the first section presents an overview on fault models and how they relate to the application. Later on, we will introduce the concept of Fault Injection (FI), a very important tool used to study the effects of hardware faults in complex systems.

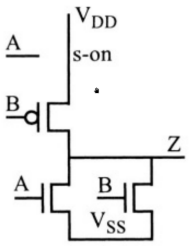


Figure 2.11: Stuck-on representation. A is supposed to be a transistor, but is modeled as a short circuit. Image from [11].

2.2.1 Fault models

As defined in the previous chapter, a *Fault* is a representation of a defect at the abstracted function level. That is, a representation of the behavior of a physical defect as "seen" at a logical level. For example, a poorly connected transistor (for whatever reason) might always be on, resulting in a stuck-on fault, which is represented, at a logical level, by a short circuit rather than a transistor, like in Figure 2.11. Fault taxonomy is intricate. A fault can be modeled at different abstraction layers: it could

be a gate-level model or a transistor-level model. But this is not enough, as the fault model a defect found between two wire segments, thus we talk about bridging fault models. But it could be the case that the logic is correct, while the circuit cannot operate at the system's frequency. Due to the complexity of these definitions, we will focus on the duration of the faults, from which three main categories could be identified: permanent, transient and intermittent faults, each with distinct causes, characteristics, and implications. Permanent faults arise from irreversible physical damage to electronic components. These faults typically result from manufacturing defects, material degradation over time, or external physical damage. Once they occur, they persist until the faulty component is repaired or replaced. The most commonly used abstraction for modeling permanent faults is the stuck-at-fault model, where a component or node in a circuit is permanently "stuck" at either a high (1) or low (0) voltage state [15]. Transient faults, conversely, are temporary disruptions caused by external environmental factors. High-energy particle strikes, electrical noise, and electromagnetic interference are common sources of transient faults [15]. Unlike permanent faults, transient faults do not result in lasting damage to the hardware; instead, they momentarily alter the device's behavior. A typical manifestation of transient faults is a random bit-flip in memory or register values, which can lead to erroneous computations if not detected and corrected promptly. The different nature of permanent and transient faults necessitates tailored approaches for their management [16]. Intermittent faults manifest temporarily, like transient faults, but recurrently.

In general, a fault can be masked or propagated. Figure 2.12 shows a representation of the way an external physical fault is propagated up in the abstraction layers. It is possible to see that many faults are just masked and do not alter the computations in any way. When a fault is not masked, it becomes an error, by altering the functionality. Altering the functionality can still produce reliable outcomes. When that does not happen, we talk about failure: the system does not comply with the specification.

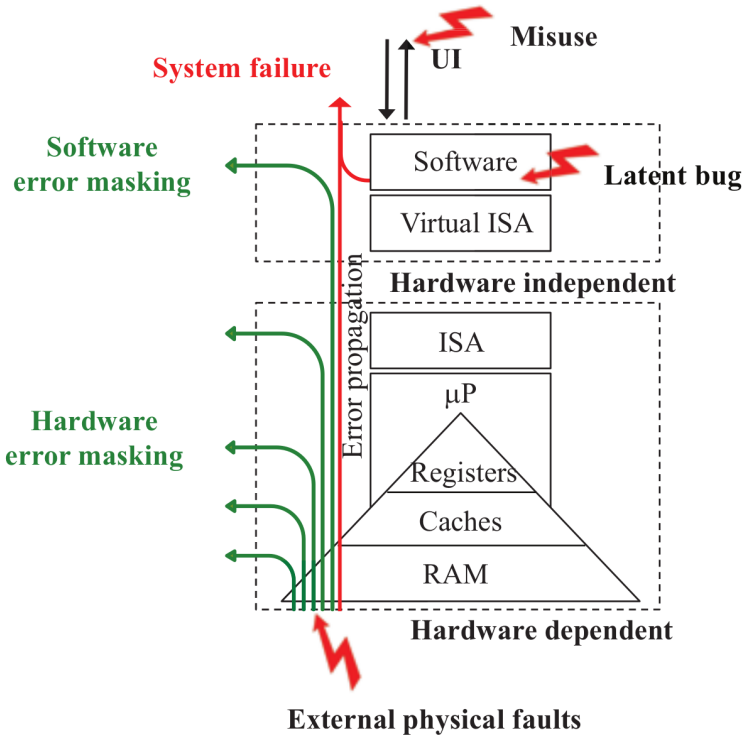


Figure 2.12: Fault propagation in an electronic device. Figure taken from [15].

2.2.2 Fault Injection

FI is the most common way of carrying experiments to study the reliability of systems. Studying complex systems is difficult, in that so many behaviors can arise that it is virtually impossible to take them all into account, and even then, the complexity of the system makes it impossible to run experiments in acceptable times. For these reasons FI is one of the most used tools to study systems like the one at hand.

In 2024, an excellent study [17] was published showcasing all the different methodologies to perform reliability assessment for the NN's. Given the complexity of the problem, many different techniques were established to study the reliability of harder systems. We can distinguish loosely three categories, following the taxonomy of [18]:

Radiation campaigns The fault injection is carried out using radiations. Radiations are useful to simulate aging and adverse conditions like space. We can find many studies in this category. There are three categories of this work, as is possible to expose an Field Programmable Gate Array (FPGA), a GPU, For example, [19] performs fault injection through high-energy neutron radiation beam on a Tensor Processing Unit (TPU) running a ResNet-50, finding that the output of the convolution is slightly altered, despite the high error rate. Other studies tried to fault inject GPU. For example, in [20] the authors performed the irradiation through a neutron beam on an NVIDIA GPU finding that the existing chip-level protections are capable of mitigating all the Silent Data Corruption (SDC). The drawback of these methods is the cost: using radiation is expensive in itself, as it requires specialized equipments, and to that there is also the coset of the chip which might become irreversibly damaged. Also, determining where the fault is and how it reached is challenging if not impossible.

Fault Emulation In this category fall all the studies which emulated the effect of a fault directly on the hardware used to run the inference. Most of the works are carried out in GPU, although it is not uncommon to see Central Processing Unit (CPU) or FPGA studies. The injection is performed by altering the fault-free behavior of the device via software. For example, in the FPGA is possible to alter the configuration bits, leading to different connections and LUTs (LookUp Tables). On the CPU, is possible to alter instructions or the register file, while on the GPU it is possible to alter also weights and feature maps (i.e. the output of a layer).

Fault Simulation This is by a great margin the most common technique. It is cost-effective and allows precise control over the experimental conditions. Faults can be injected either at the architectural representation of the hardware model or directly at the software level of the application. This category can be differentiated in: (i) hardware-independent platforms and (ii) hardware-aware ones. The former has the objective of studying the network as such, without doing any assumptions on hardware that runs the models, while the other better models reality. These techniques allow researchers to explore various fault scenarios without the need for extensive hardware resources. However, the primary challenge of simulation-based FI lies in the time required for exhaustive campaigns, especially for large DNNs.

Although there so many ways of performing FI, we will explain the process that we used to evaluate all the FI in this manuscript. Please note that this kind of assessment is related to the application functionality.

2.2.3 Functional Assessment through FI

The basics of FI is in the comparison between two systems: one which is called the Golden System the other one that is called the Faulty System. With the term *Golden System* we refer to the system operating as expected. A synonym is *fault-free system*. Similarly, the *Faulty System* is the system in which a fault is

injected. Figure 2.13 represents the situation of a FI. We inject a fault in the faulty system and simulate the two systems, using the same input. At the end of the simulation, the outputs of the two systems are gathered and compared using some metric. There is a number of metrics that can be used for our purposes. For NN we will present those that are recurrently used in the rest of the manuscript. An important fact to point out is that the simulation of the systems has to happen with some input data and with some faults injected. Each different fault and input datum need a new simulation. For this reason, FI takes an extremely long time: the number of simulations is often in the order of 10^5.

Definition: Golden System, Faulty System

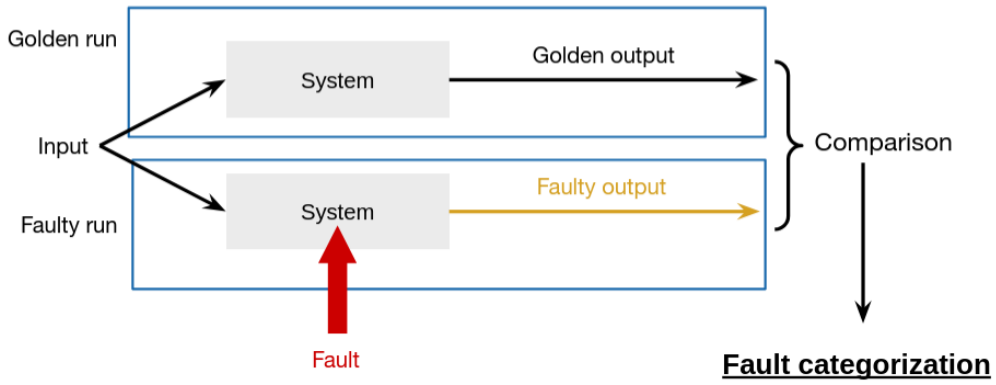


Figure 2.13: Fault Injection Setup

In the context of classification, the output of the NN is a vector with as many components as the classes that exist in the problem. So, for each input we have the fault-free output $\hat{Y}$ and the faulty output a vector $\tilde{Y}$. The classification labels $\hat{y}$ and $\tilde{y}$ are obtained as $\hat{y} = argmax(\hat{Y})$ and $\tilde{y} = argmax(\tilde{Y})$. In other words, the classification label is the class with the highest value in the output.

Fault Classification

For the purpose of the resilience classification, we compare the two vectors $\hat{Y}$ and $\tilde{Y}$ for each pair input-fault as follows:

- we call a fault *masked* when $\hat{Y} = \tilde{Y}$,
- *good* when $\hat{y} = \tilde{y}$ and $\frac{max(\tilde{Y})}{max(\hat{Y})} > 1$,
- *accept* when $\hat{y} = \tilde{y}$ and $0.95 < \frac{max(\tilde{Y})}{max(\hat{Y})} < 1$,
- *warning* when $\hat{y} = \tilde{y}$ and $\frac{max(\tilde{Y})}{max(\hat{Y})} < 0.95$,
- *critical* when $\hat{y} \neq \tilde{y}$.

Sometime, the two categories *good, accept* are grouped together and called *safe*, and the two categories *warning, critical* are grouped together into *critical*. Then, the results are given showing the portion of faults that are classified within each of the given categories.

2.2.4 Statistical Fault Injection

To perform a FI, ideally, we would inject all the possible faults that can happen in a system. Nevertheless, oftentimes the space of injectable faults is so big that such a campaign would take years to finish (if not

decades). For this reason, researchers have developed statistical sampling techniques [21]. Statistical Fault Injection (SFI) involves selecting a statistically significant subset of faults to inject, based on predefined confidence intervals and error margins.

The number of injections to be randomly performed can be computed using Equation (2.14), from [22], where, N is the fault population size, p is the estimated probability of faults resulting in a failure; e is the desired margin of and t is the cut-off point (or critical value) corresponding to the desired confidence level.

$$n = \frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 \cdot p \cdot (1-p)}} \tag{2.14}$$

For example:

- if fault universe had $N = 100'000$ individuals, i.e. possible faults,
- and we wanted an error $e = 5\%$, i.e. the error with which we know n,
- and a confidence level of 90%, implying $t = 1.28$, i.e. the probability that our measurements are within the error interval,
- and a probability that an individual manifests as an error op $p = 50\%$, i.e. a fault has equal probability of manifesting itself as an error (this is the worst case),

we have $n = 163.9 \approx 164$ faults to inject. Figure 2.14 shows a plot of Equation 2.14 with n as a function of N.

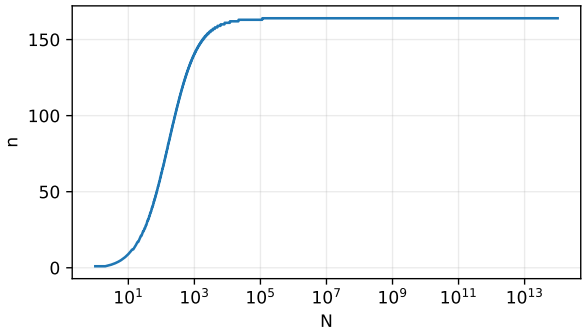


Figure 2.14: This figure is a plot of n as a function of n, with parameters $p = 0.5, t = 1.28, e = 5\%$. As you can see, the value of n saturates with N large enough. Figure based on [22].

As a final note to this section, we want to bring the attention of the reader to the fact that this manuscript is involved solely in functional safety of NNs, meaning that all the FI campaigns carried out, are used to validate the NN at hand to ensure correct functionality.

2.3 Systolic Array

The history of Systolic Arrays (SAs) dates back to the late '70s. Back then, somewhat similarly to today, the researchers Kung and Leiserson were interested in high-performance parallel structures that could be implemented directly as low-cost hardware devices [23]. They designed the *systolic architecture*. The name refers to an analogy with the regular pumping of blood by the heart, reflecting the rhythmic data movement employed by their architecture. Such an architecture is based around the Multiply And Accumlate (MAC) operation: $c \leftarrow c + a \cdot b$. This operation is implemented in a cell, which we call Processing Element (PE). As we will see later, in section 3.2, there are also operations that can be done with SAs, but the original only talks about MAC. At each clock cycle, the data is shifted along the registers. So at each clock cycle the three values of A, B and C can be different.

The architecture was popular in those years, since SAs are very good at performing specialized operations. The architecture seemed so promising that inter prototyped a whole super computer based on the idea of SAs [24].

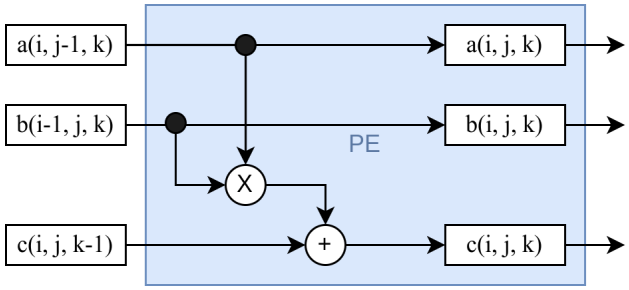


Figure 2.15: General model of PE for matrix multiplication.

Unfortunately, due to the rise of general purpose CPU and the lack of commercial success, this architecture was abandoned until the recent years.

Today they became popular in NN computing due to their regular structure and their efficiency, since the intermediate computations do not need to be written in main memory. On top of that, this architecture is eligible for novel in-memory computing practices, at its proven by the emergence of crossbar arrays [25]. As a witness of the popularity of SA there are many works which implements this architecture. For example, the university of Berkeley, in California, made their own implementation of a complete SA coprocessor for RISC-v based architectures [26]. Similarly, Google developed their own, as they realized this architecture would increase their throughput considerably [27] A similar implementation of SA can also be found in NVIDIA implementation [28]. In general, the use of SA has seen an explosion in recent years [29], [30], [31], [32].

In the next section we will see the basic inner working of a SA designed to perform matrix multiplication.

2.3.1 A Matrix-Multiplier Systolic Array

Modern iterations of SAs are generally implemented to solve the matrix multiplication, thus we speak of "matrix multiplier SA", or just SA in the context of this manuscript.

This type is a very flexible choice since matrix multiplication is among the most recurrent and expensive operations. A SA can compute the $\mathbf{R} = \mathbf{A} \times \mathbf{B} + \mathbf{C}$ equation, where $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are known matrices. To compute $\mathbf{R}$, the rows of $\mathbf{A}$ are multiplied by the columns of $\mathbf{B}$ and then an element of $\mathbf{C}$ is summed. In this way, each PE performs a single MAC operation per Clock Cycle (CC).

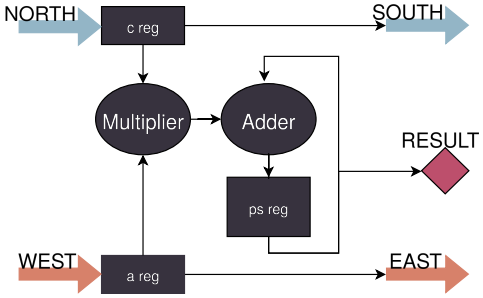


Figure 2.16: PE structure of a Output Stationary (OS) SA, showing the internal registers and the forwarding lines.

To do this, each PE has three registers, as depicted in Figure 2.16. This figure shows a PE for a OS

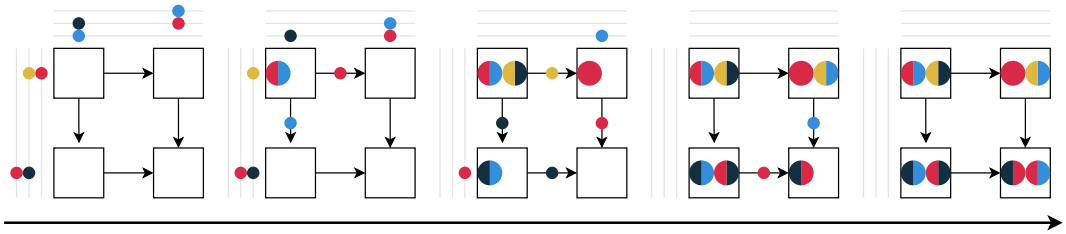


Figure 2.17: Static animation of a SA. Each depiction, from right to left, indicate a different CC.

SA. In such a SA, the PEs are arranged in a regular grid, and they are connected from top to bottom (or north to east) and from left to right (or east to south). The three registers used `a reg`, `b reg` and `ps reg`, containing respectively the values of the matrix **A**, **B** and the partial sum accumulated until that point. The last register can be initialized with the values of the matrix **C**. Every CC, three things happen in a PE:

1. the partial sum is updated: $ps \leftarrow a \cdot b + ps$,
2. the values of the register are forwarded: a towards `SOUTH` and b towards `EAST`,
3. the registers are updated with the new values: a takes the values in `NORTH`, while b takes the value in `WEST`.

To see how the PEs fit together, and consequently how the SA operates, let us look at Figure 2.17. This figure shows a *static animation* of the operation of an output stationary SA. Output stationary means that the register c is local to the PE, and it is not forwarded to other PEs. At the first cycle (leftmost depiction), the data is at the border, waiting to enter in the SA. In the next CC, the PE on the top-left performs the first MAC, preparing also the PEs on the top-right and bottom-left. Following this, in the next CC, the top-left PE performed all the needed operations, while the top-right and the bottom-left performed their own first MAC. In this CC, also the power of this architecture is shown: the bottom-right PE is ready to perform its operation and the data arrive from the neighboring PEs. This process continues until the end of the operation. In cases like this, where the data-reuse degree is so high, the PE really shines, performing the matrix multiplication in just a few CC. In fact, the SA implements Cannons' algorithm [33], which has complexity $O(\frac{n^3}{\sqrt{p}})$, where n is the length of the side of a matrix, assuming only square matrices, and p is the total number of processors available. This fact would drive us to believe that a bigger number of PEs, for big matrices, is beneficial. Nevertheless, when we take into account the latency introduced by the data movement, it is not true. In fact the function that correlates the number of PE and the time needed to obtain a result is convex [32], [34].

2.3.2 Systolic Array Dataflow and advantages

Definition: SA dataflow

In the past section we demonstrated the way the SA works using the OS dataflow. Nevertheless, as [32] points out, there are three main dataflows of systolic arrays:

- Weight Stationary (WS) - in which the weights are stationary to the PE and the accumulation operation is performed through the columns;
- OS - in which both weights and activations flow in the array in perpendicular directions, while the PEs perform the MAC operation using an accumulation register;
- No Local Reuse (NLR) - similarly to the first one, the partial sum flows through the columns, but in this case the weights have dedicated forwarding lines.

All of them have the same model of PE, as depicted in figure Figure 2.15. The difference depends on the way the registers a, b and ps are connected together[7]. For example, in Figure 2.16 the register named $c(i, j, k)$ in Figure 2.15 is the register ps in Figure 2.16. On top of that, being OS, the registers $c(i, j, k)$ and $c(i, j.k - 1)$. This aspect is discussed in details in subsection 3.2.5.

[7] Please note that the registers outside the shaded area in Figure 2.15 are meant to live in neighboring PEs

This architecture has gained so much popularity due to the wide employment of NNs: bigger NNs required more flops than what GPUs can offer. Reference [34] shows that the Google TPU has 30x the performance

This architecture has gained so much popularity due to their distinct advantages over GPUs for specific computational tasks. Unlike GPUs, which are designed as flexible, general-purpose accelerators, SAs are specialized hardware optimized for regular, data-parallel operations like matrix multiplication. This specialization allows SAs to achieve higher energy efficiency and throughput for tasks such as CNN [35] inference and training. GPUs, while powerful, suffer from overhead due to their general-purpose architecture, including instruction fetch, decode, and branch prediction—mechanisms that are unnecessary in the fixed, rhythmic data flow of systolic arrays. As a result, SAs can deliver superior performance per watt for workloads with predictable, uniform data dependencies.

Another key advantage of SAs is their deterministic data movement. In SAs, data flows synchronously through a grid of PEs, eliminating the need for complex memory hierarchies and reducing memory access bottlenecks. GPUs, on the other hand, rely on shared memory and cache hierarchies, which can lead to latency and contention, especially in large-scale matrix operations. The regular, localized communication in SAs minimizes data movement overhead, making them ideal for tasks like large-scale matrix multiplication and tensor operations in deep neural networks. This deterministic behavior also simplifies programming and optimization, as developers do not need to manage memory access patterns manually.

Although these facts are true for all types of SAs, their efficiency is heavily influenced by the choice of dataflow, and it depends on the application. In WS, the weight matrix remains fixed in the PEs, while input activations and partial sums flow through the array. This approach minimizes weight movement, reducing energy consumption and improving efficiency for workloads where weights are reused frequently, such as in CNN. OS fixes the partial sums in the PE, with both weights and inputs flowing through the array. OS is beneficial for operations requiring frequent accumulation of results, such as in recurrent neural networks or attention mechanisms [36]. The choice of dataflow directly impacts performance, energy efficiency, and hardware utilization, making it a critical design consideration for SA-based accelerators.

Among the three dataflows, WS is frequently the most efficient for deep learning workloads. This is because WS minimizes the movement of weights, which are typically larger in volume and reused more often than inputs or outputs. By keeping weights stationary, the array reduces energy consumption associated with data transfer, a major bottleneck in large-scale neural networks. Studies have shown that WS can achieve up to 20–30% higher energy efficiency compared to IS and OS for convolutional layers, where filter weights are applied repeatedly to different input patches. Additionally, WS aligns well with the memory access patterns of modern deep learning models, where weights are often preloaded and reused across multiple batches or time steps. This reduces the need for frequent off-chip memory access, further improving performance. However, the optimal dataflow can vary depending on the specific layer type and model architecture. For example, IS may be preferable for layers with large input activations, while OS can be more efficient for layers with heavy output accumulation, such as in transformer models [35].

2.4 Conclusions

In this chapter we presented the background knowledge necessary to comprehend the rest of the manuscript. The first section described the basics of NNs. Firstly, we explained their principle form, having perceptrons connected together to form MLPs. Later on, we showed the basis for classification tasks, which are poured all over the manuscript. Finally, we presented CNNs, highlighting popular architectures to show the biggest advancements in modern AI history. In the second section, we explained the origin of faults in hardware, to then talk about FI as a tool to assess reliability of complex systems. Finally, we closed the section showing SFI, an improvement on FI: by exploiting the statistical properties of the fault population we can reduce the time needed and only run a FI on a subset of faults, rather than perform all the injections, systematically. In the last section we showed the architecture of the SA, starting with bits of history to proceed explaining the working principle. We saw then how the most popular type is the Matrix-Multiplier, as it has many applications in the current landscape. Finally, we showed the different dataflows that a Matrix-Multiplier SA have taking examples from the literature.

The next chapter is dedicated to simulation of SA when they are used as accelerators for NNs and subjected to faults.

References

[1] Y. LeCun et al., "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[2] OpenAI, *GPT-4 technical report*, Mar. 4, 2024. DOI: 10.48550/arXiv.2303.08774 arXiv: 2303.08774[cs]. Accessed: Sep. 18, 2025. [Online]. Available: http://arxiv.org/abs/2303.08774

[3] "Data on AI models," Epoch AI, Accessed: Oct. 10, 2025. [Online]. Available: https://epoch.ai/data/ai-models

[4] Y. LeCun, C. Cortes, and C. Burges, *MNIST Handwritten digit database*, http://yann.lecun.com/exdb/mnist/, 1998.

[5] S. Amari, "A theory of adaptive pattern classifiers," *IEEE Transactions on Electronic Computers*, vol. EC-16, no. 3, pp. 299–307, Jun. 1967, ISSN: 0367-7508. DOI: 10.1109/PGEC.1967.264666 Accessed: Sep. 17, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/4039068/

[6] W. Sarle, "Neural networks and statistical models," 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:2562349

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2012, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3065386 Accessed: Sep. 17, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3065386

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, 84â 90, May 2017. DOI: 10.1145/3065386

[9] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, Apr. 10, 2015. DOI: 10.48550/arXiv.1409.1556 arXiv: 1409.1556[cs]. Accessed: Oct. 5, 2025. [Online]. Available: http://arxiv.org/abs/1409.1556

[10] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, Dec. 10, 2015. DOI: 10.48550/arXiv.1512.03385 arXiv: 1512.03385[cs]. Accessed: Oct. 5, 2025. [Online]. Available: http://arxiv.org/abs/1512.03385

[11] C. E. Stroud, *A Designer's Guide to Built-In Self-Test*. 2002, ch. Fault Models, Detection, and Simulation.

[12] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021. DOI: 10.1109/TVLSI.2020.3048829

[13] S. Burel, A. Evans, and L. Anghel, "Mozart+: Masking outputs with zeros for improved architectural robustness and testing of dnn accelerators," *IEEE Transactions on Device and Materials Reliability*, vol. 22, no. 2, pp. 120–128, 2022. DOI: 10.1109/TDMR.2022.3159089

[14] A. Ruospo, E. Sanchez, L. M. Luza, L. Dilillo, M. Traiola, and A. Bosio, "A survey on deep learning resilience assessment methodologies," *Computer*, vol. 56, no. 2, pp. 57–66, 2023. DOI: 10.1109/MC.2022.3217841

[15] G. Di Natale, D. Gizopoulos, S. Di Carlo, A. Bosio, and R. Canal, *Cross-Layer Reliability of Computing Systems*, eng. USA, 2020, ISBN: 978-1-78561-798-0. DOI: 10.1049/PBCS057E

[16] N. Bellarmino, A. Bosio, R. Cantoro, A. Ruospo, E. Sanchez, and G. Squillero, "Investigating on Gradient Regularization for Testing Neural Networks," in *LOD 2024 : 10th International Conference on machine Learning, Optimization and Data science*, Riva de Sole, Italy, Sep. 2024. [Online]. Available: https://hal.science/hal-04604264

[17] M. H. Ahmadilivani, M. Taheri, J. Raik, M. Daneshtalab, and M. Jenihhin, "A systematic literature review on hardware reliability assessment methods for deep neural networks," *ACM Computing Surveys*, vol. 56, no. 6, pp. 1–39, Jun. 30, 2024, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3638242 Accessed: Nov. 19, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3638242

[18] A. Ruospo, E. Sanchez, L. M. Luza, L. Dilillo, M. Traiola, and A. Bosio, "A survey on deep learning resilience assessment methodologies," *Computer*, vol. 56, no. 2, pp. 57–66, Feb. 2023, ISSN: 0018-9162, 1558-0814. DOI: 10.1109/MC.2022.3217841 Accessed: Mar. 27, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10042090/

[19] R. L. Rech Junior et al., "High energy and thermal neutron sensitivity of google tensor processing units," *IEEE Transactions on Nuclear Science*, vol. 69, no. 3, pp. 567–575, Mar. 2022, ISSN: 1558-1578. DOI: 10.1109/TNS.2022.3142092 Accessed: Jun. 22, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9676631

[20] A. Lotfi et al., "Resiliency of automotive object detection networks on GPU architectures," in *2019 IEEE International Test Conference (ITC)*, ISSN: 2378-2250, Nov. 2019, pp. 1–9. DOI: 10.1109/ITC44170.2019.9000150 Accessed: Oct. 5, 2025. [Online]. Available: https://undefined/document/9000150

[21] A. Ruospo et al., "Assessing Convolutional Neural Networks Reliability through Statistical Fault Injections," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10136998

[22] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506. DOI: 10.1109/DATE.2009.5090716

[23] H. T. Kung and E. L. Charles, "Systolic arrays (for vlsi)," in *Sparse matrix proceedings*, vol. 1, 1978, pp. 1–29.

[24] T. Gross, *iWARP: Anatomy of a Parallel Computing System*. Mit Press, 1998, ISBN: 978-0-262-51912-0.

[25] X. Wang, M. A. Zidan, and W. D. Lu, "A crossbar-based in-memory computing architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 4224–4232, Dec. 2020, ISSN: 1558-0806. DOI: 10.1109/TCSI.2020.3000468 Accessed: Oct. 10, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9120335

[26] H. Genc et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[27] N. P. Jouppi et al., *TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings*, Apr. 20, 2023. DOI: 10.48550/arXiv.2304.01433 arXiv: 2304.01433[cs]. Accessed: Oct. 7, 2025. [Online]. Available: http://arxiv.org/abs/2304.01433

[28] nvidia, *Nvdla*, 2021. [Online]. Available: https://nvdla.org/

[29] V. Kumar, A. K. M, Y. Li, S. Shanker, and D. John, *Bare-metal RISC-v + NVDLA SoC for efficient deep learning inference*, Aug. 22, 2025. DOI: 10.48550/arXiv.2508.16095 arXiv: 2508.16095[cs]. Accessed: Oct. 7, 2025. [Online]. Available: http://arxiv.org/abs/2508.16095

[30] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017, Conference Name: IEEE Journal of Solid-State Circuits, ISSN: 1558-173X. DOI: 10.1109/JSSC.2016.2616357

[31] Z. Du et al., "ShiDianNao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, ISSN: 1063-6897, Jun. 2015, pp. 92–104. DOI: 10.1145/2749469.2750389 Accessed: Oct. 7, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/7284058

[32] V. Sze et al., "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[33] L. E. Cannon, "A cellular computer to implement the kalman filter algorithm," AAI7010025, Ph.D. dissertation, Montana State University, USA, 1969.

[34] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, Toronto ON Canada: ACM, Jun. 24, 2017, pp. 1–12, ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246 Accessed: Sep. 2, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3079856.3080246

[35] T. Raja, *Systolic array data flows for efficient matrix multiplication in deep neural networks*, Oct. 29, 2024. DOI: 10.48550/arXiv.2410.22595 arXiv: 2410.22595[cs]. Accessed: Oct. 11, 2025. [Online]. Available: http://arxiv.org/abs/2410.22595

[36] A. Vaswani et al., *Attention is all you need*, Aug. 1, 2023. DOI: 10.48550/arXiv.1706.03762 arXiv: 1706.03762[cs]. Accessed: May 29, 2024. [Online]. Available: http://arxiv.org/abs/1706.03762

CHAPTER 3

Fault Injection on Systolic Array

Having understood that Systolic Arrays (SAs) are essential for Neural Networks (NNs), given all of their advantages, in this chapter we will describe our first contribution: a novel Fault Injection (FI) strategy, dedicated completely to SA. This strategy gave birth to a tool which we call FLAUERS (Faulty Logic Assessment Unit for Enhancing Reliability of Systolic systems). From here on, the tool will be referred using lowercase letters, as in "flauers", pronounced as in *flowers*. The tool is completely free and open source and available at the following link `https://github.com/jspkay/flauers`. Although concerned with FI, it would be more precise to define this tool as "an error injector", since the way faults are injected is analogous to seeing their effect at software level. This choice is done as the main objective of our simulations is to study functional reliability.

This chapter is structured as follows: section 3.1 motivates the basis of this work, section 3.2 gives some background knowledge about SAs in Artificial Intelligence (AI), proceeding with showing the mathematical formalization needed for this technique, section 3.3 ties together all the concepts, showing the basis for the FI process, going over the details of such implementation and finally section 3.4 shows some results obtained with the employment of this tool.

3.1 Motivation

Given the complexity of NN and the fact that they are oftentimes treated as black boxes, one of the main techniques employed to study the reliability of NNs relies on FI. As explained in subsection 2.2.2, there are three families of FI methods: hardware-based, platform-based and simulation-based. Given that the first two are very expensive, and in general not easily accessible, this work focused on simulation-based techniques. More specifically, we can distinguish between hardware-agnostic and hardware-aware simulations.

Hardware-Agnostic In hardware-agnostic simulations, faults are injected into the NN model without considering the specifics of the underlying hardware. This approach is flexible and widely used for initial reliability assessments, but it may not capture hardware-specific vulnerabilities or provide accurate reliability metrics for real-world deployment [1], [2]. These techniques are useful for rapid prototyping and general robustness testing, but they can yield misleading results when evaluating software hardening strategies, as the abstraction level may not reflect actual hardware behavior. On top of that, although these techniques are very useful especially for studying explainability, they fail in the assessment of hardware level faults, as the variables available are only those of the

software implementation of the NNs. Typically, it involves simulating bit-flips in network weights or activations, which are often represented in formats like 32-bit floating-point (FP32).

Hardware-Aware Hardware-aware techniques take into account the specifics of the hardware platform and is more accurate than the other type. Faults are injected at lower abstraction levels, such as the instruction set architecture (ISA) or Register Transfer Level (RTL), which allows for more accurate reliability and resilience assessments. These methods are essential for safety-critical applications, as they provide insights into how faults propagate through the actual hardware and can guide the design of more robust systems. These approaches are also used to develop and validate fault-tolerant architectures, such as redundant or error-correcting memory systems, and to optimize the placement and scheduling of computations to minimize the impact of faults [1].

Although it is impossible to study the effect of faults in SA in a hardware-agnostic simulation, it is still worth mentioning some techniques and tools, as they are the basis for this kind of study and offer ideas that are effectively reused by hardware-aware methodologies. Most notably, we find PyTorchFI [3] and TensorFI [4]. These two tools allow the injection of NN models that are built using PyTorch and TensorFlow respectively, the industry standard tools for NNs. While the first tool allows injections in the weights of the models, the second one allows perturbation in any *TensorFlow operator*. Although these tools are generally hardware-agnostic, we might frame such injections as hardware-aware for Graphical Processing Unit (GPU) and Central Processing Unit (CPU) faults. Another noteworthy strategy is LLTFI [5]. This tool takes a model in PyTorch or TensorFlow and generates the machine code to execute the model on CPU. While doing this, faults are generated manipulating the intermediate representation of the machine code.

In the context of this manuscript, there was a need to study the NNs while being executed (at least partly) in the SA. For this reason, we aimed at using a hardware-aware simulation-based FI technique. In the literature, the only true way of doing this is by using ad-hoc simulators or valid Hardware Description Language (HDL) models. For example, we find that the majority of works are carried out using HDL models, either simulated at RTL level or ran via FPGA [6], [7], [8], but also simulation at gate level [9], [10], [11], [12] are not uncommon. A minority of works use ad-hoc simulators. Among the simulators, we can cite SCALE-sim [13], which has been used in [14] to implement a proper fault injector and also in [15], which used it to perform FI, independently. Another simulator worth mentioning is S-SIM [13], which implements a complete SA in C++, but has never used for FI to our knowledge.

Authors in [16] implemented their Deep Neural Network (DNN) and the fault injector in software, inspired by an FPGA-based DNN accelerator. Moreover, in [17], DNN and FI are implemented in Keras, and the architecture of a SA accelerator is considered for a fault-tolerant design. Similarly, authors in [18] evaluate their proposed reliability improvement technique on memories in TensorFlow while injecting transient faults into the weights. PyTorch is used in [19] to implement the DNN, and transient faults are injected into activations (data path or MAC units) and weights (memory) regarding the SA accelerator model. Reference [20] also uses PyTorch and injects faults by a custom framework called TorchFI to inject faults into the outputs of convolutional and fully-connected layers of the network.

The main shortcomings of all these strategies are that:

1. they require a host processor to hand over the data,
2. are very slow to simulate because of the amount of details present,
3. extra-effort must be spent to prepare the simulation for FI.

Given the state of the art, one thing becomes clear: there is no software FI framework in hardware-aware platforms. To this extent, the objective of our injection framework was to find a mid-ground between closeness to reality and speed. On one side, recent models are extremely computation-intensive, so arises the need of having a tool that is comparable with the current state-of-the-art of NN models.

On the other side, we still want to maintain a fault aware injection framework. To address these limitations, we propose a hybrid fault injection methodology tailored specifically for SA accelerators. Our approach bridges the divide between hardware-aware precision and simulation efficiency by leveraging a framework that operates at an intermediate abstraction level. By focusing on the SA's dataflow and micro-architectural features, we enable fault injection campaigns that are both computationally feasible and representative of real-world hardware faults. This hybrid technique retains the realism of hardware-specific fault models while mitigating the prohibitive costs of full RTL simulation, thereby offering a pragmatic path forward for reliability assessment in modern neural network accelerators.

3.2 Systolic Arrays in AI and their formalization

The recent developments on NNs lead to the adoption of Domain Specific Architectures (DSA) to perform the most expensive operation in NN: convolutions. The convolution is composed by many Multiply And Accumlate (MAC) operations. The implementation can be done in different ways:

- it is possible to implement it directly - this solution is the one that was firstly adopted, but not the most efficient: there was no software optimized for any architecture for this operation;
- use a Fourier transform - this was already a better option: FFTs are used extensively today since they have a great impact in the signal processing industry,
- transform it into a matrix multiplication and use BLAS[8] - this solution was ultimately the best, as also low-power devices could use this method, even when the device didn't have the special capabilities.

[8] BLAS stands for Basic Linear Algebra Subprograms. It is a deeply architecture-optimized library to perform all sort of algebraic operations, among which Matrix Multiplication (matmul).

In a short time span, matmul became the standard way of performing convolution. And the DSA developed for NNs are all based on the same concept: the SA. It was initially developed in 1978 [21] but only recently exploded in popularity. The way SAs work is described in section 2.3, nevertheless, in this chapter we will build the theoretical basis to design a SA, so it is not necessary to read that section first.

The following sections describe the formalization of a SA and how it is used in the context of NN. Specifically, we will firstly show Lowering and Lifting (LoLif), which is the class of transformation used to perform matmul rather than convolution. Later on, we will describe the mathematical framework used for the formalization: Uniform Recurrent Equation (URE) and Regular Iterative Algorithm (RIA). Then these concepts will be used to frame matrix multiplication with these formalizations. Finally, we will describe the injection facilities, make some examples and talk about performance.

3.2.1 LoLif

The name **LoLif** was chosen based on the concept of **Lo**wering and **Lif**ting. As explained above, this strategy is well known in the High-Performance Computing (HPC) field. The main idea is to use the high-performance libraries (mainly Basic Linear Algebra Subprograms (BLAS)), which are highly architecture-optimized, to execute more complex operations. In the context of NNs, it is possible to perform a convolution via matrix multiplication. This idea is not new: the mathematician Otto Toepliz already discovered this at the beginning of the last century. A Toepliz matrix is a matrix such that its diagonals are all constants:

$$\begin{pmatrix} a & b & c & d & e \\ f & a & b & c & d \\ g & f & a & b & c \\ h & g & f & a & b \\ i & h & g & f & a \end{pmatrix}$$

Without giving too much details, it is possible to construct a Toepliz matrix $\mathcal{T}(A)$ from any matrix A such that:

$$\begin{aligned} &\mathbf{A} * \mathbf{B} = \\ &\mathcal{T}(\mathbf{A})\mathcal{F}(\mathbf{B})^T = \\ &\mathcal{F}(\mathbf{A})\mathcal{T}(\mathbf{B}), \end{aligned} \tag{3.1}$$

where the transformations $\mathcal{F}(\cdot)$ correspond to flatten a matrix to a vector, effectively concatenating the rows (or the columns[9]) of such matrix. The concept of LoLif is essentially the same, but generalized to N dimensions.

[9] Whether to do it by rows or columns is irrelevant. However, it is necessary to adapt $\mathcal{T}$ accordingly.

Figure 3.1 (got from [22]) shows the idea. In this case, the authors show a convolution between three-dimensional tensors, very common in DNNs. The three dimensions are: height, width and channels. Typically, for RGB images, the height and width (in pixels) give the resolution, while the channels are three, for the three color components: red, blue and green. For grayscale images, it might be the case to have only one channel indicating the brightness of each pixel. Supposing $\mathbf{D} \in \mathbb{R}^{n,n,d}$ and $\mathbf{K} \in \mathbb{R}^{k,k,d}$, the convolution $\mathbf{R} \in \mathbb{R}^{m,m}$, with $m = n - k + 1$, is computed as:

$$R_{r,c} = \sum_{i=1}^{d} \sum_{c'=0}^{k-1} \sum_{r'=0}^{k-1} D_{r+r',c+c',i} \times K_{r',c',i} \tag{3.2}$$

The authors provide three strategies for lowering/lifting that depend on the complexity of the two operations (lowering and lifting).

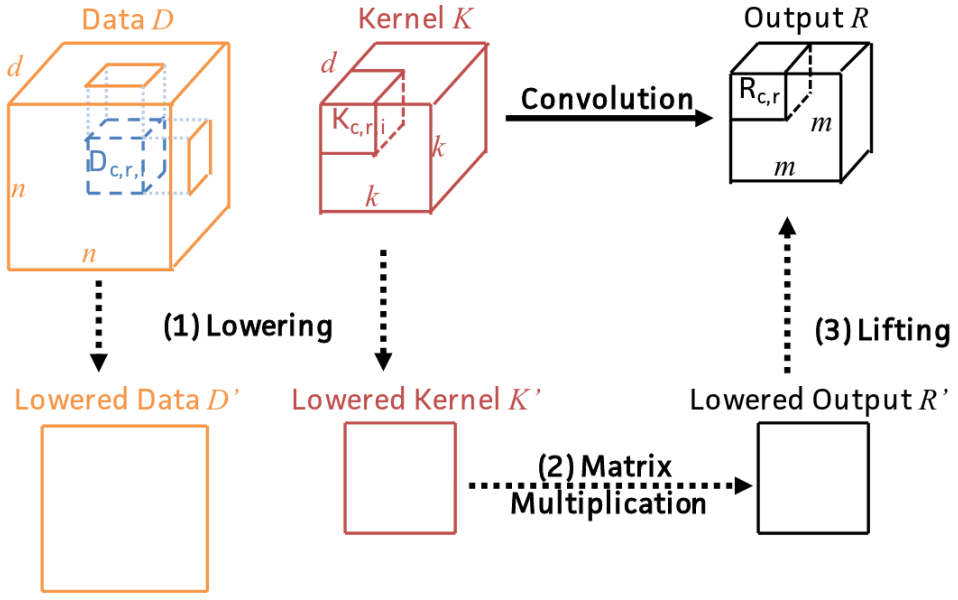


Figure 3.1: LoLif as explained in [22]. Figure taken from the same paper.

Many strategies exist [22], [23], [24], [25]. The algorithms to achieve the objective are many: a single matmul has at least three trivial ways to be performed, as pointed out it in [22]. On top of that, it

is possible to exploit the other dimensions of a convolution. As shown in Equation 2.1.3, a convolution has three dimensions, four if we consider the batch itself: the batch, the channel, the height and the width. Depending on the transversal order, many alternatives can be found.

Probably the most famous example is *im2col*. This algorithm rearranges the blocks of a matrix into columns. This algorithm is also the only one correctly implemented on flauers. Nevertheless, it only takes care of two-dimensional matrices. It is possible to also consider the channel (as shown before) and also the batch. Each of these strategies have its own strengths and weaknesses. For example, in the context of embedded systems it makes little sense to process the images in batches, since this would mean increasing the latency between inferences. Also note that some strategies may provide more resilience than others.

3.2.2 Uniform Recurrent Equations

During the 80s systolic architectures were among the most popular in the research community. Many works were developed in that period. Many authors proposed different systolic architectures able to perform different algorithms. For example, the authors in [26] show a linear systolic array able to perform string (or pattern) matching, and a pentagonal systolic array that perform band matrix multiplications. These implementations have a common factor: the problem to solve can be formalized as a URE systems. Such formalization was given by Karp, Miller and Winograd in [27] in 1967.

To formally define such a system, we need to consider the lattice L_n of points p in the n-dimensional Euclidean space E_n. The points p all have integer coordinates. We are interested in solving a system of functions $a_1(p), a_2(p), \cdots, a_m(p)$ for all points $p \in R$, where $R \subseteq L_n$. The functions $a_i(p)$ have the following form.

$$a_i(p) = f_i(a_{i_1}(p - w_{i_1}), a_{i_2}(p - w_{i_2}), \cdots, a_{i_k}(p - w_{i_k}), \tag{3.3}$$

where $i = 1, \cdots, m$. The points $p - w_{i_k}$ belong to L_n. The vectors w_k are constants independent of p and this is why they are said to have *uniform dependence*. Each equation $a_i(p)$ depends on the points $p - w_{i_k}$.

The authors of [28] showed a strategy to design a systolic array starting from a problem to solve. Specifically, the authors explain three steps:

1. find a URE system for the problem to solve,
2. find a timing function compatible with the dependencies of the URE system,
3. find an allocation function to map the URE onto a finite architecture.

Even if not said by the authors, the paper shows the strategy for a relaxed problem, in which all the functions but one are basically delays:

$$\begin{aligned} a_1(p) &= f(a_1(p - w_1), \cdots, a_m(p - w_p)), \\ a_2(p) &= a_2(p - w_2), \\ &\vdots \\ a_m(p) &= a_m(p - w_p) \end{aligned} \tag{3.4}$$

We now describe a more general approach that exploits RIAs.

3.2.3 Regular Iterative Algorithms

The concept of RIA was introduced in the paper [29]. Although the authors don't formalize the concept, they give some features that a RIA should have, specifically:

1. each variable is expressed in the single assignment format,
2. each variable is identified by a label and an index vector,
3. the direct dependencies among variables are regular.

Given these features, the authors explain that all the UREs systems are RIAs. They also point out that a clear distinction between multivariate and univariate RIAs, as done by the original coiner author of the systolic array [30]. Furthermore, the authors give an algorithm (somewhat similar to the one in [28]) to design a systolic array given a RIAs. We will briefly explain the algorithm since it is the base of flauers.

Let us give some non-complete definition of a RIA. In general, we have V labels, each for a different variable. The variables' equations constitute together a system with the following form.

$$\begin{aligned} a_1(p) &= f(a_1(p - w_{1,1}), \cdots, a_V(p - 2_{1,V})) \\ a_2(p) &= f(a_1(p - w_{2,1}), \cdots, a_V(p - 2_{2,V})) \\ &\vdots \\ a_V(p) &= f(a_1(p - w_{V,1}), \cdots, a_V(p - 2_{V,V})) \end{aligned} \tag{3.5}$$

Where p is the index S-dimensional vector and each a_i is a labeled variable. In the simplest case $V = 1$. The index vector define an *iteration space* S_i. Each point in S_i is associated with a single computation of the problem to solve. Each labeled variable identify a dependence graph since each iteration p is dependent on some other computations in $p - w_{x,y}$. The vector $w_{x,y}$ identify the data flow, since the computation flows from $-w_{x,y}$ to $w_{x,y}$, from the point of view of node p.

Now we will frame the matrix multiplication as a RIA. This step will allow us to define the fault injection framework.

3.2.4 Matrix Multiplication

Given two matrices $A \in \mathbb{R}^{N_1,N_3}, B \in \mathbb{R}^{N_3,N_2}$ we are interested in the problem of computing $C \in \mathbb{R}^{N_1,N_2}$ s.t. $C = A \times B$. Each element c_{ij} of matrix C is computed as the sum of multiple products:

$$c_{ij} = \sum_{k=0}^{N3} a_{ik} \times b_{kj} \tag{3.6}$$

where a_{ij} corresponds to the element of A in the i-th row and the j-th column.

The RIA associated with this problem is a three-variate one, with the following equations:

$$\begin{aligned} c(i,j,k) &= c(i,j,k-1) + a(i,j-1,k) \times b(i-1,j,k) \\ a(i,j,k) &= a(i,j-1,k) \\ b(i,j,k) &= b(i-1,j,k) \\ &\text{initial conditions} \\ a(i,0,k) &= a_{ik}, \quad \forall i,k \\ b(0,j,k) &= b_{kj}, \quad \forall k,j \\ c(i,j,0) &= 0, \quad \forall i,j \end{aligned} \tag{3.7}$$

the index vector is $p = (i, j, k)$, with

$$\begin{aligned} 1 \leq & i \leq N_1, \\ 1 \leq & j \leq N_2, \\ 1 \leq & k \leq N_3. \end{aligned}$$

This index vector defines a three-dimensional space S_i which we call *iteration space*. Please, note that we need to take into account a special case for the initial conditions, in which the value 0 for the components of s is allowed. The dependence graph is three-dimensional, as there are three variables. Although visualizing the 3D dependence graph is a hard task on a 2D electronic screen (or piece of paper), it is almost impossible to reproduce the whole graph, for this reason Figure 3.2 shows a single node.

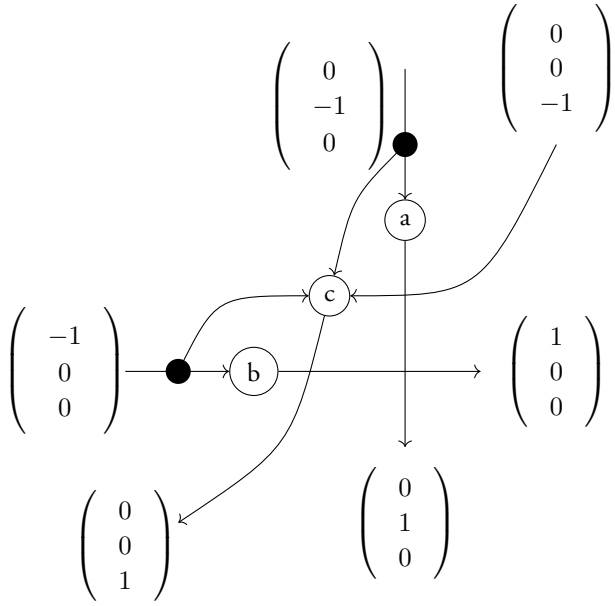


Figure 3.2: Local dependency graph for the matrix multiplication RIA

It is possible to see that (as show in the equations of system) how the data flows from node to node. Specifically, the computation of $c(p)$, with $p = (i, j, k)$, is dependent on $c(i, j, k-1)$ which is shown in the figure with the vector $(0, 0, -1)$, and forwards the computation to the next iteration $p + (0, 0, 1)$. In the same way, the values of a comes from $(0, -1, 0)$ and are forwarded to $(0, 1, 0)$. The same concept applies to the values of c. We name the dependency vectors, as they will be used later.

$$d_a = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \qquad d_b = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \qquad d_c = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{3.8}$$

3.2.5 Processing Element instantiation

In this formalization framework, the point is to now project the iteration space using a timing function (or a schedule) and a space projection. The timing function is used to determine which iterations $p = (i, j, k)$ are processed at any given point in time. Although the function could be complex, [29] states that "determining this schedule appears to be difficult" and end up choosing a linear schedule which produces isotemporal planes on the iteration space S_i. Also, current designs have the same kind of

schedule. For these reason, we do the same. Thus, it follows the relationship between the iteration space and the physical time of its execution t:

$$t = \pi p = \pi \begin{pmatrix} i \\ j \\ k \end{pmatrix} \tag{3.9}$$

Thus, in this context, finding such a projection means determining a plane V onto which project S_i. The orientation of the plane determines which iterations are computed together, as the isotemporal planes are parallel to V. Note that V is perpendicular to π. The easiest choice to take $\pi = (1, 1, 1)$, as it means that every iteration p happens at time $t = i + j + k$.

The next step is to allocate the processors physically, which really means finding a vector u along which project S_i to the processor space S_p. The choice of u determines which variables "move" in space and which are stationary. subsection 3.3.4 gives a few examples to explain better the concept. More in general, the vector u is then associated with a matrix P such that $Pu = 0$. The resulting space correspond to the physical processor space. The relationship between S_i and S_p is the following:

$$z = \begin{pmatrix} x \\ y \end{pmatrix} = P \begin{pmatrix} i \\ j \\ k \end{pmatrix} = Pp \tag{3.10}$$

The ensemble of the RIA, P and t completely determines the design of the SA. The complete relationship between S_i and S_p is the following:

$$T = \begin{pmatrix} P \\ \pi \end{pmatrix} \qquad s = \begin{pmatrix} x \\ y \\ t \end{pmatrix} = Tp \tag{3.11}$$

The tool flauers provides some projection matrices associated with some vectors u. Note that the matrix P ties together the iteration space S_i and the physical space following the relationship $s = P \times p$, where s is the physical coordinate (x, y, t) of the computation associated to iteration $p = (i, j, k)$. Nevertheless, it is important to note that the choice of P is significant, since it enables different designs to arise.

The math explained in this section is already enough to simulate a systolic array: after all, the computations are the same of the RIA. This means that, for computing the convolution, it is sufficient to solve the RIA. The important passage lays in the fault propagation that will be explained in subsection 3.3.1.

3.2.6 The Physical Parameters

It should be clear, by now, that there is a tight connection between the Processing Elements (PEs) of a SA and the operations that happen for a matrix multiplication. To this extent, we can define the physical parameters of the SA as (i) the values of N_1, N_2 and N_3 and, (ii) the projection matrix P. Depending on P, the values of N_i represent either the number of PEs of one side of the array, or the length of the input buffer. It is also possible to determine the amount of clock cycles needed to finish a single matmul and that is

$$tdone = 1 + tmax - tmin. \tag{3.12}$$

Basically, $tmax$ and $tmin$ are simply the extremes of t computed using Equation 3.9. In turn, $tmin = 3$ and $tmax = N_1 + N_2 + N_3$, which are respectively the first and last Clock Cycle (CC) when a computation happens. Such values can be derived directly from Equation 3.9. The space projected in that equation is a convex hull. Consequently, $tmax$ and $tmin$ correspond to the vertices of such hull. Specifically, they are the maximum and the minimum of such space [31].

3.3 Flauers

This section goes over the main contribution of this chapter: the FI process, details and implementation.

The workflow previously explained is already enough to explain the inner working of the tool. Nevertheless, the current implementation is capable to only deal with matrix multiplications, so we will start introducing the RIA for it. Afterwards, we will see how we can treat a convolution as a matrix multiplication. Finally, the injection mechanism is described.

3.3.1 Fault injection mechanism

The previous sections explained that it is possible to simulate a systolic array implementing and solving the RIA. We will see how to inject values in a hardware-like way using the mathematical framework.

In order to identify a fault, some parameters are needed:

1. register (to choose among a, b and c),
2. fault type,
3. physical position (x, y),
4. initial and stop times t_{start}, t_{stop},

Let us discuss each parameter. The choice of (i) the register depends on which values we want to inject. For instance, when talking about the convolution, one may choose to inject the weights or the activations. Both are associated with either a or b depending on the implementation. It is possible to inject the partial sum, which is associated with c. (ii) The fault type is trickier to consider. At this level, we can only inject the values of the register before and after a CC. Nevertheless, it is possible to take into account the "history" of the register thus constructing complex fault models. As an example, it is quite easy (although not yet implemented) to introduce delay faults, by simply missing the update of a register until the next CC. In the same way, one could consider dominant bridging faults, by changing the value of one bit to be the same as some neighbor. The best way to implement fault models is to simulate the faults at lower level and then generate a compatible representation for this injector. As of the writing of this manuscript, only three types of faults are implemented: stuck-at-0, stuck-at-1 and bitflip. (iii) The physical position identify which PE to inject. The coordinates are expressed in the physical space. (iv) Finally, the times identify the CCs affected by the fault. It is important to note that the first clock cycle is different for each PE and is indeed the first CC in which the PE receives the first useful piece of data. This fact can be considered a drawback, as it is not possible to decide a specific CC in absolute values, although that is achievable with some basic math.

To correctly perform the injection, it is necessary to consider the propagation model of the fault. To achieve this, we need to find the specific iterations points (i, j, k) to inject. In general, we need to identify the set P_i of points in the iteration space S_i to inject only the involved computations. The rest of this section addresses this problem.

We highlighted that there is a direct connection between the iteration coordinate $\mathbf{p} = (i, j, k)$ and the physical coordinate $\mathbf{s} = (x, y, t)$. After the choice of the physical position (x, y) and the start time t_{start} and the stop time t_{stop}, we can compute which elements in the iteration space are affected by the fault, but first, we need to give an obvious notion regarding dependence. We explored the meaning of dependence in the iteration space in subsection 3.2.3. It should be noted that the same dependencies hold in the physical space.

For each labeled variable $i \in a, b, c$, we can compute the dependence $\mathbf{d}_i$ of the general point $\mathbf{p}$ with respect to i as shown in subsection 3.2.4. We can obtain the analog dependence in the physical space δx_i through the equation $\delta \mathbf{x}_i = \mathbf{P} \times d_i$ [31]. We will see in 12 that this relationship can also be used to infer the cell structure, nevertheless in this context, we can interpret vector $\delta \mathbf{x}_i$ as the data flow. It describes how the values are forwarded among the cell. We can use the vector $\delta \mathbf{x}_i$ to understand which

cells are affected by some injected value. A similar argument can be made with the time, using the vector **v**: $\delta t_i = \mathbf{v} \times \mathbf{d}_i$.

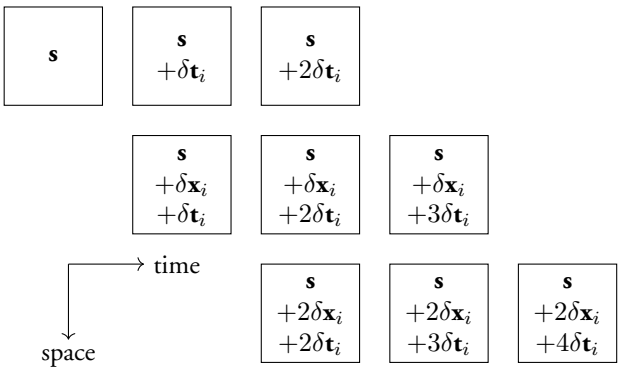


Figure 3.3: Injected elements.

Figure 3.3 shows an example of injection. In this case the fault is injected in the physical point s. The fault will affect all the points between $\mathbf{s} = (x, y, t_{start})$ and (x, y, t_{stop}), so all the points in between are also affected by the fault. This can be seen as the *time component* of the fault. Regarding the space, the same injection will affect the elements cascading from **s** in the following clock cycles: the corrupted value is forwarded to the first neighbor in the next clock cycle, thus another injected point is $\mathbf{s} + \delta\mathbf{x}_i + \delta\mathbf{t}_i$. The next clock cycle, the corrupted value will be again forwarded to the element cascading that, thus also the point $\mathbf{s} + 2\delta\mathbf{x}_i + 2\delta\mathbf{t}_i$ is affected.

Once the set of all the elements is gathered, it is possible to compute the iteration points that are affected by the fault using the inverse (usually pseudo-inverse since not square) of **P**:

$$\mathbf{p}_{injected} = \mathbf{P}^{-1} s \tag{3.13}$$

Nevertheless, mathematically this problem is not as simple, since this operation might involve non-integer indices, which do not have a meaning in S_i. To solve this issue, it is worth noting that p belongs to the physical space S_p only if $|\mathbf{p}|_1 = p_x + p_y + p_t \pmod{|\mathbf{P}|} = 0$. This has a consequence: trivially, one would assume that all the points between $\mathbf{p}_{start} = (p_x, p_y, p_{tstart})$ and $\mathbf{p}_{stop} = (p_x, p_y, p_{tstop})$ include are $\mathbf{p} = (p_x, p_y, p_t) \forall t = tstart, tstart + 1, \ldots, tstop$, but in reality it is needed to consider only those points that belong to S_p.

To this extent, algorithm 1 shows the process used to gather the points in S_p.

The first point to notice is that $tdone$ is given, as it depends on the parameters of the array. In fact as explained in subsection 3.2.6 The lines 4-6 set the initial starting time to the minimum required, which is 3. That is because, as explained in section 3.2.5, the time projection generating the isotemporal planes is always $[1, 1, 1]$, implying that the computation always start at time $t = 3$. As seen in subsection 3.2.6, this value corresponds to $tmin$. This is a small detail to facilitate the user in the usage of the tool: if the user puts $tstart = 0$, as one would expect, the algorithm corrects the intention. In line 10 the cycle for spatial propagation begins. Every point $\mathbf{p} = (xstart, ystart, zstart)$ is checked to ensure it belongs to S_p. This is achieved by explicitly computing S_p and verifying **p** is inside it. Lines 14-18 are the temporal propagation: the user decides the extremes $tstart, tstop$ and all the points assigned to the same processor $xstart, ystart$ are added to the set of points to inject $\hat{S}$. Please note that t is increased by the determinant $|\mathbf{P}|$ of **P**. As previously said, $\mathbf{p} \in S_p$ only if $p_x + p_y + p_t \pmod{|\mathbf{P}|} = 0$ Finally, lines 19-21 perform space propagation, as shown in Figure 3.3. The next processor is chosen, by incrementing $xstart, ystrart$ by d_i, which is the dependency vector as shown in Equation 3.8. In the same way, once

Algorithm 1 Gather all the points to inject

```
1:  procedure GatherPoints(i, xstart, ystart, tstart, tstop)
2:      Ŝ = ∅                                  ▷ The set of points to inject is initially empty
3:      Given tdone                            ▷ This depends on the parameters of the array
4:      if tstart < 3 then
5:          tstart = 3
6:      end if
7:      while xstart + ystart + tstart mod |P| ≠ 0 do
8:          t = t + 1                          ▷ We find the first point that respect the constraint
9:      end while
10:     while (xstart, ystart, tstart) ∈ S_p do
11:         x = xstart
12:         y = ystart
13:         t = tstart
14:         while t ≤ tstop or t ≤ tdone do
15:             (i, j, k) = P^{-1}(x, y, t)^T
16:             add point (i, j, k) to Ŝ
17:             t += |P|
18:         end while
19:         (xstart, ystart) += d_i            ▷ d_i is the dependency for register i
20:         tstop += 1                         ▷ Other than the dependency, we consider the next CC
21:         tstart += 1
22:     end while
23:     return Ŝ
24: end procedure
```

the data is handed over to the next processor, also one CC will be passed, thus we increment also $tstart$ and $tstop$.

At this point, the actual injection is performed by checking whether the iteration we are computing belongs to the set of injected points and modify the values accordingly. Algorithm 2 shows the procedure for the injection, which is actually implemented as a matmul. If we were to ignore the two internal **if** blocks (lines 10-13 and 15-17), the result is a proper matmul. The parameters A and B are obviously the matrices to multiply to obtain $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. The parameter $\hat{S}$ is the set obtained with algorithm 1. Those additional lines are where the magic happens: a fault is injected[10]. The function `inject` is a generic placeholder for the fault model simulated. In fact, the fault depends on the parameter $\hat{F}_{i,j,k}$, which defines how to modify the value. For example, if we were to inject a bitflip, that parameter should specify which bit to flip.

[10] In reality, it would be more precise to call this an *error injection*, as this framework only simulates the *aftermath* of a fault.

Although outside the scope of this text, the kind of faults that is possible to inject with this framework are many. As said before, it should be possible to simulate delay faults if the function `inject` had a state. It should also be possible to simulate cross-talk if `inject` depended on all a, b and p.

3.3.2 Additional details

The following subsection is related to some additional details and perks arising from the use of this methodology, but for which no substantial use has been found, either due to interest or time constraints. Nevertheless, the author finds these aspects valuable and might be useful in the future. Section 3.3.2 shows how to determine the structure of the RIA with the cell. Finally, section 12 shows an example of automatic SA generation, given the projection matrix **P**.

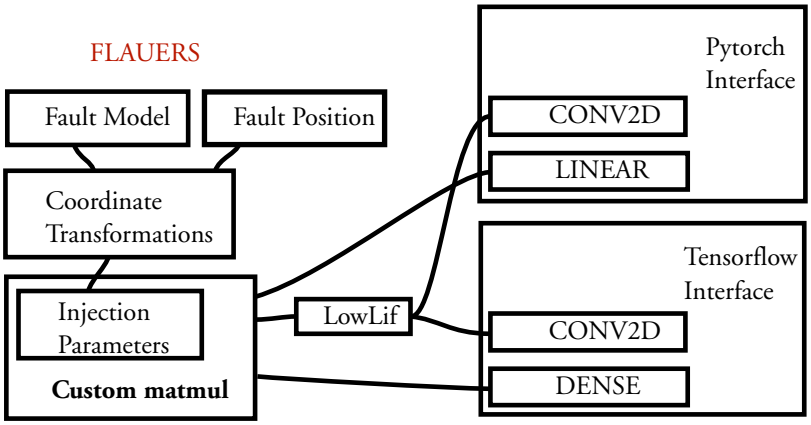


Figure 3.4: Basic components of flauers and their interconnection

Cell Structure and HDL Generation

We will now discuss how to generate a systolic array starting from the RIA described above. We could say that the array generation is done in two parts: the time-dependence and the space-dependence.

Each labeled variable of the RIA, that is $a(i,j,k)$, $b(i,j,k)$ and $c(i,j,k)$ in our case, is associated with a number of registers (in the physical model, since there is a register for a in each PE) that depends on the time-dependence between the values. In order to explain the concepts we will use the RIA associated with the matrix multiplication, available in section 3.2.3, and of particular interest for our goals. In this case we have three registers: one for a, one for b and one for c. The presence of the register is justified by the time-dependence between the values. As seen before (in subsection 3.3.1), the value of $\delta t_i = \pi d_i$ is dependent on the dependence vector d_i. We assumed since the beginning that the time projection vector v (associated with the isotemporal planes of the iteration space S_i) is always $(1,1,1)$. Since the dependence vectors are $d_a = (0,1,0)$, $d_b = (1,0,0)$ and $d_c = (0,0,1)$, the values of δt_i is always 1, for $i = a,b,c$; thus only one register is needed for each computation. Figure 3.5 shows the basic structure of the cell. It is possible to see the three registers for the three labeled values. Each register has then an input, to read the values forwarded by the previous cell, and an output, to forward the computed value to the next cell. This means that the structure of the PE is completely determined by RIA.

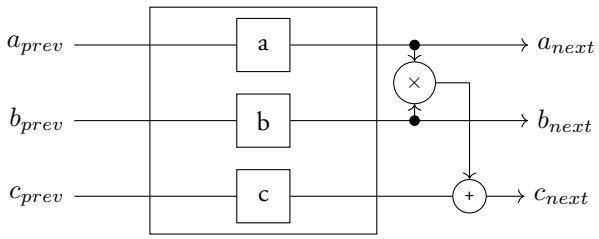


Figure 3.5: Basic cell structure

The interconnection between the PEs (i.e. the cells) is determined via the Projection Matrix **P**. Depending on the choice of P, one can have a row-stationary, weight stationary or non-local-reuse SA. The matrices are discussed in subsection 3.3.4. As anticipated in section 3.3.1, we can use the vector δx_i to infer the structure of the SA. Which means, determining the relative positions of the cells and, consequently, the positions of input and output lines of the cell. In order to do that we only need to

Algorithm 2 matmul in flauers

```
1:  procedure FLAUERSMATMUL(A, B, Ŝ, F̂)
2:      N1 = A.shape[0]
3:      N2, N3 = B.shape
4:      for i = 1..N1 do
5:          for j = 1..N2 do
6:              p = 0
7:              for k = 1..N3 do
8:                  a = A_{i,k}
9:                  b = B_{k,j}
10:                 if (i, j, k) ∈ Ŝ then
11:                     a = inject(a, F̂_{i,j,k})
12:                     b = inject(b, F̂_{i,j,k})
13:                 end if
14:                 p += a · b
15:                 if (i, j, k) ∈ Ŝ then
16:                     p = inject(p, F̂_{i,j,k})
17:                 end if
18:             end for
19:             C_{i,j} = p
20:         end for
21:     end for
22:     return C
23: end procedure
```

compute $\delta s_i, \forall i = a, b, c$. This value determines how the cells are connected, and consequently also their allocation in space. Specifically, each cell will have a neighbor for each component of δs_i and $-\delta s_i$. We can exploit this information to generate the associated HDL code. Stationary variables (i.e. those variables that are not forwarded to other elements but really stay in the same cell) are covered: simply the output and the input of that register are coincident. This is the same concept explained in subsection 2.3.2. Figure 3.6 shows the concept.

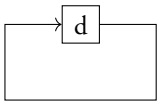


Figure 3.6: Stationary variable implementation.

Although the focus of flauers is low-cost fault simulation, the same mathematical framework can be used to generate a hardware description of the implemented systolic arrays. Furthermore, the HDL code also enables the emulation on FPGA, although the problem of coupling novel hardware with existing software is not trivial and is discussed in section 3.3.3. An example of such generation is shown in section 12

The ultimate goal for the HDL generation is a system that is able to build all the necessary pieces for a complete system: systolic array, data memory, io interface and also injector blocks[11]. In theory, these blocks should be generated starting from some formal model. The reason for the formal model is dual: firstly it enables *fast simulation*, exactly as the RIA enables the systolic array simulation by simply solving

[11] For injector blocks, we mean special HDL modules that are able to modify the behavior of the hardware in such a way to inject a fault.

Listing 1 Python code for HDL generation with Migen

```
PEs_mapping = computeCoordinateWithMatrix(P)

for p, el in PEs_mapping.items():

        # connections on line a
        p_a = el.flowing[-1][utils.LineType.a.value-1]
        if p_a is not None:
                PEs[p].a_out.eq( PEs[p_a].a_in )
        else:  # if this is None, then the associated signal is an input
                inputs[utils.LineType.a.value-1].append(PEs[p].a_in)
                input_positions[utils.LineType.a.value-1].append(p)

        # connections on line b
        p_b = el.flowing[-1][utils.LineType.b.value-1]
        if p_b is not None:
                PEs[p].b_out.eq( PEs[p_b].b_in )
        else:  # external input
                inputs[utils.LineType.b.value-1].append(PEs[p].b_in)
                input_positions[utils.LineType.b.value - 1].append(p)

        # connections on line c
        p_c = el.flowing[-1][utils.LineType.c.value - 1]
        if p_c is not None:
                PEs[p].c_out.eq(PEs[p_c].c_in)
        else:  # external input
                inputs[utils.LineType.c.value-1].append(PEs[p].c_in)
                input_positions[utils.LineType.c.value - 1].append(p)
```

a system of equations; secondly a formal model enables space exploration and might provide parameters that are usually hidden without such a model.

Systolic Array HDL generation example

Generating the HDL code using our mathematical framework is quite easy, although it should be noted that this generation does not involve the necessary hardware for the input to the array. That piece is a different story and, it will be part of future studies as explained in subsection 3.2.1.

In order to generate the HDL, although it is possible to use directly Verilog or VHDL, it is suggested to use some high-level HDL like SpinalHDL, Chisel or Migen. The generation basically consist in interconnecting the PEs in the correct way to achieve the desired results. In order to do that, the first thing to do is iterate over the space S_i and compute the "coordinates" for each PE, storing the coordinates themselves. Then, it is necessary to compute $\delta x_i, i = \{a, b, c\}$. Finally, it is sufficient to iterate over the PEs computed previously and connecting the registers according to δx_i. An example of this code in Migen is shown in the following page. This code is actually part of flauers, although it is quite useless since it is missing the IO logic.

3.3.3 Software considerations

Having such a simulator is an improvement over the current simulation strategies, nevertheless it is necessary to couple it with current software, capable of running modern NN. This side of things proved to be more challenging than expected. Modern software uses mainly python to do training, as it is easy to use and many high performance libraries exist. Most of the Machine Learning (ML) community uses either of two software packages: PyTorch or TensorFlow[12]. While PyTorch is backed by the PyTorch Foundation and develops open source software (it is in fact part of the Linux Foundation), TensorFlow is part of Google and was born in 2015. In the past years, also a third actor gained popularity: ONNX Runtime, developed by Microsoft.

[12] The two are available at the links: pytorch.org/ and www.tensorflow.org/

The challenge is to integrate flauers with these frameworks (or some of them) in order to execute the available NN library using flauers. Due to the level of abstraction of the tool, the integration was almost impossible for many tools, due the specialized software involved in the tools. So this section is dedicated at exploring these problems and potential solutions.

There are a number of challenges to discuss when trying to integrate a low-level simulator with high-level software platforms:

1. although such tools do have low level Application Programming Interfaces (APIs) to integrate novel hardware, as it as the case, they are way too specialized, and do not offer basic many flexibilities: as they sit at the lowest level of abstraction, their programming is anything but simple, making it impossible for non-low-level-programmers to access them; on the other hand, high level API are too generic, and offer very little customization;

2. another option, was to modify the current implementations of algorithms and customize the behavior - ideally, flauers would be implemented to run on the GPU, as it is faster for simulations of RIA, nevertheless, due to optimizations, the cuda kernel[13] is created at runtime, making customization virtually impossible;

3. the alternative to using the GPU is using the CPU - this path was actually implemented but ended up being unusable, due to the huge amounts of computation that the NNs require, and the time necessary to run a single inference was so long, that waiting for experiments to finish would mean doing a 7 years PhD, which is not allowed in France anyway.

[13] A cuda kernel (or shader) is a program that run on the GPU and has a specific language to program the GPU.

All the limitations just discussed are grouped under a single term: optimizations. The software described (PyTorch, TensorFlow and ONNX Runtime) have all the same objective: being used in production. This aspect makes this type of software unavailable for academia in the customization sense. To be used in production, the software must be executed as fast as possible, with the result that every optimizable aspect is indeed optimized. From our point of view, this is a limitation in that it is not possible to *penetrate* the software layers easily (or at all, depending on the case).

Nevertheless, solutions to these problems do exist.

- It is possible to simulate the whole system, including the CPU and main memory - although this is a good solution, it defeat the entire purpose of having a high-level fast simulator. This is the method used by [32]

- Alternatively, most of the software packages do allow creating a specialized layer, which is a computational unit to manipulate the data arbitrarily[14]. This method was used to integrate the original version of flauers with PyTorch and is available in the GitHub repository.

- The last viable alternative that we found to resolve this issue was to use a custom tool, which is compatible with the NN format used by the main platforms, and academia oriented, making it relatively simple to modify. This option was implemented, using a tool developed at Università

[14] Although technically true, it is not really possible that the data can be processed *arbitrarily*: in order to perform the gradient descent, it is indeed necessary to have a differentiable function when implementing custom layers, but this is outside the scope, since we are implementing a simple convolution.

di Napoli Federico II, but not publicly available yet. In any case, this iteration of the tool is not mature yet and need more work to fix the present bugs.

As a final note on the topic, we should underline a drawback of the tool as it can perform only a single matmul at a time, rendering many optimizations impossible. In conclusion, this tool is hard to categorize, as it tries to place itself between hardware and software.

Drawbacks

This section is focused on the drawbacks of such strategy for performing simulations and injection.

Firstly, the fault models that are possible to inject are very limited: in this case we are considering an extremely abstracted model of the hardware, thus many fault models associated with a finer level of detail are not accessible.

Another problem we may face is the implementation and simulations of counter-measures. It would be desirable to evaluate the same architecture after implementing some sort of counter-measures for increasing the reliability. This is feasible if and only if the counter-measure can be expressed by extending the RIA presented in 3.2.3, as the PE is seen as a succession of operations, rather than a single contained computational unit. All the counter-measures that are local to the SA are outside of reach for this method.

At the current state of the project, many details are missing. For example, the tool does not give any information about the energy used or the delay. Although theoretically implementable, some efforts should be done to achieve this goal. Furthermore, we should really take into consideration the interaction between systolic array and memory. Once again, although theoretically possible some efforts should be put in this sense.

3.3.4 Projection Matrices Examples

The choice of the projection matrix $\mathbf{P}$, as explained before, determines the topology of the systolic array completely. Remember that the choice of $\mathbf{P}$ depends on the projection vector $\mathbf{u}$. Also, the matrix $\mathbf{P}$ must satisfy the equation $\mathbf{Pu} = 0$. This is because, two computations $\mathbf{p}_1 = (i_1, j_1, k_1)$ and $\mathbf{p}_2 = (i_2, j_2, k_2)$ are mapped to the same physical cell at (x, y) if and only if $\exists\alpha : \mathbf{p}_1 - \mathbf{p}_2 = \alpha\mathbf{u}$. Because of this, $\mathbf{P}$ is not unique, thus the same projection vector can produce different systolic arrays, functionally the equivalent: the difference would be in the flow direction of the data, basically rotating the same array.

Another point to underline is the fact that the performance of the array depends on it as well. That is because, depending on $\mathbf{P}$, the individual cell may be idle for some period before executing another useful computation[15]. We will now give some examples of projection matrices.

[15] Although the meaning of *useful computation* may not be clear right now, it will be explained in the following sections.

The state of the art provides some examples of systolic arrays and different shapes and data flow. Vivienne Sze did an awesome survey about systolic arrays [33]. In their work, they explain at three different types of systolic array:

- output stationary, in which the output is a stationary variable and the accumulation is performed directly on the cell[16],
- weight stationary, in which the weights are preloaded on the cell accumulation is performed through the rows (or columns), and
- no local reuse, in which three different flows are necessary for activations, weights and accumulation.

[16] The authors of [33] actually talk about 3 different types of output stationary

The three types correspond to different projection vectors and projection matrix. We will dedicate a subsection to each.

One last note is to remember the structure of the iteration space S_i and the associations between S_i and the three labeled variable a, b, c. Specifically, one should remember that the iteration space is

three-dimensional with $\mathbf{p} = (i, j, k)$ and the three dimensions are $1 \leq i \leq N_1, 1 \leq j \leq N_2, 1 \leq N_3$. Also, due to the structure of the RIA and the dependence variables d_i, a is associated with the index j, b is associated with the index i and c is associated with k.

Output Stationary

In the output stationary type, as anticipated, the output is accumulated directly in the cell. This means that we need to project S_i along $\mathbf{u} = (0, 0, 1)$. The solutions of the equations $\mathbf{Pu} = 0$ are the following.

$$\mathbf{P} = \begin{pmatrix} p_{11} & p_{12} & 0 \\ p_{21} & p_{22} & 0 \end{pmatrix} \quad \forall p_{ij} : \quad i, j = \{1, 2, 3\} \tag{3.14}$$

Furthermore, it is necessary for the determinant of $\mathbf{T}$ to be $|\mathbf{T}| > 0$, as the determinant is used as the time unit. A time unit, in this context, represents the time interval between two successive computations. Nevertheless, in our mathematical framework, not every CC comes with a *useful computation*. This is visible from Equation 3.11, which shows that the computations are separated by $|T|$. The CC in the middle are do not synchronize the values at the PE. Nevertheless, this is a good opportunity for interleaving multiple **matmuls!** (**matmuls!**) [31]. Equivalently, we need to have a full-rank matrix P; if this was not the case, the systolic array would have some coincident cells, collapsing two dimensions on each other. Obviously, the solutions are infinite and still produce the same topology. The difference resides in the physical positions (x, y) of the cells and the vectors δx_i; more specifically, on the "distance" between two PEs. Nevertheless, as pointed out previously, all the solutions basically produce the same array.

Definition: useful computation

The simplest choice for matrix $\mathbf{P}$ is probably $p_{11} = p_{22} = 1, p_{12} = p_{21} = 0$, leading to

$$\mathbf{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{3.15}$$

The resulting systolic array is depicted in figure 3.7. It is possible to see that the values of a flow from top to bottom, while the values of b flow from left to right. The values of c are strictly tied to the cells, as we chose an output-stationary array.

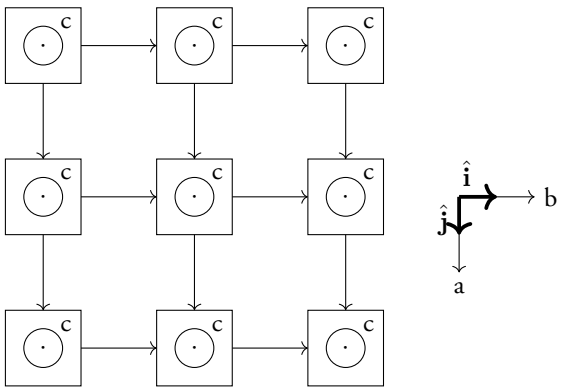


Figure 3.7: Output Stationary systolic array with $N_1 = N_2 = N_3 = 3$

We can verify the flowing of the elements via using equation 3.8:

$$\begin{aligned}\delta\mathbf{x}_a &= \mathbf{P}\times\mathbf{d}_a = \mathbf{P}\times\begin{pmatrix}0 & 1 & 0\end{pmatrix}^T = (0,1)\\ \delta\mathbf{x}_b &= \mathbf{P}\times\mathbf{d}_b = \mathbf{P}\times\begin{pmatrix}1 & 0 & 0\end{pmatrix}^T = (1,0)\\ \delta\mathbf{x}_c &= \mathbf{P}\times\mathbf{d}_c = \mathbf{P}\times\begin{pmatrix}0 & 0 & 1\end{pmatrix}^T = (0,0)\end{aligned} \tag{3.16}$$

Figure 3.7 shows the unit vectors $\hat{\mathbf{i}}$ and $\hat{\mathbf{j}}$.

Weight Stationary

In this case we need to decide whether the weights (i.e. the values of the kernel) will flow in a or b. It is not an important choice, but it would change slightly the design. Let us say the weights are the values of b. Since the index associated with b is j, we would have $\mathbf{u} = (0,1,0)$, to which the correspondent matrices satisfying $\mathbf{Pu} = 0$ are

$$\mathbf{P} = \begin{pmatrix}p_{11} & 0 & p_{13}\\ p_{21} & 0 & p_{23}\end{pmatrix}, \forall p_{ij} : i,j = \{1,2\} \tag{3.17}$$

Once again, the solutions are infinite. Furthermore, looking at this problem formally, we see that the matrix $\mathbf{P}$ is very similar to the one saw in the previous chapter. Actually, we could consider this a rotation of the previous case. Indeed, the difference between the two architecture only lies in the fact that b and c are swapped together. Figure 3.8 shows a weight stationary array with $N_1 = N_2 = N_3 = 2$.

A note to highlight is that the name *weight stationary* is sometimes used in place of *row stationary*. The difference is mainly relevant in the context of NNs, as we speak about "weights", while the problem is at all effects translated into a matmul. On the other hand, some design that are purely "weight stationary" exist, but are outside the scope of this text, since to implement them there are some more details to discuss, as the convolution is not seen as a matmul.

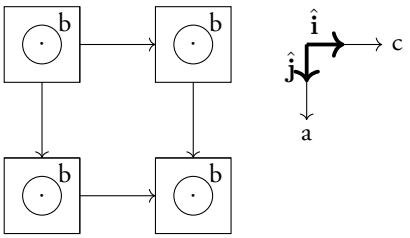


Figure 3.8: Weight Stationary systolic array with $N_1 = N_2 = N_3 = 2$

The same constraints we found before for the output stationary are applied to $\mathbf{P}$ in this case also. Finally, we could imagine an *activation stationary* systolic array that would satisfy $\mathbf{Pu} = 0$ for $\mathbf{u} = (1,0,0)$. The proof is left to the reader as an exercise.

No Local Reuse Version 1

In this configuration, there is no stationary variable, all the three variables flow in different directions. The corresponding space projection vector $\mathbf{u}$ is $\mathbf{u} = (1,1,1)$, to which the corresponding matrix $\mathbf{P}$ is the following.

$$\begin{pmatrix}p_{11} & p_{12} & p_{13}\\ p_{21} & p_{22} & p_{23}\end{pmatrix}, \forall p_{ij} \text{ such that } \begin{cases}p_{11}+p_{12}+p_{13}=0\\ p_{21}+p_{22}+p_{23}=0\end{cases}$$

We give two examples for this configuration, to show how the choice of the projection matrix changes the array.

In version 1 we have

$$\mathbf{P} = \begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \tag{3.18}$$

This projection matrix produces a hexagonal systolic array. Computing δx_i gives:

$$\begin{aligned} \delta \mathbf{x}_a &= \mathbf{P} \times \mathbf{d}_a = (1, 1) \\ \delta \mathbf{x}_b &= \mathbf{P} \times \mathbf{d}_b = (-1, 0) \\ \delta \mathbf{x}_c &= \mathbf{P} \times \mathbf{d}_c = (0, 1) \end{aligned} \tag{3.19}$$

The structure is completely inferred, thus figure 3.9 shows the architecture.

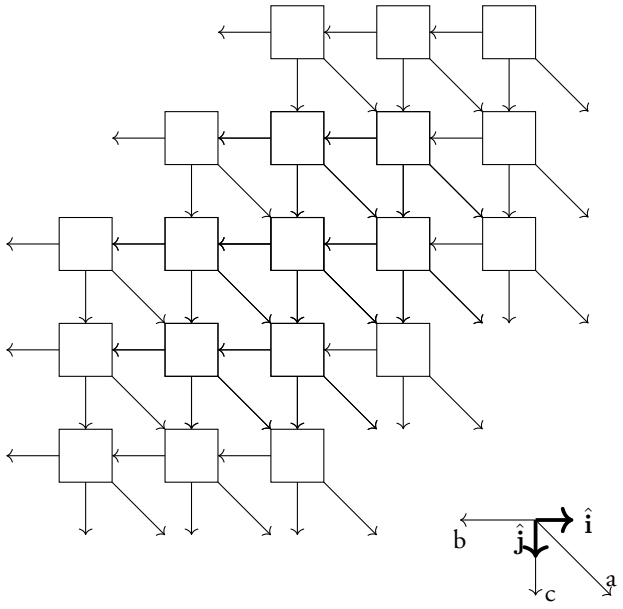


Figure 3.9: Non-local reuse Version 1

Non-Local Reuse Version 2

Version 2 produces basically the same thing, but is "rotated" in 3D. **P** will this time be as following:

$$\mathbf{P} = \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \tag{3.20}$$

It is very easy to verify that, also in this case, $\mathbf{Pu} = 0$. Nevertheless, we have different δx_i:

$$\begin{aligned} \delta \mathbf{x}_a &= \mathbf{P} \times \mathbf{d}_a = (-1, 1) \\ \delta \mathbf{x}_b &= \mathbf{P} \times \mathbf{d}_b = (0, -1) \\ \delta \mathbf{x}_c &= \mathbf{P} \times \mathbf{d}_c = (1, 0) \end{aligned} \tag{3.21}$$

The structure is essentially the same as before, and it is shown in figure 3.10.

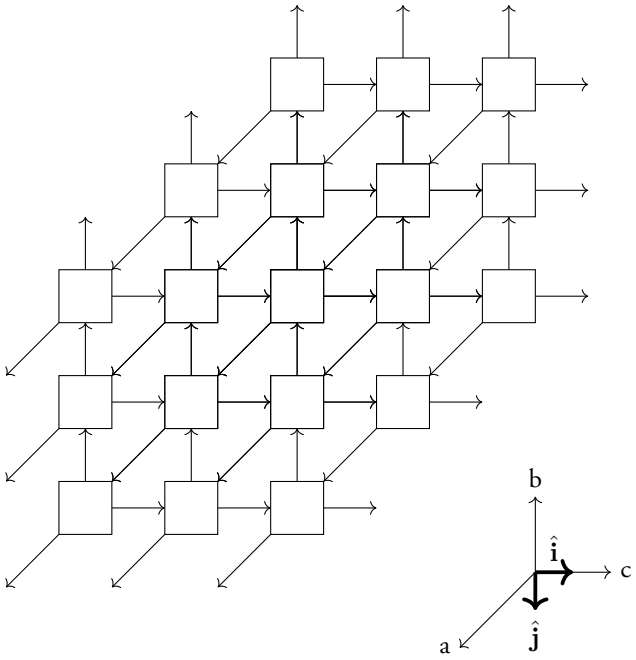


Figure 3.10: Version 2

It is obvious that the structure is basically the same. Nevertheless, the axes are the projections of a different rotation in space S_i.

Future Challenges

Many questions are yet to be answered. The following sections will highlight some problems whose research is highly likely in the following months. This section highlights the future challenges and problems to be explored. This method has potential for future research in both testability of SA and simulation framework. We will briefly discuss every point presented.

- Other Systolic Algorithms - It is in theory possible to use the same mathematics to produce systolic arrays that perform different algorithms, as long as they meet some constraints. Specifically, the algorithm must be a Regular Iterative Algorithm (RIA) as explained in [29]. One can potentially imagine a fully-blown general purpose core implemented as a Systolic Array. Although this is outside the purpose of this thesis, it is possible to imagine how flexible this strategy is. Furthermore, if the fault-tolerant strategies found are in general valid for systolic arrays (rather than be tightly related to DNNs), it may be possible to build high-resilient general purpose cores to be used in safety-critical applications.

- Systolic Array for Neural Networks - As said, systolic arrays are able to implement a quite wide class of algorithms (namely, RIAs). This means that it is possible to implement a systolic array that directly performs convolution, instead of using matrix multiplication. This approach might have an impact on the resilience, since the computations may be mapped in a completely different (somewhat new) way. Furthermore, they could implement also activation functions, directly in hardware, which would speed up the computation, thus decreasing the latency. Implementing the convolution is not a difficult task, although this option was not explored yet. The involved RIA

is the following.

$$\begin{aligned} c(i,j,k,l) =& c(i,j,k-1,l-1)+ \\ &+ c(i,j,k,l-1)+ \\ &+ c(i,j,k-1,l)+ \\ &+ a(i+k,j+l)\cdot b(k,l), \\ 1 \leq i,& j \leq m-n+1 \\ 1 \leq k,& l \leq m \end{aligned} \tag{3.22}$$

With proper initial conditions. This RIA produces a 4D space which can be projected into a tridimensional space. Future technology should allow for 3D systolic arrays.

- flauers Validation - The tool works correctly for matrix multiplications and fault injections on SA. Nevertheless, it was not formally validated. To achieve such a goal, the idea is to compare actual HDL simulations to the results obtained with the tool. The problem is that this method needs the HDL code of the array. A future version of the tool will be able to produce such a code, that can be in turn used to validate the injection process of the systolic array. Furthermore, this tool should allow the simulation and generation of other components than just a systolic array, so that to have a complete description of a system.

- LoLif Strategy Choice - As seen before, there exist some LoLif strategies that enable the convolution to be performed as a matrix multiplication. Future research will investigate whether the strategy choice influences the resilience and in what measure. Potentially, it would be possible to increase the resilience of the hardware basically for free, just using a different mapping for the computations. Furthermore, it is in theory possible to choose some custom mapping that avoid some PEs. Such a method could be used for excluding faulty PEs from the computations, virtually reconfiguring the systolic array to avoid faulty computations to happen.

3.4 Effect of the Dataflow on Resilience

Flauers is very useful to study different aspects of the resilience and reliability of NNs with contained costs. To this extent, the most important study we carried out using flauers is related to the effect of the Dataflow on the resilience of NNs. We carried out the study with and without tiling.

Tiling

Tiling is an optimization technique used in computing to enhance the efficiency and performance of algorithms, particularly those involving large-scale data processing tasks such as matrix operations in neural networks. The fundamental idea behind tiling is to divide a large computational task into smaller, more manageable sub-tasks, or "tiles." This approach aims to improve data locality, optimize memory access patterns, and increase cache utilization, which collectively lead to better overall performance. Furthermore, tiling can be exploited to cope with the constraints of the hardware, as it is the case in this study. More specifically, it often occurs that the computational workload exceeds the physical capacity of a SA. Subdividing the problem into smaller problems enables the use of the same hardware.

In this context we will only explain tiling applied to matrix multiplication. Let us assume the computation of $R = A \times B$, with A having dimensions $n_1 \times n_3$ and B having dimensions $n_3 \times n_2$. We could characterize the matrix multiplication by the triple (n_1, n_2, n_3). Assuming a SA constrained to process matrices with dimension (N_1, N_2, N_3) with $n_i > N_i$, we could proceed computing R with the following three fundamental steps:

1. **Partitioning** - A and B are subdivided in smaller sub-matrices a_i, b_i, such that the matrix multiplication has size (n_1, n_2, n_3).
2. **Independent processing** - Each sub-multiplication is processed separately $r_{ij} = a_i \times b_j$.
3. **Combining results** - The results r_i are combined to form R. Specifically, combination is formed by sum and concatenation.

Let us give an example of a $(4, 4, 4)$ matrix multiplication with a SA of $(2, 2, 2)$.

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix}$$

From here, we compute the following sub-matrices, padding with 0 if necessary:

$$\begin{aligned} a_1 &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} a_2 = \begin{pmatrix} A_{13} & A_{14} \\ A_{24} & A_{24} \end{pmatrix} a_3 = \begin{pmatrix} A_{31} & A_{32} \\ A_{41} & A_{42} \end{pmatrix} a_4 = \begin{pmatrix} A_{33} & A_{34} \\ A_{43} & A_{34} \end{pmatrix} \\ b_1 &= \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} b_2 = \begin{pmatrix} B_{13} & B_{14} \\ B_{23} & B_{24} \end{pmatrix} b_3 = \begin{pmatrix} B_{31} & B_{32} \\ B_{41} & B_{42} \end{pmatrix} b_4 = \begin{pmatrix} B_{33} & B_{34} \\ B_{43} & B_{34} \end{pmatrix} \end{aligned} \tag{3.23}$$

Then the sub-matrices can be computed separately and the resulting matrix R would be composed as follows:

$$R = \begin{pmatrix} r_{11} + r_{23} & r_{12} + r_{24} \\ r_{31} + r_{43} & r_{32} + r_{44} \end{pmatrix} \tag{3.24}$$

where the term $r_{i,j}$ corresponds to the multiplication $a_i \times b_j$

This kind of tiling takes into consideration the three parameters of theSA: height, width and the length of the input buffer queues.

3.4.1 Convolutional Neural Networks and Datasets

The Convolutional Neural Network (CNN) used as case study is a LeNet-5 model [34] composed of 7 layers: the first four are convolutional, while the last three are linear. The convolutional layers' sizes are 6@28×28, 6@14×14, 16@10×10 and the last is 16@5×5. The notation $C@H \times W$ indicates the size of the kernel with H and W, and the number of channels C for that layer We employed three versions of the CNN model, one for each used dataset, all the three versions leverage on IEEE 754 32bit floating point data type. The accuracy of the CNNs are reported in table 3.1.

Model	Classification Accuracy	Average Dataset Entropy
LeNet-5 MNIST	98.94%	1.61
LeNet-5 FashionMNIST	90.36%	4.13
LeNet-5 Cifar10	74.45%	7.26

Table 3.1: Models employed for the experiments with baseline classification accuracy.

The datasets grow in complexity in the order listed in the table. The simplest dataset is MNIST [35] and has a few white lines over a black background. Dataset FashionMNIST [36] is more complex: many

more details are captured in the same 32x32 pixels. Finally, Cifar10 [37] includes colors, which makes the classification task even harder. To measure the complexity, we computed the entropy of the datasets using Shannon's definition [38]. The entropy is computed by flattening the image into a one-dimensional vector with N components, and then applying the following formula:

$$H(\mathbf{x}) = -\sum_{\mathbf{x}\in\mathcal{X}} p(\mathbf{x}) \log p(\mathbf{x}), \tag{3.25}$$

where $p(\mathbf{x})$ is the frequency of the value x in the image and $\mathcal{X}$ is the set of all the possible values in the image. The complexity of the dataset is also visible in the performance of the CNNs: even though the neural architecture is the same and the training is performed with state-of-the-art techniques, the accuracy on FashionMNIST is almost 10%pt less than the MNIST model, and the Cifar10 model performs even worse.

3.4.2 Dataflows

The main aim of this study is to assess the effect of the SA dataflow on the resilience to hardware faults. Specifically, we inject random stuck-at-1 and stuck-at-0 faults in the SA accelerating the first convolutional layer of each network. This choice is due to the fact that the first layer is the most critical when the faults affect the inputs of the layer [39]. For this reason, we implemented six SAs: three that have dimensions tailored to our application, and three that have all the same area. Table 3.2 gives a summary.

- The three tailored SAs have the following parameters $N_1 = 28, N_2 = 28, N_3 = 160$. The parameters give different number of PEs depending on the dataflow. These three SA do not need tiling, since they perfectly fit the first layer of the CNN under test. These SAs are identified with the "-tailored" suffix.
- The other three SAs have the same area (i.e. number of PEs) but different parameters ($N1, N2, N3$), thus requiring tiling.

The three types of SA are the same as described in 3.3.4, and are called, in this context, Output Stationary (OS), Row Stationary (RS) and No Local Reuse (NLR). Figure 3.11

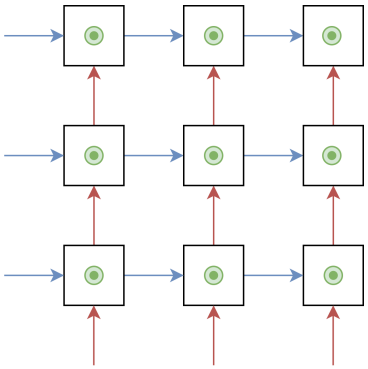
The strategy employed to implement convolution as a matrix multiplication is given in [22]. Specifically, we use the algorithm "im2col", for each convolution between two channels, simply iterating over the batch and the channels.

Dataflow	N1	N2	N3	# PEs	Relative Area (%)
OS-tailored	28	28	160	784	100
RS-tailored	28	28	160	4480	571
NLR-tailored	28	28	160	9529	1225
OS	28	28	16	784	100
RS	28	16	28	784	100
NLR	17	17	16	784	100

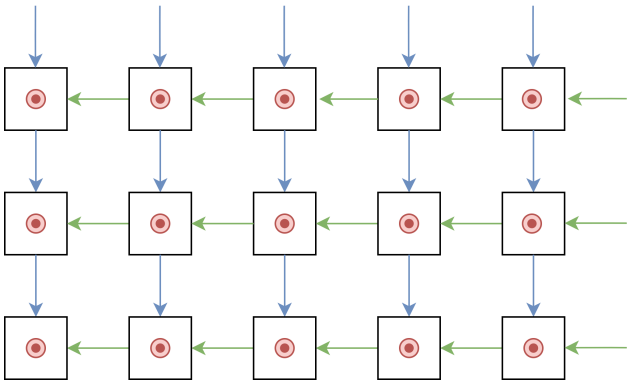
Table 3.2: Number of PEs for each dataflow.

3.4.3 Fault Injection setup

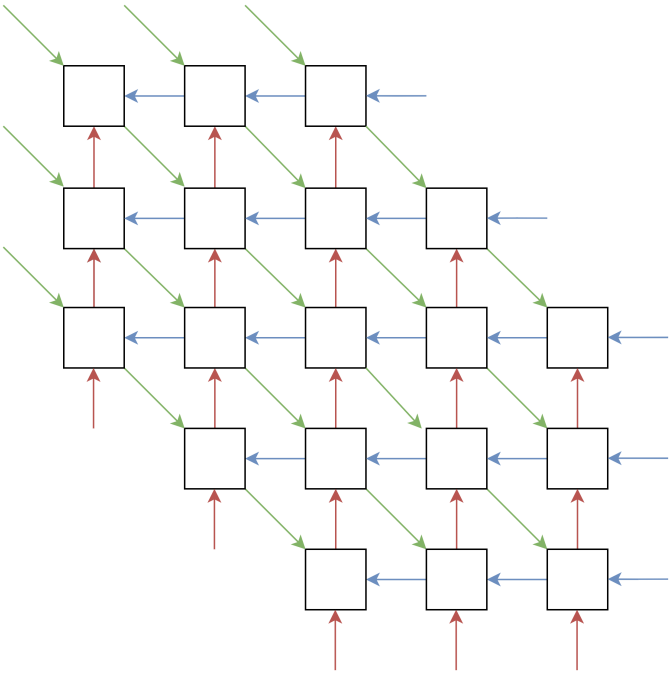
In order to evaluate the resilience of the CNNs when injecting the underlying SA, we ran statistical FI. Specifically, we injected stuck-at faults employing the (by now standard) technique described in [40]: the



(a) Output Stationary (OS)



(b) Row Stationary (RS)



(c) No Local Reuse (NLR)

Figure 3.11: The dataflows used in the experiments. The arrows indicate the interconnection between inter-cell registers: in blue the weights, in red the activations and in green the accumulations.

authors propose a formula to determine the number n of faults to inject in a hardware processor to assess its resilience. The formula is the following:

$$n = \sum_{i=0}^{7} \frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 \cdot p_i \cdot (1-p_i)}}, \tag{3.26}$$

where e is the desired error margin, t corresponds to the confidence level, N is the population size and finally p corresponds to the probability that a fault results in an error, which in our experiments corresponds to a misclassification, and it is assumed to be 0.5% (worst case scenario). In our experiments we considered a 5% error margin and a confidence level of 99% ($t = 2.58$). Table 3.3 shows the number of faults injected in each experiment. To compute N we considered all the possible stuck-at faults that could occur in the registers (inputs and output) of the PE. Specifically, N is computed as:

$$N = L \times P \times b \times 2, \tag{3.27}$$

where L are the registers that can be faulty (a, b, c as shown in figure 3.5), P is the total number of PEs of the SA under analysis, b is the number of bits per each register and finally 2 corresponds to the polarities that the fault can have, namely "high" polarity, when the bit assumes value 1 at logic level, and "low" when the bit assumes value 0. This follows from the fact that we injected stuck-at faults, thus we only have two possibility for each bit.

Table 3.3: Number of injected faults. In this table, N is the size of the fault space as described in Equation 3.27; n is the number of faults to inject to guarantee statistical significance, as described in Equation 3.26.

Dataflow	**N**	**n**
OS-tailored	968'832	666
RS-tailored	5'378'688	666
NLR-tailored	11'475'072	666
OS, RS, NLR	150'528	663

3.4.4 Fault Classification

Although this concept was already defined in subsection 2.2.3, we report it here for the convenience of the reader. The resilience of the CNN is evaluated by comparing the output probability vector of the golden run, corresponding to a fault-free inference of the CNN, and the faulty run, that is an inference with an injected fault. For each input image, two outputs are collected: the fault-free $\hat{Y}$ and the faulty output $\tilde{Y}$. They are vectors of 10 components, each corresponding to the probability of a class. The classification labels $\hat{y}$ and $\tilde{y}$ are obtained as $\hat{y} = argmax(\hat{Y})$ and $\tilde{y} = argmax(\tilde{Y})$. For the purpose of the resilience classification, $\hat{Y}$ and $\tilde{Y}$ are compared for each pair input-fault as follows:

- we call a fault **Masked** when $\hat{Y} = \tilde{Y}$,
- **Safe** when $\hat{Y} \neq \tilde{Y}$ and $\hat{y} = \tilde{y}$,
- **Critical** when $\hat{Y} \neq \tilde{Y}$ and $\hat{y} \neq \tilde{y}$.

Basically, we have either an output not affected by the presence of the fault (thus *masked*), an output affected by the fault such that the classification is correct anyway (*safe*) or an output that is affected by the fault and changes the top-1 item causing misclassification (*critical*).

Please note that in the context of this study, we evaluate the network as-is. This means that we do not take into account the case in which the faulty neural network correctly predicts a golden misclassified input.

3.4.5 Results

General Resilience

The first analysis that we performed was related to the general resilience of the CNNs. Figure 3.12 shows frequency of each fault type (masked, safe, critical), normalized to the total injected faults. The first row pertains to the experiment with the tailored SAs, showing the same trend for different dataflows. Specifically, NLR consistently shows the greatest amount of masked faults among the three architecture. Similarly, OS has the least amount of critical faults. We can conclude that there are three cases: (i) the fault affects a small portion of the computations, thus producing a masked fault; (ii) the fault affects enough computations such that the output is different, but not enough to produce misclassification, thus a safe fault; (iii) the fault affects so many computations that the difference is big enough to produce a critical fault.

The second row pertains to the experiment with tiling. In this case, different dataflows (with the same number of PEs) exhibit the same behavior. When tiling, the same fault affects each tile in the same way, thus rendering a fault inherently more critical. It is shown that the OS has a slight advantage, presenting the least amount of faults consistently. We can conclude here a key result of this paper: the SA dataflow does not affect fault resilience when using tiling.

Register-specific Resilience

In this section we assess whether different registers are more critical than others. To do so, we measured the number of critical faults per injected register, with results in table 3.4. The table shows the ratio r_i/t where r_i is the number of critical faults that were injected in register i and t is the total number of critical faults. Register c is the most critical, showing consistently the highest percentage of critical faults. Recalling figure 3.5, we see that c holds the partial sum of any given computation. The faults in this register affect the output directly, while it is not the case for the other registers. This implies that a fault in the MAC's inputs is less critical because it may be masked during the operation. Implementing selective TMR [41] by targeting this register would significantly improve the resilience.

This result is coherent with the trends in the previous section. OS is the only dataflow that does not propagate the partial sum to other PEs, thus a fault affecting register c remains limited to a single PE, decreasing the chance of criticality. RS is a middle-ground between the other two: larger than the OS, but forwarding the partial sum; these two factors make it the most susceptible to failures, showing the highest amount of critical outcomes.

Bit-specific Resilience

We analyzed the distribution of critical faults with respect to the injected bit. Recalling that the CNN is implemented with IEEE 754 32-bit floating point values, we noticed that the most critical bits were the exponent bits. As expected, the Most Significant Bits (MSBs) were the most problematic. Figure 3.13 shows the bit criticality for each register. We see that for registers a and b, only bit 1 (exponent's MSB) was predominantly critical. Further analysis confirmed that a stuck-at-1 fault in that bit is likely to produce Not a Number (NaN) values, which propagate all the way until the output of the CNN. In our experiments, all the stuck-at-1 faults injected in that bit resulted in NaN output. Conversely, stuck-at-0

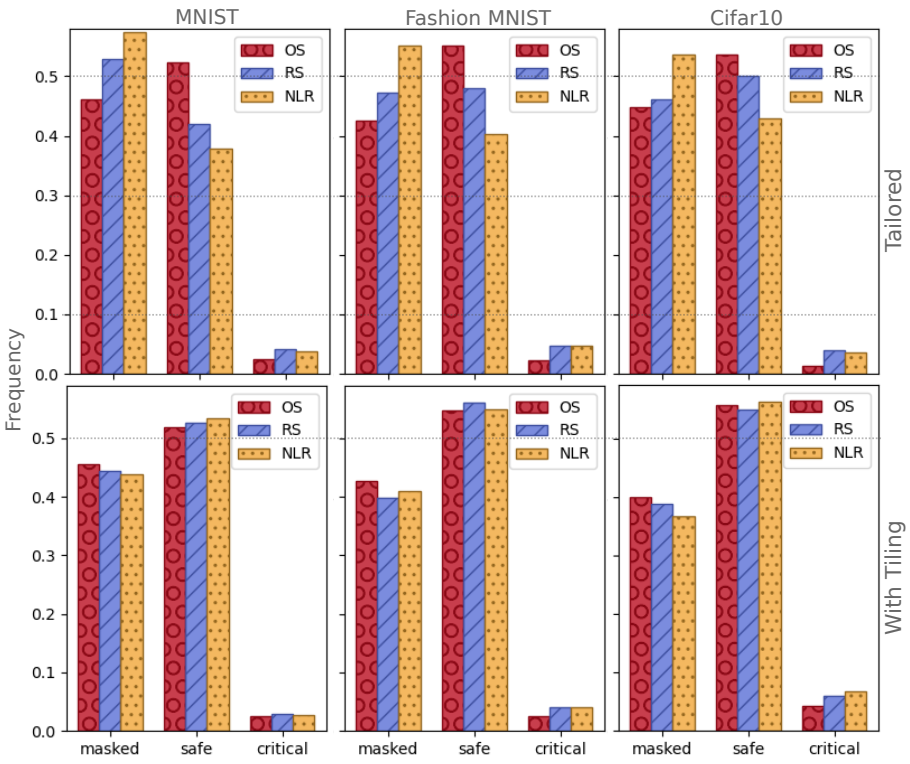


Figure 3.12: General Resilience Results

Dataset	Dataflow	Register		
		a	b	c
MNIST	OS	10.7	16.5	**72.8**
	RS	9.0	11.5	**79.5**
	NLR	9.8	14.1	**76.2**
Fashion	OS	14.7	24.5	**60.8**
	RS	7.8	15.5	**76.7**
	NLR	10.5	13.5	**76.0**
Cifar10	OS	19.7	28.4	**51.9**
	RS	11.4	25.2	**63.3**
	NLR	26.7	18.5	**54.8**

Table 3.4: Percentages of critical faults with respect to the injected register.

fault did not affect the output significantly. This result shows that we could again use TMR [41] only on this bit selectively, rather than implementing it for the whole register.

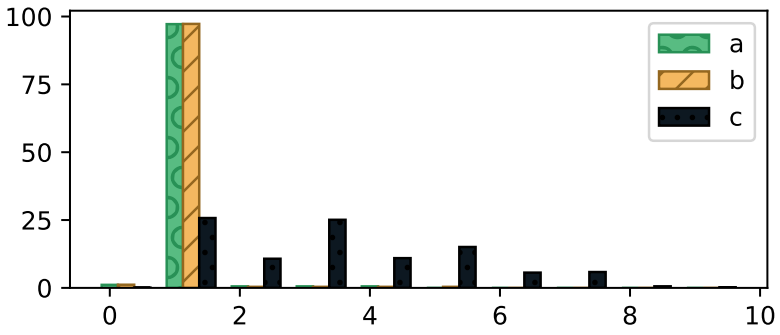


Figure 3.13: Criticality of bits

Misprediction confidence

A prediction is based on the highest value of the output vector of the CNN. Specifically, the predicted class corresponds to the $argmax$ of the output. Applying $softmax$ to the output gives us an estimation of the *certainty* of the network of its prediction. One would expect that, in presence of fault, misprediction happens to those inputs for which the network's certainty is already small, producing an uncertain misprediction in turn. We show that this is not the case. Figure 3.14 shows the frequency of the certainty in for critical faults on Cifar10. We see that about half of the critical faults are due to NaN values. This is an artifact of the format and are due to specific faults, as explained in the previous section. The other half is dominated by 100% confidence. We observed the same trend also on the other dataset. More specifically, the simpler the dataset the more confident the network is. This aspect is particularly interesting: it could give insight on the explainability of neural networks. For this reason, it will be part of future investigations.

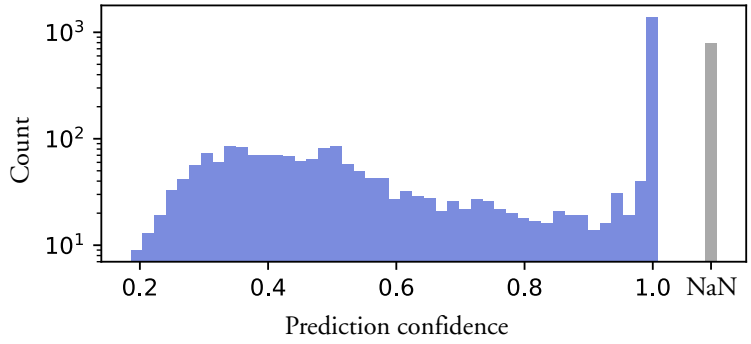


Figure 3.14: Frequency of critical faults' misprediction certainty on Cifar10.

3.5 Conclusions and Future Perspectives

All of these results hold because we used size-tailored SAs and a specific algorithm for lowering convolutions to matrix multiplications. Nevertheless, there are a few more points to investigate. More specifically, in reality a SA is manufactured with a fixed size and the experiments we performed are not fair in terms of area and power consumption. This aspect, as underlined before, have an effect on the reliability since the number of PEs directly affects the masked faults. Nevertheless, a reality-fair comparison would require the SAs to be comparable in area and power, i.e. in number of PEs, and this aspect comes with new challenges: in order to compute a bigger matrix than the limits of the SA, it is necessary to introduce tiling (or folding) which is the technique of decomposing the computation in smaller independent computations that can be processed separately and then composed together to obtain the wanted result. Tiling would affect not only the area of the SA. A key aspect is the reutilization. Reusing faulty PEs would introduce multiple fault points in the same computation, potentially producing a less reliable SAs. On top of that multiple tiling schemes exist [42], producing potentially different levels of reliability. Studying such schemes is interesting since there may exist a scheme that maximizes the reliability.

A similar discussion can be done talking about the lowering algorithms (cf. subsection 3.2.1). In fact, the way the convolution is affected by the fault is strictly dependent on the lowering strategies. As well as for the tiling schemes, multiple lowering strategies exist [22] and each have their differences. Additionally, lowering algorithms can be designed to schedule different operations on different PEs introducing an opportunity to perform online testing and online re-scheduling while the inference is being computed.

References

[1] M. H. Ahmadilivani, M. Taheri, J. Raik, M. Daneshtalab, and M. Jenihhin, "A systematic literature review on hardware reliability assessment methods for deep neural networks," *ACM Computing Surveys*, vol. 56, no. 6, pp. 1–39, Jun. 30, 2024, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3638242 Accessed: Nov. 19, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3638242

[2] N. Bellarmino, A. Bosio, R. Cantoro, A. Ruospo, E. Sanchez, and G. Squillero, "Investigating on Gradient Regularization for Testing Neural Networks," in *LOD 2024 : 10th International Conference on machine Learning, Optimization and Data science*, Riva de Sole, Italy, Sep. 2024. [Online]. Available: https://hal.science/hal-04604264

[3] A. Mahmoud et al., "PyTorchFI: A runtime perturbation tool for DNNs," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, ISSN: 2325-6664, Jun. 2020, pp. 25–31. DOI: 10.1109/DSN-W50199.2020.00014

[4] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A flexible fault injection framework for TensorFlow applications," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, ISSN: 2332-6549, Oct. 2020, pp. 426–435. DOI: 10.1109/ISSRE5003.2020.00047

[5] A. Chan, U. K. Agarwal, and K. Pattabiraman, "(WiP) LLTFI: Low-level tensor fault injector," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2021, pp. 64–68. DOI: 10.1109/ISSREW53611.2021.00045 Accessed: Oct. 4, 2025. [Online]. Available: https://undefined/document/9700378

[6] P. Corneliou, P. Nikolaou, M. K. Michael, and T. Theocharides, "Fine-grained vulnerability analysis of resource constrained neural inference accelerators," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6. DOI: 10.1109/DFT52944.2021.9568281

[7] E. Ozen and A. Orailoglu, "Low-cost error detection in deep neural network accelerators with linear algorithmic checksums," *Journal of Electronic Testing*, vol. 36, no. 6, pp. 703–718, Dec. 1, 2020, ISSN: 1573-0727. DOI: 10.1007/s10836-020-05920-2 [Online]. Available: https://doi.org/10.1007/s10836-020-05920-2

[8] U. K. Agarwal, A. Chan, A. Asgari, and K. Pattabiraman, "Towards reliability assessment of systolic arrays against stuck-at faults," in *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S)*, ISSN: 2833-292X, Jun. 2023, pp. 230–236. DOI: 10.1109/DSN-S58398.2023.00063 Accessed: Jul. 9, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10206919/?arnumber=10206919

[9] A. Azizimazreah, Y. Gu, X. Gu, and L. Chen, "Tolerating soft errors in deep learning accelerators with reliable on-chip memory designs," in *2018 IEEE International Conference on Networking, Architecture and Storage (NAS)*, Oct. 2018, pp. 1–10. DOI: 10.1109/NAS.2018.8515692 Accessed: Oct. 4, 2025. [Online]. Available: https://undefined/document/8515692/

[10] C. Liu et al., "Hyca: A hybrid computing architecture for fault-tolerant deep learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 10, pp. 3400–3413, 2022. DOI: 10.1109/TCAD.2021.3124763

[11] D. Xu et al., "A hybrid computing architecture for fault-tolerant deep learning accelerators," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 478–485. DOI: 10.1109/ICCD50377.2020.00087

[12] J. J. Zhang, K. Basu, and S. Garg, "Fault-tolerant systolic array based accelerators for deep neural network execution," *IEEE Design & Test*, vol. 36, no. 5, pp. 44–53, 2019. DOI: 10.1109/MDAT.2019.2915656

[13] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, *SCALE-sim: Systolic CNN accelerator simulator*, Feb. 1, 2019. DOI: 10.48550/arXiv.1811.02883 arXiv: 1811.02883[cs]. Accessed: May 9, 2023. [Online]. Available: http://arxiv.org/abs/1811.02883

[14] J. Tan, Q. Wang, K. Yan, X. Wei, and X. Fu, "Saca-FI: A microarchitecture-level fault injection framework for reliability analysis of systolic array based CNN accelerator," *Future Generation Computer Systems*, vol. 147, pp. 251–264, Oct. 1, 2023, ISSN: 0167-739X. DOI: 10.1016/j.future.2023.05.009 Accessed: Aug. 31, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X2300184X

[15] K. T. Chitty-Venkata and A. K. Somani, "Model compression on faulty array-based neural network accelerator," in *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2020, pp. 90–99. DOI: 10.1109/PRDC50213.2020.00020

[16] W. Li et al., "Soft error mitigation for deep convolution neural network on fpga accelerators," in *2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, IEEE, 2020, pp. 1–5.

[17] E. Ozen and A. Orailoglu, "Low-cost error detection in deep neural network accelerators with linear algorithmic checksums," *Journal of Electronic Testing*, vol. 36, no. 6, pp. 703–718, 2020.

[18] M. Jasemi, S. Hessabi, and N. Bagherzadeh, "Enhancing reliability of emerging memory technology for machine learning accelerators," *IEEE Transactions on Emerging Topics in Computing*, vol. 9, no. 4, pp. 2234–2240, 2020.

[19] E. Ozen and A. Orailoglu, "Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3250–3262, Nov. 2020, ISSN: 1937-4151. DOI: 10.1109/TCAD.2020.3012209

[20] B. F. Goldstein et al., "A Lightweight Error-Resiliency Mechanism for Deep Neural Networks," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, Apr. 2021, pp. 311–316. DOI: 10.1109/ISQED51717.2021.9424287

[21] H. T. Kung and C. E. Leiserson, "Systolic arrays for (VLSI)," Department of Computer Science Carnegie-Mellon University, Pittsburg, Pennsylvania, Tech. Rep., 1978.

[22] S. Hadjis, F. Abuzaid, C. Zhang, and C. Ré, "Caffe con troll: Shallow ideas to speed up deep learning," in *Proceedings of the Fourth Workshop on Data analytics in the Cloud*, 2015, pp. 1–4.

[23] S. Chetlur et al., "Cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[24] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "Low-memory gemm-based convolution algorithms for deep neural networks," *arXiv preprint arXiv:1709.03395*, 2017.

[25] A. Anderson, A. Vasudevan, C. Keane, and D. Gregg, "High-performance low-memory lowering: Gemm-based algorithms for dnn convolution," in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 99–106. DOI: 10.1109/SBAC-PAD49847.2020.00024

[26] U. Weiser and A. Davis, "A wavefront notation tool for vlsi array design," in *VLSI Systems and Computations*, Springer, 1981, pp. 226–234.

[27] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *Journal of the ACM (JACM)*, vol. 14, no. 3, pp. 563–590, 1967.

[28] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," *ACM SIGARCH Computer architecture news*, vol. 12, no. 3, pp. 208–214, 1984.

[29] S. K. Rao and T. Kailath, "Regular iterative algorithms and their implementation on processor arrays," *Proceedings of the IEEE*, vol. 76, no. 3, pp. 259–269, 1988.

[30] H. T. Kung and C. E. Leiserson, "Systolic arrays (for vlsi)," in *Sparse Matrix Proceedings 1978*, Society for industrial and applied mathematics Philadelphia, PA, USA, vol. 1, 1979, pp. 256–282.

[31] I. A. Miklós, *Algorithms of informatics, chapter 12*. [Online]. Available: http://compalg.inf.elte.hu/~%20tony/Informatikai-Konyvtar/03-Algorithms%20of%20Informatics%201,%202,%20%203/Systolic30May.pdf

[32] H. Genc et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[33] V. Sze et al., "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[34] Y. LeCun et al., "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[35] C. J. B. Yann, Y. LeCun, and C. Cortes, *"The MNIST DATABASE of Handwritten Digits"*, http://yann.lecun.com/exdb/mnist/, [Online].

[36] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms," *arXiv preprint arXiv:1708.07747*, 2017.

[37] A. Krizhevsky, G. Hinton, et al., "Learning multiple layers of features from tiny images," 2009.

[38] T. M. Cover and J. A. Thomas, *Elements of Information Theory*, 1st ed. Wiley, Sep. 16, 2005, ISBN: 978-0-471-24195-9 978-0-471-74882-3. DOI: 10.1002/047174882X Accessed: Sep. 2, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/book/10.1002/047174882X

[39] A. Ruospo, E. Sanchez, M. Traiola, I. O'connor, and A. Bosio, "Investigating data representation for efficient and reliable convolutional neural networks," *Microprocessors and Microsystems*, vol. 86, p. 104 318, 2021.

[40] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*.

[41] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM journal of research and development*, vol. 6, no. 2, pp. 200–209, 1962.

[42] A. C. Yüzügüler, C. Sönmez, M. Drumond, Y. Oh, B. Falsafi, and P. Frossard, "Scale-out systolic arrays," *ACM Transactions on Architecture and Code Optimization*, vol. 20, no. 2, pp. 1–25, Jun. 30, 2023, ISSN: 1544-3566, 1544-3973. DOI: 10.1145/3572917 Accessed: Jul. 4, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3572917

CHAPTER 4

Fault Injection Optimization

This chapter explores our second contribution: the role of input workload for FI campaigns, and propose some methodologies and optimizations. These are based on strategically selecting a subset of inputs, focusing on those that are *challenging* for the NNs and the task under evaluation. By prioritizing these *difficult* samples, the proposed methodology aims to identify network vulnerabilities more efficiently. The difficulty of each input is quantified using metrics that can be computed from the output probability vectors. We conduct our experiments across two scenarios: (i) Hardware-Agnostic FI, simulating memory faults through bit-flips in DNN weights, and (ii) Hardware-Aware FI, simulating faults in a systolic array used to deploy DNN.

The chapter is structured as follows: at first, we will see the motivations for this work, later on, we will describe some background needed in the context of these experiments. Finally, the last two sections are dedicated to show the experiments performed, and the last section concludes the chapter.

4.1 Motivation

When performing FI in NNs, we need to use a dataset to perform the inference. Traditionally, the dataset used to carry out this kind of experiments were the test-set or the validation-set [1], [2], [3]: the same dataset used to assess the accuracy of the NN. Yet often the time needed to carry out the whole experiment is prohibitive. That is mainly due to the increasing complexity of NNs models and the datasets, which imposes significant computational challenges for comprehensive reliability assessments: exhaustive FI across entire datasets and all possible fault locations remains simply computationally impossible. The same is true when performing hardware-aware FI, as the time increases due to simulation. The most popular technique to cope with these limitations is surely statistical FI (cf. subsection 2.2.4), that aims to drastically reduce the number of faults considered during the campaign. Oftentimes, researchers employing FI in their NN experiments, only use a reduced subset of the dataset at their disposal, introducing noise and a luck factor. For example, [4], uses only a handful of images for their experiments. There are other examples [5], [6] that we can cite. This aspect introduces also a problem of comparison, as the sub-dataset used to perform FI vary. Although statistically it does not represent a problem, direct comparison becomes spooky, if not impossible, especially considering that the majority of the datasets are publicly available and are used by the community and reproducibility is achievable.

On top of that, we can highlight the problem of online testing NNs: at runtime, we want to ensure the correctness of operations, in order to avoid disastrous outcomes [7]. Online Testing is then the practice

Definition: Online Testing

of running tests periodically to ensure correctness of operation or discover faults. When executing a test, the system goes offline for this purpose. To this extent, tests must be a short as possible and discover as many faults as possible. NN are statistical machines in principle, since they try to generalize patterns from data rather than following predefined rules. For this reason, remain inherently difficult to evaluate, despite their deterministic execution. The "testing" introduced in this context differs fundamentally from traditional hardware testing: the primary concern is evaluating a model's robustness and generalization capabilities. Fault detection aims to identify test cases with a higher likelihood of misclassification where the network is more prone to errors due to statistical uncertainties or limitations in training data [8].

The primary strategy for identifying these cases involves using a subset of the available data, commonly known as the *test set*. In [9], it is shown that *faults* in NN are more easily detectable while using inputs lying on the decision boundary of the NN. In their study, the authors propose metrics to select samples that can correctly identify erroneous behavior (mainly misclassification) in a trained model. The metrics can then be used to prioritize error-revealing inputs in order to run tests more efficiently. In [10], authors show that uncertainty-based metrics correlate strongly with the accuracy of the DNN, indicating that such metrics can be used to identify misclassifications. The paper also shows that the metrics can be used to discriminate adversarial samples. Assessing NN performances becomes more important in the presence of *hardware* faults, where the networks behave differently than fault-free conditions. Hardware faults can induce errors during the inference process, potentially leading to incorrect predictions and reducing NNs reliability. A fault in the underlying hardware can be, to some extent, modeled as noise introduced in the parameters of the networks or in the intermediate representation that is computed during the forward pass. These perturbations may affect the probability vector associated to each input. As a result of this, a perturbed probability vector may result in a wrong inference.

In the context of *hardware faults*, only a few studies have explored strategies for ranking and selecting test set samples for FI experiments [11], [12]. Some of these have focused on crafting ad-hoc images specifically designed to detect critical faults in neural networks. For example, gradient-based adversarial techniques [13] have been extensively explored alongside. Reference [14], [15] target image classification as task, and they proposed two methodologies for selecting test images from the dataset: a model-agnostic approach and a model-aided approach. The model-agnostic method uses distance-based metrics between images, while the model-aided test selection leverages confidence prediction metrics from the network's probability vector. These methods were validated using simulated hardware faults on a TPU, but were limited to Multi-Layer Perceptron (MLP) architectures, resulting in only 35% accuracy on CIFAR-10 dataset. Additionally, their approach excludes misclassified images from the test set, potentially overlooking cases critical to fault resilience assessment in state-of-the-art DNNs used in modern computer vision tasks.

Another set of studies [16], [17] proposed techniques for test image selection aimed at fault detection. These methods share similar objectives with previous works but differ in several significant ways. First, their proposed approach is designed to create a test set for continuous online fault detection, rather than offline fault assessment. Second, the fault models they use introduce errors at a larger scale than those traditionally used in CNNs reliability assessments. Specifically, they inject faults at rates from 0.05% to 1%, affecting multiple weights simultaneously. For example, in their "worst-case scenario" they select a predefined percentage of weights and flip their MSB, while in the "random case" bits are selected and flipped across the weights according to a set fault rate. According to this, applying a 1% fault rate to a VGG16 model [18], which contains 15,253,578 parameters, involves flipping the MSB of approximately 152,536 weights in the worst-case scenario. In the random case, for an 8-bit quantized network, around 1,220,286 bits are flipped. This completely disrupts the CNNs behavior. By contrast, in our research a fault affect a single bit, making detection significantly more challenging. Thus, our work considers the very worst-case scenario in terms of fault detectability. Results from previous studies are not directly comparable with our work.

A recent study by [19] offers an innovative approach to reduce the computational load of reliabil-

ity assessments, specifically targeting Approximate Computing hardware. They employ the Gini Index Computation on the probability vector to select samples expected to be more sensitive to approximation, allowing accurate loss estimation based solely on this subset. This technique models hardware-induced perturbations in neural networks through changes in the output probability vector, an approach closely related to fault modeling in CNNs, where perturbations can result from faults in either weights or intermediate activations during forward passes.

This section presents the proposed approach to efficiently computing the resilience assessment of a given DNN and its dataset. The proposed work targets image classification tasks.

4.2 Preliminaries

Let us use the same notation given in Equation 2.1.2 and let f_θ be a generic model trained to infer N different classes with an inference function. When a sample $\mathbf{x}_t$ is fed into the model, an inference generates the likelihood vector $\mathbf{p} = f_\theta(\mathbf{x}_t)$. This vector has N components corresponding to all possible classification outcomes, such that:

$$\mathbf{p} = (p_0, p_1, \ldots, p_{N-1}), \tag{4.1}$$

where each element p_j in the vector represents the model's predicted probability that the sample x_t belongs to the class j. Because of the softmax layer Figure 6, the following condition must hold for the likelihood vector:

$$0 \leq p_j \leq 1 \; \forall j, \quad \sum_{j=0}^{N-1} p_j = 1. \tag{4.2}$$

The label $\hat{y}_t$ given by the model f_θ to the sample $\mathbf{x_t}$ is then the index of the highest score of $\mathbf{p}$, given by:

$$\hat{y}_t = j = \arg\max(\mathbf{p}). \tag{4.3}$$

The likelihood that $\mathbf{x_t}$ belongs to the inferred class $\hat{y}_t$ is given by $p_j = \max(\mathbf{p})$. If the inferred class $\hat{y}_t$ matches the true label y_t (i.e. $\hat{y}_t = y_t$), the model is said to have correctly classified the input.

The accuracy is defined as the portion of correct classifications across the entire test set. In addition to classification accuracy, it is important to consider the confidence of the model in its predictions.

Measuring prediction confidence is a complex task, and various methods and metrics have been proposed in the scientific literature. Some of these approaches estimate confidence based on the likelihood vector $\mathbf{p}$. Metrics such as the normalized Gini-Simpson index Equation 4.4, minimum entropy Equation 4.5, Error L2 Norm Equation 4.6, and cross-entropy Equation 4.7 have been shown to be effective for quantifying prediction confidence [20], [21], [22], [23].

$$I_G(\boldsymbol{p}) = \frac{N}{N-1}\left(1 - \sum_{j=0}^{N-1} p_i^2\right) \tag{4.4}$$

$$H_{min} = -log\max(\boldsymbol{p}) \tag{4.5}$$

$$\text{EL2N} = ||\boldsymbol{y} - \boldsymbol{p}|| \tag{4.6}$$

$$C(\boldsymbol{y}, \boldsymbol{p}) = -\sum_{i=0}^{N-1} y_i \log(p_i) \tag{4.7}$$

In these, $\mathbf{p} = f_\theta(\mathbf{x}_t)$ represents the likelihood vector obtained during the testing phase for the input sample $\mathbf{x}_t$, and $\boldsymbol{y}$ is the one-hot encoding of the true class y_t corresponding to that sample.

The confidence of a model can be interpreted in two ways: confidence in the predefined true label and disagreement among labels [8], [11]. In the first, the measure reflects the model's certainty about its predicted class relative to the true label in a supervised learning scenario. High-confidence predictions indicate that the model is confident in correctly classifying the sample, with a large difference between the predicted probability for the chosen class and the others. Low-confidence predictions, on the other hand, suggest that the model is uncertain or close to making a misclassification.

This concept extends naturally to the multi-class setting, where the model assigns a probability distribution over all possible classes. In this context, confidence can be interpreted through a *one-versus-other* lens, in which the prediction for a given class is contrasted against the aggregated predictions of all other classes. Formally, let $\hat{y} = \arg\max\{\mathbf{p}\}$ be the predicted class for an input sample, where $\mathbf{p}$ is the output of the softmax, and p_i, the i-th element of $\mathbf{p}$, is the probability assigned to the class i. A high confidence score corresponds to a large margin between $p_{\hat{y}}$ and $p_j = \max_{j \neq \hat{y}}\{\mathbf{p}\}$, indicating that the model strongly prefers one class, $\hat{y}$ in this specific case, over all alternatives. Conversely, when this margin is small, the model expresses ambiguity among classes, reflecting low confidence. This interpretation is central to many multi-class classification reliability analyses, as it enables quantification of uncertainty based on the separability of the predicted class from the rest of the distribution.

Confidence can also be related to the degree of disagreement among different predicted labels. This is particularly suited whether the ground truth is not available, such as in an unsupervised-learning scenario. This disagreement, captured by metrics such as entropy (Equation 4.7) or Gini-Simpson index (Equation 4.4), indicates the level of uncertainty the model has in distinguishing between classes. Higher disagreement typically signifies that the model is unsure about which class the sample belongs to, potentially leading to more vulnerable or unreliable predictions.

We evaluated metrics that account for both the model's confidence in the true label (e.g. EL2N) and its internal disagreements among predicted labels (e.g. Gini Index). By analyzing these metrics, we can better understand which samples expose the model's vulnerabilities and improve the efficiency of fault detection processes.

4.3 Error resilience in DNNs

Several contributions to the scientific literature have extensively demonstrated that classification systems are generally resilient to errors [24], [25]. This resilience is primarily due to the training algorithms that leverage datasets with a sufficiently high number of well-distributed labeled samples. This property also supports some degree of fault tolerance when considering hardware accelerators that perform the inference phase [26], [27].

Suppose a fault, or Single Event Transient (SET) occurs during the processing of a sample $\mathbf{x}_t$ from the test set, and let f'_θ represent the faulty model. The perturbed likelihood vector is then denoted as $\mathbf{p}' = f'_\theta(\mathbf{x}_t)$. The predicted class corresponding to the perturbed model is $\hat{y}'_t = \arg\max(\mathbf{p}')$ This difference can lead to one of the following three outcomes - although it was already defined in subsection 2.2.3, we report it here for the convenience of the reader:

(a) *Masked Fault* - The fault has no effect on the final prediction, and the likelihood vector remains unchanged:

$$\mathbf{p}' \equiv \mathbf{p}. \tag{4.8}$$

(b) *Non-Critical Fault* - The fault alters the likelihood vector, but the prediction remains correct.

$$\mathbf{p}' \not\equiv \mathbf{p}, \qquad \hat{y}'_t = \hat{y}_t. \tag{4.9}$$

(c) *Critical Fault* – The fault causes a significant alteration of the likelihood vector, leading to a misclassification:

$$\mathbf{p}' \not\equiv \mathbf{p}, \qquad \hat{y}'_t \neq \hat{y}_t \tag{4.10}$$

Cases (a) and (b) are typically expected due to the inherent resiliency of neural networks. We refer to these as *masked* and *non-critical* faults, respectively. In contrast, case (c) refers to *critical* faults, which are more concerning as they lead to significant deviations in the model's behavior, potentially resulting in unacceptable outcomes, especially in safety-critical applications.

The complexity of both trained models and hardware accelerators makes it challenging to characterize the element-wise difference $\boldsymbol{\epsilon}$ between the fault-free and faulty likelihood vectors, even from a statistical perspective. Characterizing these differences would require testing the model across a wide range of inputs and fault scenarios, which is computationally infeasible.

Now, consider that $\mathbf{p} = f_\theta(\mathbf{x}_t)$ and let $\hat{y}_t = \arg\max(\mathbf{p})$ represent the predicted class. The model's confidence in its prediction can be categorized into the following cases:

1. *Reliable Inference:* The model is highly confident in its prediction, with the highest probability concentrated on the predicted class, i.e. $p_j \approx 1$ and $p_i \approx 0, \forall i \neq j$.
2. *Unreliable Inference:* The model assigns similar probabilities to two classes, indicating a lack of confidence in the predicted classes, i.e. $p_j \approx p_l \approx \frac{1}{2}$, with $p_j > p_l$, and $p_i \approx 0, \forall i \neq j, l$.
3. *Degenerate Case:* The model assigns nearly equal probabilities to all classes, i.e. $p_i \approx \frac{1}{N}, \forall i$.

From these scenarios, it can be inferred that, to cause a misclassification in the model, the error introduced by the fault must be large when the model is confident (case 1). On the other hand, a smaller error may suffice in cases 2 and 3. This suggests that, for robustness analysis, using samples where the model is confident (case 1) would likely result in the fault being undetected. However, samples falling into cases 2 and 3 are more likely to trigger detectable faults, causing critical or non-critical failures. Thus, focusing on *challenging* samples – those in cases 2 and 3 – is more effective for evaluating the reliability of neural networks. This targeted approach can lead to a more efficient and comprehensive analysis of network vulnerabilities, particularly for fault detection tasks.

4.3.1 Metric-based Test Samples Ranking Method

As mentioned, earlier, conducting a full FI campaign on a DNN is often impractical due to the immense dimensionality of the fault space. To address this, Statistical Fault Injection (SFI) is typically employed: this significantly reduces the number of faults to be injected while maintaining statistical significance of results. Specifically, the number of injections can be computed by resorting to subsection 2.2.4, that allows the required number of faults to inject to decrease exponentially as the acceptable error margin increases. For instance, an accepted error margin of 1% with a confidence interval of 99% gives at least 16,640 faults injected for a highly parametrized network [7], [28], [29].

A significant challenge in traditional SFI lies in its choice of workload, that is the images used as input (usually part of the test-set). The entire test set is often used, which proves to be suboptimal as (i) the effect of a fault is often uniform across a large portion of the test set, with critical faults impacting multiple images simultaneously, and (ii) many faults are masked, producing no observable effect during inference.

Given these observations, a reduced subset of images can often capture the majority of critical faults. Nevertheless, we should discuss its effectiveness. Ideally, we would be able to compute the *Average Case*, evaluating the reliability of the NN under normal operational conditions, without introducing any bias in image selection. This scenario, however, is hard to configure, as NNs are not very well understood and often treated as black boxes. Conversely, we may consider the *Worst Case*, which focuses on the

NN's performance under the most challenging conditions, and overestimating the errors it can commit. To this extent, we propose a metric-based approach to prioritize *challenging* samples, i.e. those samples for which the *prediction confidence* is low, as indicated in section 4.3. The scientific literature suggests that these challenging samples, which activate the highest number of faults, are often associated with complex or borderline conditions [13], [15], [17], though formal proof is lacking. In section 4.3.2 we will empirically prove that such samples are crucial for uncovering latent vulnerabilities in the network.

The proposed framework for metric-based sample selection consists of the following steps:

1. **Metric Selection:** Identify a suitable metric to measure the complexity or challenging nature of the samples. Common metrics include confidence scores, Gini Index, or distance-based measures in the feature space.
2. **Inference** Perform an inference on the entire test set to compute embeddings or probability vectors for each sample, as required by the chosen metric.
3. **Ranking:** Rank the test samples based on the computed metric values.
4. **Subset Selection:** Select the best n samples from the ranked list for inclusion in the FI workload.

An important strength of the proposed workload analysis and test selection method lies in its computational efficiency, especially when compared to fault injection-based reliability evaluation. The metric used for selecting informative test samples requires only a single inference per image, making it extremely lightweight. For most neural networks, computing this metric over the entire test set takes only a few seconds on a standard GPU.

In contrast, standard FIs campaigns incur a significant computational cost. Each fault from a predefined list must be injected individually, and inference must be repeated for each fault-image pair. For example, evaluating 16'641 faults on a single image with ResNet-18 requires 16'641 inferences. Extending this to the full test set of 10'000 images results in about 166 million inferences. However, imagine that by leveraging our proposed metric, we can identify a reduced and more informative subset of, say, 1000 images. This lowers the required inferences for FI from 166 million to just 16 million, plus the 10'000 inferences needed to compute the golden outputs and the metric, achieving an order-of-magnitude reduction.

Crucially, our framework introduces very little overhead. Since fault injection inherently requires computing the fault-free (golden) output for each input as a reference, the metric can be calculated during this initial inference. This integration makes the approach in practice cost-free. In our experiments, computing the metric for 10'000 test images took under 10 seconds, while performing fault injection on a single image with 16'641 faults took over 40 minutes.

This stark contrast highlights the efficiency of our method, which enables scalable and targeted reliability evaluations without the need to exhaustively analyse the full test set. By focusing on critical samples and bypassing redundant ones, the proposed approach offers a practical and computationally viable alternative for assessing the robustness of AI systems. Consequently, the selected samples can be integrated into any existing FI framework with minimal modifications to the codebase.

A high-level pseudocode description of the aforementioned approach is provided in algorithm 3: it requires the fault list $\boldsymbol{F}$, the network model $\boldsymbol{N}$, the dataset[17] $\boldsymbol{T}$, the confidence metric m, and the number of test samples to be considered as workload n. Lines 2 to 7 build the ranked dataset $\boldsymbol{R}$: for each sample belonging to T, the inference is performed (line 4) and the sample, and the likelihood vector resulting from the model are stored in the $\boldsymbol{R}$ set for the subsequent analysis. Later on, at line 7, $\boldsymbol{R}$ is sorted based on the confidence metric m, which serves to prioritize samples with lower degrees of prediction confidence.

A fault f from the fault list is injected into the network at line 10; then at line 11, a sample is picked from $\boldsymbol{R}$, and the inference is performed at line 12. Line 13 compares the output of the faulty network against the fault-free network to ascertain whether the injected fault can be detected. In the latter case,

[17] For dataset, we generally indicate the Validation Set (valset), but in principle any kind of previously unseen data will do.

the fault is categorized as critical or non-critical, based on the predicted class (line 14). The counter of critical and non-critical faults is incremented accordingly (lines 15 and 17, respectively).

Algorithm 3 Pseudocode of the OpRA approach.

```
1:  procedure OPRA(F , N, T, m, n)
2:      R ← ∅                                   ▷ R will contain all the couples (img, p)
3:      for (img, l) ∈ T do                     ▷ For every image img and label l
4:          p ← N(t)                            ▷ Compute the probability vector p of t
5:          R ← R ∪ {(img, p)}                  ▷ Expand R with p.
6:      end for
7:      R ← sort(R, m(·))                       ▷ Sort R depending on m
8:      R ← R[: n]                              ▷ Take only the first n images
9:      for f ∈ F do                            ▷ Start the fault injection
10:         N′ ← injectFault(f, N)              ▷ Inject the fault in the network N
11:         for (img, p) ∈ R do                 ▷ Iterate over the reduced dataset R′
12:             p′ ← N′(t)
13:             if p′ ≢ p then
14:                 if arg max(p′) ≢ arg max(p) then   ▷ If there is misclassification
15:                     critical ← critical + 1        ▷ Count a critical inference
16:                 else
17:                     noncritical ← noncritical + 1
18:                 end if
19:             end if
20:         end for
21:     end for
22:     return noncritical, critical
23: end procedure
```

4.3.2 Experimental Setup

In order prove our metric-based raking for test selection method, we performed two sets of experiments: *Hardware agnostic FI* and *Hardware aware FI*. All the experiments are carried out with single fault models. We addressed a classification task considering a relatively lightweight network. In the former case, a bit-flip in one of the weights of the DNN, mimics a fault in the system memory. These errors are relatively easy to simulate, allowing the analysis of a large amount of data. However, the single bit flip scenario represents the worst case in terms of online detectability, as the resulting perturbation is often minimal and may not produce significant deviations in the network's behavior.

In contrast, hardware-aware FI considers the details of the DNN hardware accelerator. These simulations necessitate a greater computational effort compared to Hardware-agnostic FI. Therefore, an optimized test set is crucial for conducting FI. The hardware-aware experiments are not faster, if anything the opposite: for every convolution, our framework flauers (cf. chapter 3) needs to compute the Toepliz's matrix (cf. subsection 3.2.1) of the inputs and process all the tiles separately, resulting in substantial simulation times.

The experimental setup included a selection of representative and diverse CNNs commonly used in computer vision tasks for embedded systems. The focus was on lightweight networks, as their reduced memory footprint makes them particularly well-suited for deployment in resource-constrained embedded environments [30]. We considered four distinct datasets. Specifically, we considered a LeNet-5 [31], two ResNet (with 9 and 18 layers) [32], two VGGs (with 11 and 16 layers) [18] trained on both MNIST [33] and CIFAR10 [34] dataset, a MobileNetV2 [35] trained on CIFAR10, an HCGNet [36] trained on

CIFAR100 [34], and a VGG11 trained on Imagenette, a curated subset of ImageNet consisting of 10 classes [37].

An overview of the networks being considered, and their test accuracy, is given in Tab. 4.1.

Before proceeding to the discussion of the experiment, the following section will present some preliminary experiments conducted to find the most appropriate metric to be used.

Dataset	Network	Accuracy	Applied Faults	Critical Faults
Imagenette	VGG11	95.94%	16 641	115
CIFAR-100	HCGNetA2	82.40%	16 623	2756
CIFAR-10	ResNet18	93.52%	16 641	826
	ResNet9	92.17%	16 639	1508
	LeNet	79.36%	16 638	1158
	VGG11	90.10%	16 641	819
	VGG16	91.61%	16 641	894
	MobileNetV2	91.72%	16 640	1340
MNIST	ResNet18	99.63%	16 641	581
	ResNet9	99.65%	16 639	527
	LeNet	99.25%	16 638	484
	VGG11	99.68%	16 641	661
	VGG16	99.58%	16 641	555

Table 4.1: Hardware-Agnostic FI campaign summary grouped by dataset. Accuracy refers to the fault-free scenario.

4.4 Selecting adequate metrics

In order to determine the existence of any discrepancy between the metrics examined in section 4.3 and to determine which metric more effectively conveys prediction confidence, we conducted two preliminary experiments.

The first experiment is designed to investigate potential discrepancies among confidence metrics, specifically those outlined in Equations (4.4-4.7). To achieve this, the respective test sets were randomly sampled, and the metrics were examined for any correlations between them, employing the Spearman correlation index [38].

The Spearman correlation coefficient is employed for the evaluation of the relationship between two variables, measuring the strength of monotonic relationships, whether linear or not, between two variables. In the absence of repeated data values, a perfect Spearman correlation of +1 or -1 is indicative of a perfect monotone function between variables.

The results for LeNet-5, ResNet-18 and HCGNet CNNs are presented in Table 4.2, Table 4.3, and Table 4.4, respectively. As the index is close to 1 in each of the case, we can safely state that the considered metrics are equally effective for measuring the confidence during the inference phase.

While different metrics analyse *challenging* samples in distinct ways, the strong correlations among metrics imply flexibility in their selection. Depending on the specific requirements of an application or dataset, any of these metrics can be confidently employed for evaluating challenging samples.

The objective of the second experiment is to provide empirical evidence that a lower prediction confidence on a sample point, x_t, is more likely to be beneficial in identifying both non-critical and critical faults. To achieve this, we performed FI campaigns. Following the approach from [39], we

Table 4.2: Spearman correlation index between ((4.4)-(4.7)) metrics, for the LeNet-5 CNN on MNIST.

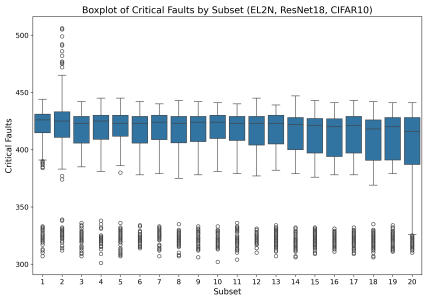
H_{min}	0.99		
EL2N	0.99	0.99	
$C(\boldsymbol{y}, \boldsymbol{p})$	0.99	0.99	0.99
	$I_G(\boldsymbol{p})$	H_{min}	EL2N

Table 4.3: Spearman correlation index between (4.4-4.7) metrics, for the ResNet-18 CNN on CIFAR-10

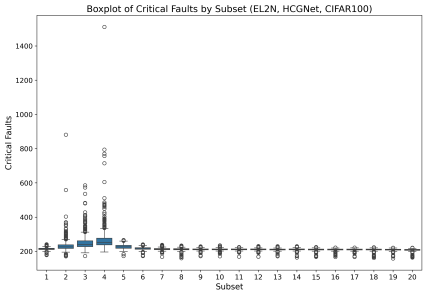
H_{min}	0.86		
EL2N	0.85	0.99	
$C(\boldsymbol{y}, \boldsymbol{p})$	0.98	0.85	0.85
	$I_G(\boldsymbol{p})$	H_{min}	EL2N

exploited Equation (2.14) to compute the number of faults to be injected to achieve 1% error and 99% confidence in the estimation. According to this, we injected about 16640 faults in each network (the precise number depends on the equation (2.14) and is shown in Tab. 4.1).

Figure 4.1-4.4 shows the number of critical faults detected for each of the images in the test dataset, varying the GINI-index (Equation (4.4)), the Cross-entropy (Equation (4.7)) or the EL2N(Equation (4.6)), respectively. Specifically, each figure shows a scatter plot in which each point is an image of the testest. Each point is associated to a metric value (in the x-axis) and a number of "detected faults" (in the y-axis), i.e. faults for which there was a misclassification of the image (with respect to the golden run). You can observe that some images have a number of detected fault much higher that the rest, and there also is a correlation with the value of the metric, as the spikes are concentrated. This happened despite the fact that all the images were processed with the same faults. It follows that some images allow more faults to be detected (that is, are more likely to be misclassified in presence of faults.) This suggests that certain samples may be better suited for fault detection, or are inherently more sensitive to fault activations. These images often correspond to specific confidence metric values, e.g. ≈ 0.5 for the Gini-Index, ≈ 1 for the Cross-Entropy, and ≈ 0.7 for the EL2N.



(a) ResNet-18 on CIFAR-10



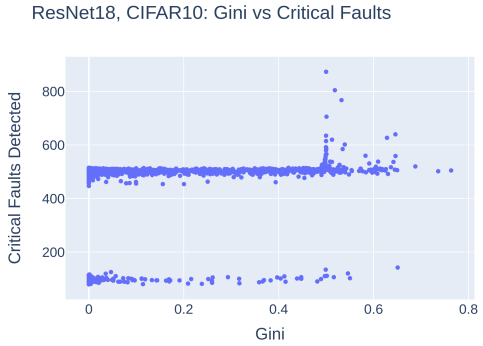
(b) HCGNet on CIFAR-100

Figure 4.5: BoxPlot of fault captured vs metric values, for a subset of 500 images sorted by confidence metric value.

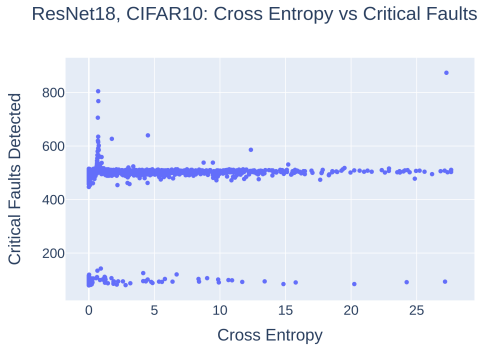
The experimental results obtained substantiate our hypothesis, which posits that as the model's prediction confidence diminishes, there is an increased propensity for misclassification, thereby leading to an elevated fault detection rate. To better illustrate this, we sorted the images based on their respective metric values and grouped them into subsets of 500 samples each. For each subset, Figure 4.5a and Figure 4.5b report the number of critical faults detected for each image in that subset. The x-axis reports the subsets in order, with the leftmost subset

Table 4.4: Spearman correlation index between (4.4-4.7) metrics, for the HCGNet CNN on CIFAR-100

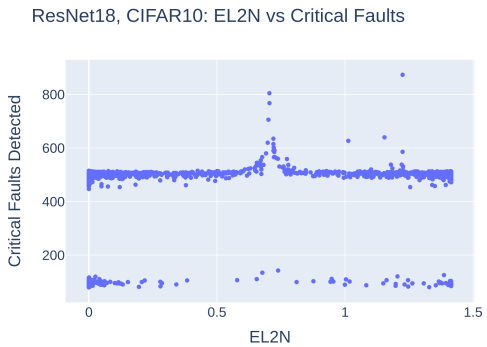
H_{min}	0.99		
EL2N	0.93	0.91	
$C(\boldsymbol{y}, \boldsymbol{p})$	0.93	0.92	0.99
	$I_G(\boldsymbol{p})$	H_{min}	EL2N



(a) GINI-Index vs Critical Faults

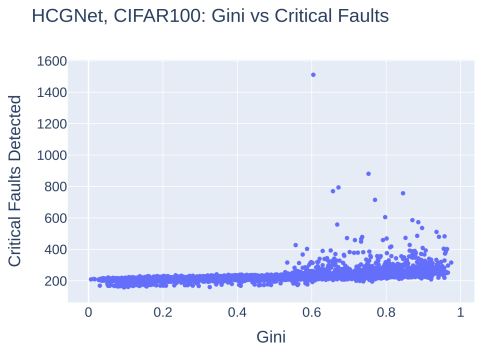


(b) Cross-Entropy vs Critical Faults

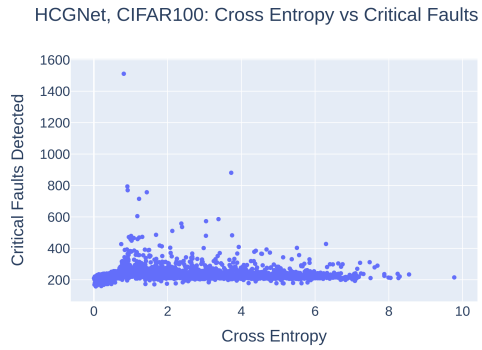


(c) EL2Nvs Critical Faults

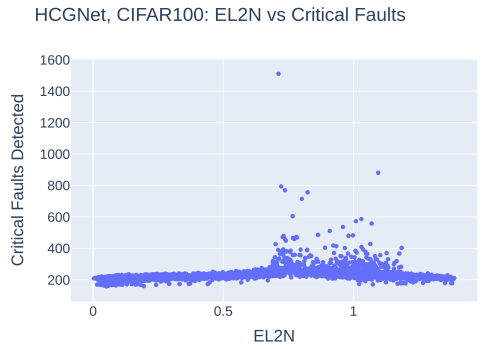
Figure 4.1: Confidence vs fault detected for ResNet-18 on CIFAR-10



(a) GINI-Index vs Critical Faults

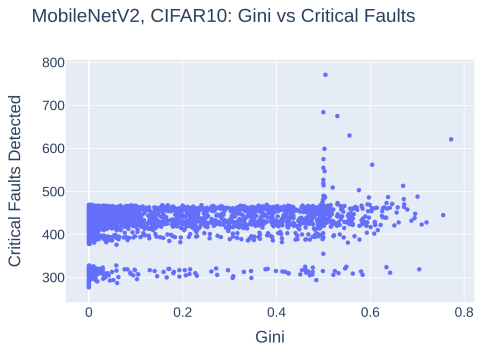


(b) Cross-Entropy vs Critical Faults

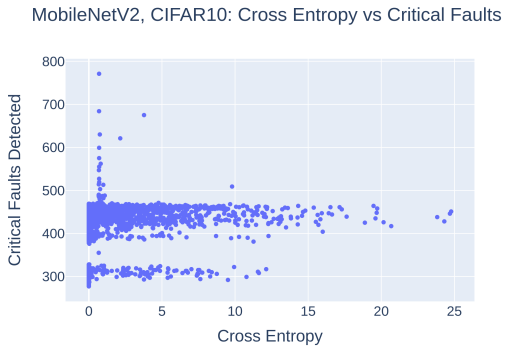


(c) EL2Nvs Critical Faults

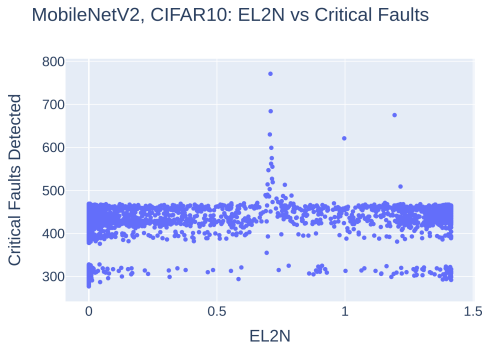
Figure 4.2: Confidence vs fault detected for HCGNet on CIFAR-100



(a) GINI-Index vs Critical Faults

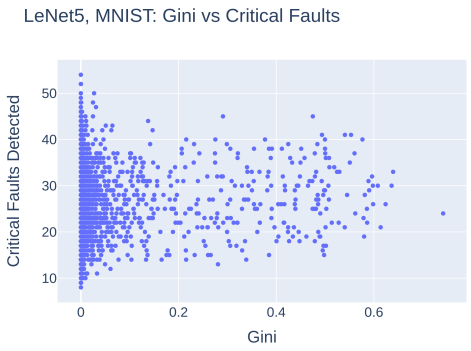


(b) Cross-Entropy vs Critical Faults

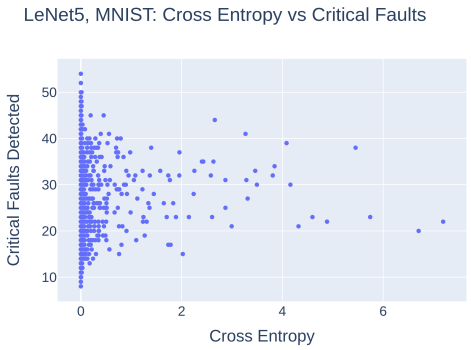


(c) EL2Nvs Critical Faults

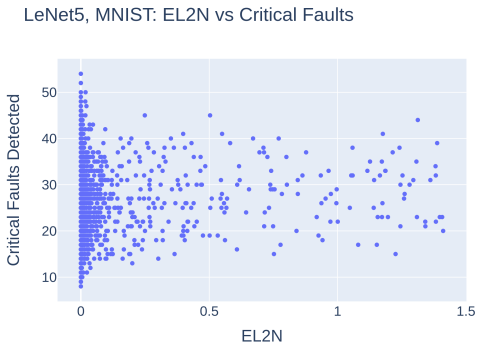
Figure 4.3: Confidence vs fault detected for MobileNet on CIFAR10



(a) GINI-Index vs Critical Faults



(b) Cross-Entropy vs Critical Faults



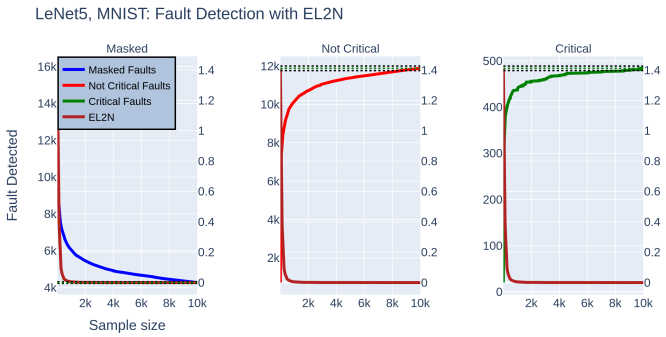
(c) EL2Nvs Critical Faults

Figure 4.4: Confidence vs fault detected for LeNet on MNIST

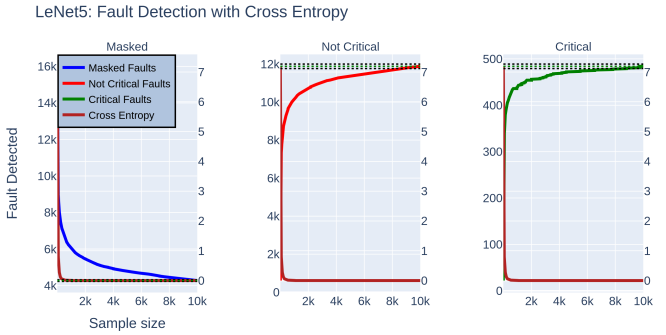
comprising test samples with the larger EL2N (4.6) value.

Despite the evident differences among the considered CNNs and datasets, a comparable tendency can be observed. The detection rate across subsets is nearly identical for non-challenging images. However, subsets that collect images with values close to the threshold ($\approx$ 0.7 for EL2N) exhibit outliers, i.e. images that activate a significantly higher number of faults compared to the typical distribution. This confirms that the prediction confidence can be exploited to identify critical samples regardless of the dataset or the CNN.

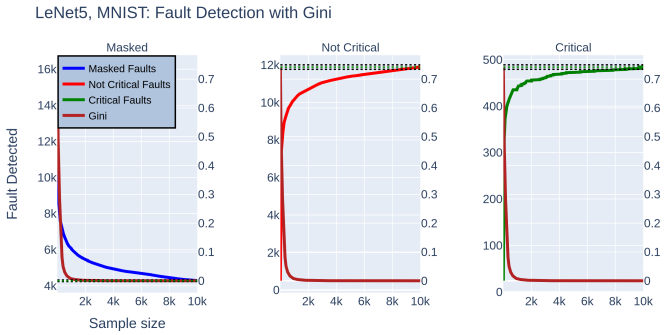
It is worth noting that the outcomes for ResNet18 may be indicative of a more optimal correspondence between EL2Nvalues and critical faults, a phenomenon that can be attributed to the relatively uncomplicated nature of CIFAR-10 and the potential for a more uniform distribution of critical samples. In contrast, the more diffuse distribution observed in the HCGNet plot might imply greater variability in the distribution of critical images, a consequence of the augmented complexity inherent in CIFAR-100. Notwithstanding the aforementioned observations, it is noteworthy that for both models and datasets, the approximate EL2Nvalue of 0.7 appears to be the most critical for images.



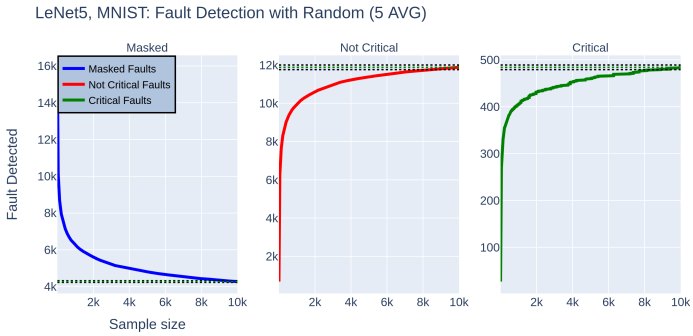
(a) EL2N



(b) Cross-Entropy

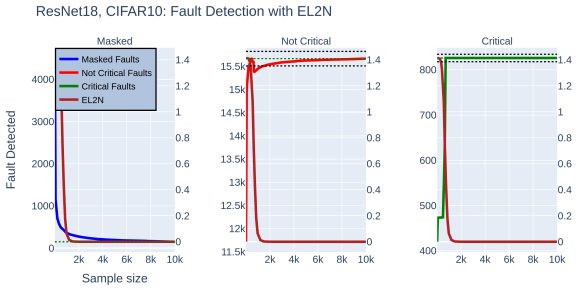


(c) Gini-Index

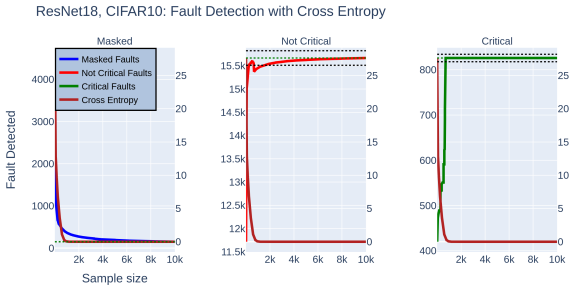


(d) Random Sampling

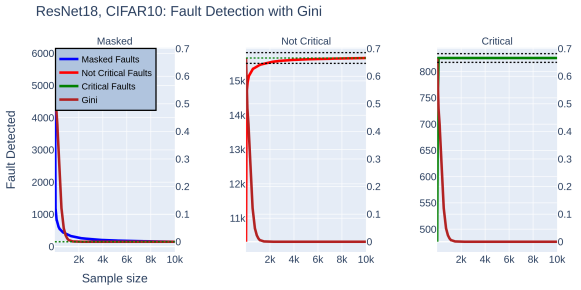
Figure 4.7: Metric values vs fault identified by each image for ResNet-18 on CIFAR-10 (Gini index, Cross Entropy and EL2N).



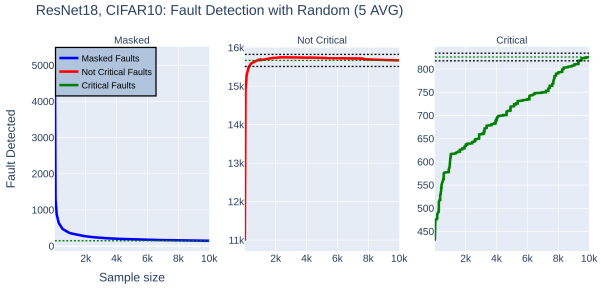
(a) EL2N



(b) Cross-entropy

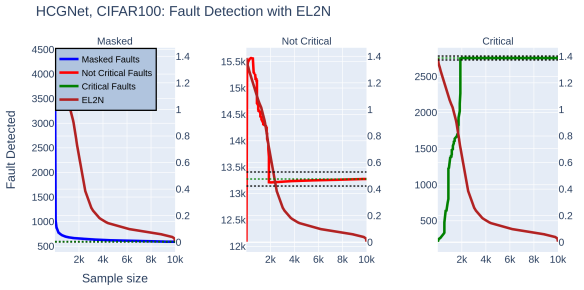


(c) Gini-Index

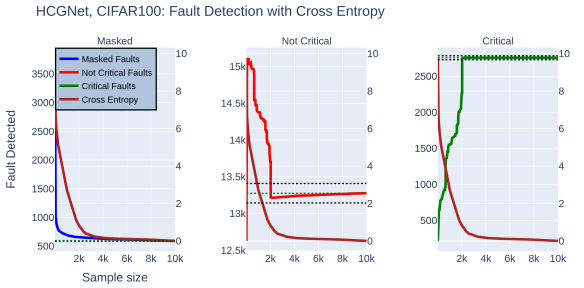


(d) Random sampling

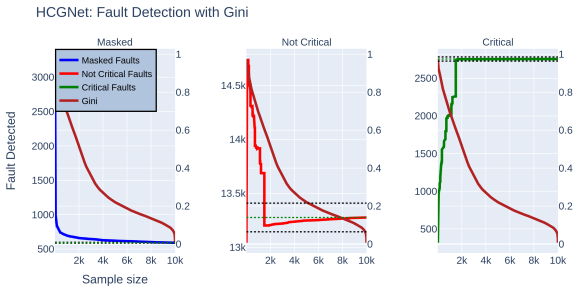
Figure 4.8: Metric values vs fault identified by each image for LeNet5 on MNIST.



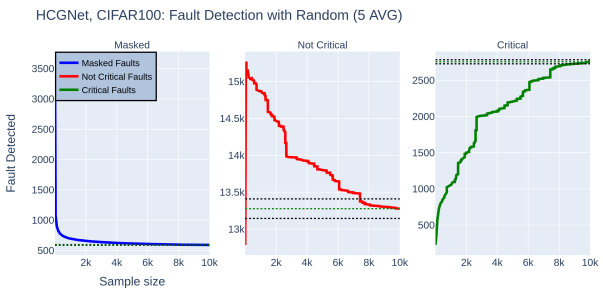
(a) EL2N



(b) Cross-Entropy

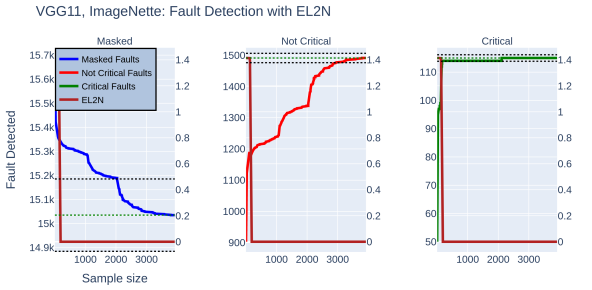


(c) Gini-Index

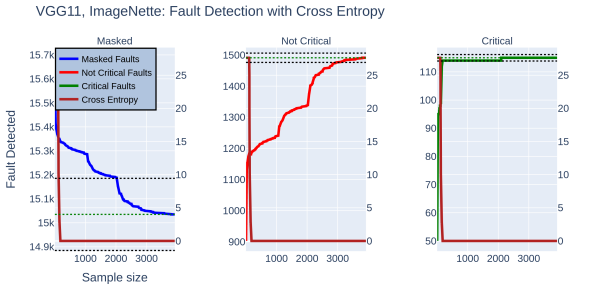


(d) Random Sampling

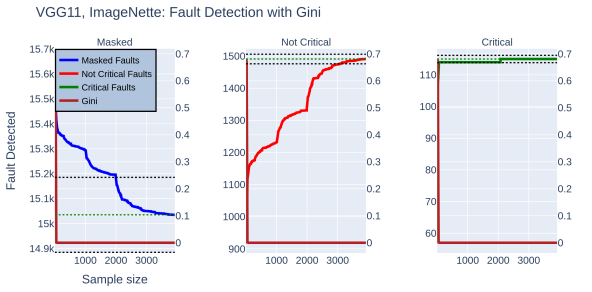
Figure 4.9: Metric values vs fault identified by each image for HCGNet on CIFAR-100.



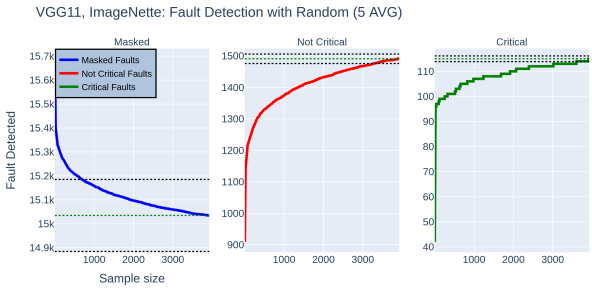
(a) EL2N



(b) Cross-Entropy



(c) Gini-Index



(d) Random Sampling

Figure 4.10: Metric values vs fault identified by each image for VGG11 on Imagenette.

4.5 Hardware Independent Experiments

We performed software FI by corrupting the weights of the model with the same parameters as shown in Table 4.1. In fact, this approach makes use of *fault detection curves*, which are plots showing the cumulative number of faults detected against the number of images used for detection. Notably, the classification of each fault is cumulative and unidirectional, meaning a fault cannot be reassigned to a less severe category once classified into a more severe one: a fault is critical even if it affects only a single image critically. These curves facilitate the comparative analysis of different metrics and their influence on fault activation and detection.

Experiments show how sorting samples based on metrics could be beneficial for this task. In our graphs, blue curves/points represent masked faults, red curves/points indicate non-critical faults, and green curves/points correspond to critical faults. Red-brick curves show the metric values. Dotted horizontal green lines represent the final values, the dotted black lines are placed at $\pm 1\%$ of the final values.

Using a random approach, nearly all images are required to detect all the critical faults. This random analysis was repeated with five different random image sorting, with the curves representing the average of the five runs.

When images are sorted and the detection algorithm is applied, we observe that after an initial transitory phase, the sorted metric values quickly approach zero and then stabilize at a *plateau*.

It is evident that all images activating critical faults fall within the transitory phase of the curve. Once the critical faults stabilize (i.e. all critical faults are detected), the metric values approach zero or quickly decrease, and the plateau is reached. This behavior is consistent across all metrics computed at the probability vector level; not all samples are required for detecting critical faults; rather, only the *challenging* samples seem sufficient to find all the critical faults.

Regarding the number of *challenging* samples to deploy, it heavily depends on the dataset and network: for the ResNet18 on CIFAR10, less than 100 test samples are enough to fully discover all critical faults. For the LeNet architecture, instead, about 8'000 samples over 10'000 are necessary to discover the 95% of the critical faults. However, this network is heavily resilient, and there are no considerable outliers in the distribution of images that discover critical faults.

Table 4.5 provides an overview of the number of samples required to detect all critical faults for at least one image. The table clearly demonstrates the advantage of sorting images by different metrics compared to a non-sorted approach.

Knowing this, to estimate the required number of samples for the FI, we may employ several strategies, detailed in Tables 4.5 and 4.6. The first rely on looking only at the metric values *prior* to the FI.

As a first approximation, we define a sample as challenging if it lies outside the so-called plateau region. The plateau is characterized by regions where the absolute value of the first derivative of a given metric function is close to zero or remains consistently below a defined threshold (e.g. $\epsilon = 5 \times 10^{-5}$). This threshold should be adapted based on the specific behavior of the metric under consideration, such as the red curve in our analysis (Fig. 4.8-4.9). Samples preceding the plateau are deemed of interest for FI purposes. Selecting only those samples for which the metric deviates from zero provides an upper bound on the number of inputs required for effective FI. Alternatively, one may use the hotspot regions (see Fig. 4.1-4.4) to filter inputs based on specific metric values. For instance, in the case of the EL2Nmetric, we may select samples within a tolerance window of $\epsilon = 0.3$ around the distribution's central mode. This value, while illustrative of the method's potential, should be tuned based on the specific dataset and metric distribution.

Using this selection strategy enables the identification of nearly all faults deemed critical for at least one input in the dataset, while significantly reducing the size of the test set. A summary of these results is presented in Table 4.6, where the random baseline is computed as the average over five randomly selected subsets of the same size as the EL2N-based selection.

A second strategy involves performing FI in an online fashion. In this case, a fault detection counter

is maintained to monitor the number of unique faults uncovered during the campaign.

Detecting the knee point of a curve is quite tricky, but the *Kneedle* approach [40] provides a valuable method for addressing this challenge. It is based on the notion of curvature, and on the observation that the points of maximum curvature in a discrete dataset are approximately the set of points in a curve that are local maxima (or minima) for the curve when the curve is rotated clockwise (or counterclockwise) through the line formed by the absolute minimum and maximum points. Nevertheless, to select the proper number of samples, the Kneedle algorithm from [40] has to be adapted to work online, iteratively refining the knee point as soon as new samples and detected faults pairs are available while performing the FI campaign. This stopping criterion ensures that the FI experiment continues only as long as new faults are being detected, optimizing computational resources while maintaining statistical relevance. This approach is guaranteed to stabilize due to the nature of the fault classification algorithm: once a fault is identified as critical, its status remains unchanged. As a result, the critical fault curve is, by definition, monotonic. For both the approaches, we analysed the number of simulations to be performed. An overview is given in the last columns of Tables 4.6, where we evaluated the time saving by using challenging samples only to estimate the number of critical faults/performing the FI campaign. To simplify, we considered a time of 1 millisecond per inference, independent of the network being used. The total time required to test all samples (100%) is given by: $\text{Total Time} = 1\,\text{ms} \times \#\text{TestImages} \times 16,640\,\text{faults}$. This lead to a total time of about 1.93 days for CIFAR10, CIFAR100 and MNIST (with 10,000 test images) and 0.76 days for Imagenette (with 3,925 test images). Furthermore, the time needed to perform the reliability assessments scales with the sample size. Specifically, for any given sample size, the time is calculated as $\text{Time} = \text{Total Time} \times \left(\frac{\%\ \text{of Samples}}{100}\right)$.

Dataset	Network	Cross Entropy	Gini	EL2N	Random (5 avg)
Imagenette	VGG11	53.4%	52.8%	53.8%	99.5%
CIFAR-100	HCGNet	20.92%	18.99%	19.14%	99.12%
CIFAR-10	MobileNetV2	9.30%	2.08%	9.23%	98.54%
	ResNet18	6.79%	0.90%	6.73%	99.30%
	ResNet9	7.96%	0.91%	7.94%	99.54%
	VGG16	8.48%	6.43%	8.46%	97.59%
	VGG11	10.58%	2.24%	10.56%	97.99%
	LeNet5	52.19%	50.01%	53.10%	99.36%
MNIST	VGG16	0.52%	2.17%	0.52%	90.50%
	VGG11	0.36%	1.54%	0.34%	99.14%
	ResNet18	0.40%	0.09%	0.40%	90.35%
	ResNet9	0.36%	0.12%	0.35%	97.65%
	LeNet5	97.39%	97.39%	97.60%	99.78%

Table 4.5: Percentage of dataset required to detect all critical faults, per network and metric. Metrics rank samples in descending order.

4.6 Hardware-aware Experiments

To evaluate whether our methodology remains effective on real hardware, we performed hardware-aware fault injection using flauers (cf. chapter 3). flauers provides a logic-level simulation of a systolic array, enabling realistic modeling of faults. During matrix multiplications, it considers faults that may affect the

registers within each processing element. Specifically, every time an inference is executed through a convolutional layer, the computation is redirected to flauers (cf. chapter 3), which simulates the convolution while accurately accounting for the impact of any injected faults.

Due to the high computational cost of simulation, fault injection was limited to transient faults occurring in the first layer of the neural network. Additionally, to further reduce simulation time, only a subset of input stimuli was used. In our experiments, we used 100 input images. This choice is linked to simulating a real environment in which to run *fault-detection tests*; when employing this technique on field to perform fault-detection, only a small subset of the test set can be used for two main reasons: (i) memory footprint, the dataset must be small enough to fit on the dedicated hardware and (ii) offline timing, the time spent doing fault-detection must be small enough not undermine the functionality of the system. As shown later, in table 4.6, often it is possible to detect most faults with a small portion of the test set. We find that 100 images can is a good compromise between test latency and fault detection. A fault affecting a PE causes an error on the register, which is then propagated to other PEs. Furthermore, for the experiments, we used a OS systolic array. Each of the registers on the PE has an equal probability of being affected by a fault. We injected 384 faults, a number determined according to the statistical guidelines proposed in [39] and in Equation 2.14. For the experiments, we used a subset of the networks illustrated above. More specifically, we used three networks: a ResNet20, a VGG11, and a MobileNetV2. Each fault injection was carried out 11 times: one time with the metric-based ranking and the other 10 times with different random subsets of images. This choice was made to properly demonstrate the effectiveness of the ranking system.

Figure 4.11 shows the results of the fault detection algorithm. The three Figures 4.11a, 4.11b, 4.11c show the cumulative number of faults detected for each sample of the dataset processed. Each graph shows three curves: (i) the dotted blue line is the number of faults detected by averaging the results of the 10 random rankings experiments; (ii) the orange dash-dotted line shows the *best* random: the run with the most faults detected; finally, (iii) the solid green line shows the faults detected using the metric-based ranking. It is visible that the fault discovery trends are very similar regardless of the network. The average of the random rankings shows that the maximum number of faults discovered at the end of the test is relatively small. In fact, looking at the solid line, we can see that the fault discovery with the metric-based ranked set can detect more than double the number of faults. Finally, in the "best" random sample among the 10 used, even though the performance is better than the average, the number of faults discovered at the end of the inferences is always smaller than that discovered with the metric-based ranking. It is worth noticing the presence of images with high metric values in the best random ranking. For instance, the best random ranking for ResNet20 had 4 images belonging to the metric-based one, the highest among all the random rankings. Similarly, the best random ranking for VGG11 had 2 images belonging to the best ranking. The ranking-based approach methodology allows a much faster fault detection: using the metric-based ranking, not only permits reaching a plateau in the fault detected, but this is also reached using just a handful of samples.

To give concrete figure, let us compare the experiments done with the ResNet20 (Figure 4.11a). On average (dotted line), only about 90 faults are detected using 100 samples. The best random ranking (dash-dotted line) shows a plateau with 150 faults detected after 40 samples. Although this ranking performed much better than average, the metric-based ranking (solid line) shows 180 faults discovered using less than 40 samples in total, implying a faster convergence and more efficient fault discovery.

Network	Method	Part of Samples	Faults Detected	Time (days)	Part of Critical Faults Detected
		CIFAR100			
HCGNet	Central Mode	11.94%	2606	0.23	94.56%

Network	Method	Part of Samples	Faults Detected	Time (days)	Part of Critical Faults Detected
	Random Detection	11.94%	1452	0.23	52.69%
	Pre-Plateau Detection	29.11%	2756	0.56	100.00%
	Full Dataset	100%	2756	1.93	100.00%
		CIFAR10			
LeNet5	Central Mode	21.18%	1132	0.41	97.75%
	Random Detection	21.18%	774	0.41	66.79%
	Pre-Plateau Detection	79.45%	1158	1.53	100.00%
	Full Dataset	100%	1158	1.93	100.00%
ResNet18	Central Mode	1.81%	822	0.04	99.52%
	Random Detection	1.81%	503	0.04	60.90%
	Pre-Plateau Detection	14.47%	826	0.28	100.00%
	Full Dataset	100%	826	1.93	100.00%
VGG11	Central Mode	1.40%	793	0.03	96.83%
	Random Detection	1.40%	518	0.03	63.25%
	Pre-Plateau Detection	16.41%	819	0.32	100.00%
	Full Dataset	100%	819	1.93	100.00%
VGG16	Central Mode	0.98%	891	0.02	99.66%
	Random Detection	0.98%	512	0.02	57.27%
	Pre-Plateau Detection	13.33%	894	0.26	100.00%
	Full Dataset	100%	894	1.93	100.00%
MobileNetV2	Central Mode	3.37%	1306	0.07	97.46%
	Random Detection	3.37%	640	0.07	47.76%
	Pre-Plateau Detection	23.28%	1340	0.45	100.00%
	Full Dataset	100%	1340	1.93	100.00%
ResNet9	Central Mode	1.77%	1327	0.03	88.02%
	Random Detection	1.77%	621	0.03	41.19%
	Pre-Plateau Detection	16.27%	1508	0.31	100.00%
	Full Dataset	100%	1508	1.93	100.00%
		MNIST			
ResNet18	Central Mode	0.26%	581	0.01	100.00%
	Random Detection	0.26%	483	0.01	83.13%
	Pre-Plateau Detection	3.64%	581	0.07	100.00%
	Full Dataset	100%	581	1.93	100.00%
VGG11	Central Mode	0.15%	645	0.00	97.58%
	Random Detection	0.15%	511	0.00	77.31%
	Pre-Plateau Detection	3.61%	661	0.07	100.00%
	Full Dataset	100%	661	1.93	100.00%
VGG16	Central Mode	0.26%	551	0.01	99.28%
	Random Detection	0.26%	515	0.01	92.79%
	Pre-Plateau Detection	3.28%	555	0.06	100.00%
	Full Dataset	100%	555	1.93	100.00%

Network	Method	Part of Samples	Faults Detected	Time (days)	Part of Critical Faults Detected
ResNet9	Central Mode	0.33%	527	0.01	100.00%
	Random Detection	0.33%	518	0.01	98.29%
	Pre-Plateau Detection	3.11%	527	0.06	100.00%
	Full Dataset	100%	527	1.93	100.00%
LeNet5	Central Mode	0.79%	342	0.02	70.66%
	Random Detection	0.79%	291	0.02	60.12%
	Pre-Plateau Detection	9.01%	436	0.17	90.08%
	Full Dataset	100%	484	1.93	100.00%
Imagenette					
VGG11	Central Mode	0.23%	97	0.00	84.35%
	Random Detection	0.23%	87	0.00	75.65%
	Pre-Plateau Detection	4.38%	114	0.03	99.13%
	Full Dataset	100%	115	0.76	100.00%

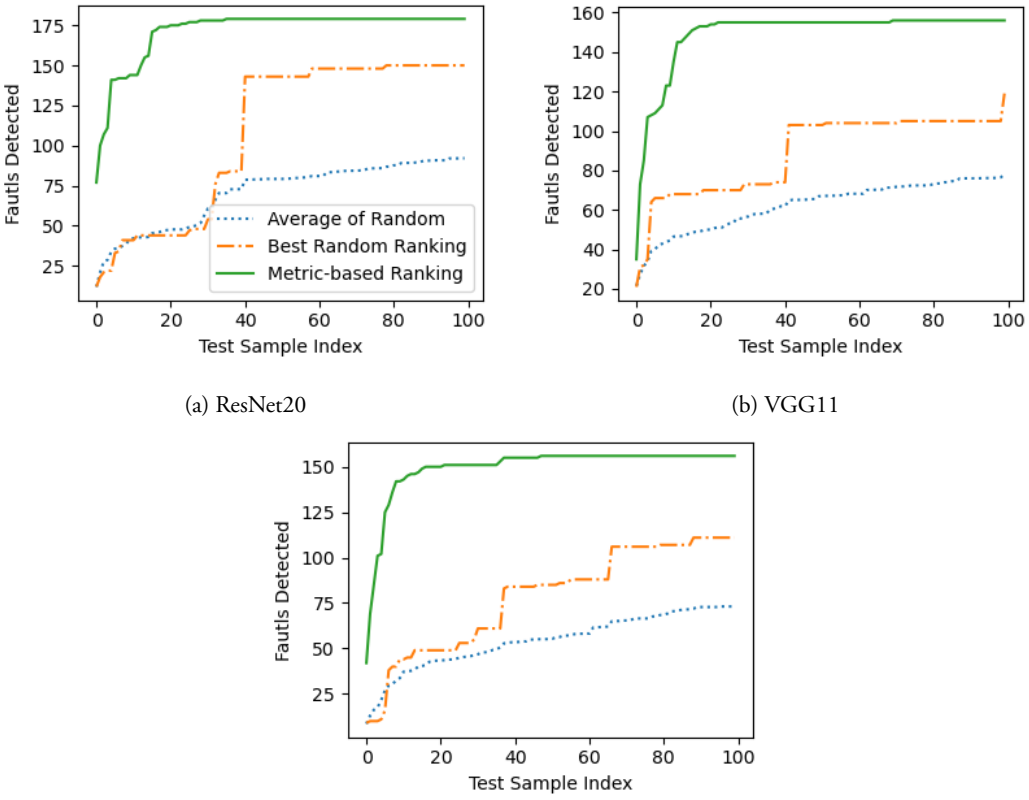
Table 4.6: Random vs EL2N-based Detection.

4.7 Conclusions

This study presents an empirical approach to evaluating the reliability of DNNs by prioritizing *challenging* samples identified through confidence metrics such as EL2Nduring FI campaigns. We demonstrate that this prioritization significantly reduces the number of samples required to achieve comprehensive fault discovery, thereby lowering computational costs. Our experiments, conducted in both hardware-agnostic and hardware-aware settings, confirm the effectiveness of the method across diverse architectures and datasets. For example, fewer than 100 samples sufficed to fully assess the criticality of ResNet-18, while fewer than 2'000 samples were adequate for more complex models like HCGNet trained on CIFAR-100. The hardware-aware evaluations further validate the approach's applicability to realistic implementations, highlighting its robustness and versatility. This is particularly important because fault injection simulations in hardware-aware scenarios are computationally intensive and time-consuming; thus, reducing the sample set directly translates into substantial savings in simulation time and resources.

Moreover, the introduced *fault detection curves* offer valuable visual insights into fault discovery dynamics, illustrating the rapid convergence of critical fault detection when employing metrics-based sample prioritization. This not only accelerates FI campaigns but also enhances understanding of the interplay between test sample characteristics and fault activation.

The proposed methodology constitutes a meaningful advancement in DNN reliability assessment by enabling efficient and targeted FI campaigns. Future work will focus on extending the framework to additional fault models and more complex DNNs, thereby broadening its scope and impact. Additionally, we intend to explore applications beyond classification, including object detection, image segmentation, and natural language processing tasks. This will require identifying suitable confidence metrics tailored to each task, with the potential to generalize the approach across a wide range of deep learning domains.



(a) ResNet20

(b) VGG11

(c) MobileNetV2

Figure 4.11: Hardware-aware fault detection experiments results. The faults are injected in the first layer of the neural networks.

References

[1] I. Goodfellow, Y. Bengio, and C. Aaron, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[2] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A flexible fault injection framework for TensorFlow applications," in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, ISSN: 2332-6549, Oct. 2020, pp. 426–435. DOI: 10.1109/ISSRE5003.2020.00047

[3] A. Mahmoud et al., "PyTorchFI: A runtime perturbation tool for DNNs," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, ISSN: 2325-6664, Jun. 2020, pp. 25–31. DOI: 10.1109/DSN-W50199.2020.00014

[4] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network," in *2019 IEEE Latin American Test Symposium (LATS)*, ISSN: 2373-0862, Mar. 2019, pp. 1–6. DOI: 10.1109/LATW.2019.8704548 Accessed: Sep. 4, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8704548/

[5] A. P. Arechiga and A. J. Michaels, "The robustness of modern deep learning architectures against single event upset errors," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, ISSN: 2377-6943, Sep. 2018, pp. 1–6. DOI: 10.1109/HPEC.2018.8547532 Accessed: Sep. 4, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8547532/

[6] S. BurelT, A. EvansT, and L. Anghel, "Improving DNN fault tolerance in semantic segmentation applications," in *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, ISSN: 2765-933X, Oct. 2022, pp. 1–6. DOI: 10.1109/DFT56152.2022.9962354 Accessed: Sep. 4, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9962354/

[7] A. Ruospo, E. Sanchez, L. M. Luza, L. Dilillo, M. Traiola, and A. Bosio, "A survey on deep learning resilience assessment methodologies," *Computer*, vol. 56, no. 2, pp. 57–66, Feb. 2023, ISSN: 0018-9162, 1558-0814. DOI: 10.1109/MC.2022.3217841 Accessed: Mar. 27, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10042090/

[8] Q. Hu et al., *Evaluating the robustness of test selection methods for deep neural networks*, Jul. 29, 2023. DOI: 10.48550/arXiv.2308.01314 arXiv: 2308.01314[cs]. Accessed: Jan. 10, 2025. [Online]. Available: http://arxiv.org/abs/2308.01314

[9] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer, "Input prioritization for testing neural networks," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, Apr. 2019, pp. 63–70. DOI: 10.1109/AITest.2019.000-6 Accessed: Nov. 15, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/8718224/?arnumber=8718224

[10] W. Ma, M. Papadakis, A. Tsakmalis, M. Cordy, and Y. L. Traon, "Test selection for deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, 13:1–13:22, Jan. 3, 2021, ISSN: 1049-331X. DOI: 10.1145/3417330 Accessed: Jan. 21, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3417330

[11] C. Zhao, Y. Mu, X. Chen, J. Zhao, X. Ju, and G. Wang, "Can test input selection methods for deep neural network guarantee test diversity? a large-scale empirical study," *Information and Software Technology*, vol. 150, p. 106 982, Oct. 1, 2022, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2022.106982 Accessed: Jan. 10, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001185

[12] T. Borovicka et al., "Selecting representative data sets," in *Advances in Data Mining Knowledge Discovery and Applications*, IntechOpen, Sep. 12, 2012, ISBN: 978-953-51-0748-4. DOI: 10.5772/50787 Accessed: Oct. 5, 2025. [Online]. Available: https://www.intechopen.com/chapters/39037

[13] W. Li, Y. Wang, K. Zou, H. Li, and X. Li, "Adversarial Testing: A Novel On-Line Testing Method for Deep Learning Processors," in *2023 IEEE 32nd Asian Test Symposium (ATS)*, 2023, pp. 1–6. DOI: 10.1109/ATS59501.2023.10317994

[14] S. Kundu, S. Banerjee, A. Raha, and K. Basu, "Special Session: Effective In-field Testing of Deep Neural Network Hardware Accelerators," in *2022 IEEE 40th VLSI Test Symposium (VTS)*, 2022, pp. 1–4. DOI: 10.1109/VTS52500.2021.9794227

[15] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021. DOI: 10.1109/TVLSI.2020.3048829

[16] F. Meng, F. S. Hosseini, and C. Yang, "A self-test framework for detecting fault-induced accuracy drop in neural network accelerators," in *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, ISSN: 2153-697X, Jan. 2021, pp. 722–727. Accessed: Oct. 5, 2025. [Online]. Available: https://undefined/document/9371530

[17] F. Meng and C. Yang, "Exploring image selection for self-testing in neural network accelerators," in *2022 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, ISSN: 2159-3477, Jul. 2022, pp. 345–350. DOI: 10.1109/ISVLSI54635.2022.00076 Accessed: Oct. 5, 2025. [Online]. Available: https://undefined/document/9912080

[18] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, Apr. 10, 2015. DOI: 10.48550/arXiv.1409.1556 arXiv: 1409.1556[cs]. Accessed: Oct. 5, 2025. [Online]. Available: http://arxiv.org/abs/1409.1556

[19] M. Barbareschi and S. Barone, "Investigating the resilience source of classification systems for approximate computing techniques," *IEEE Transactions on Emerging Topics in Computing*, vol. 13, no. 2, pp. 382–394, Apr. 1, 2025, Publisher: IEEE Computer Society, ISSN: 2168-6750. DOI: 10.1109/TETC.2024.3403757 Accessed: Oct. 5, 2025. [Online]. Available: https://www.computer.org/csdl/journal/ec/2025/02/10542568/1XmMGLjxlRe

[20] M. Barbareschi and S. Barone, "Investigating the Resilience Source of Classification Systems for Approximate Computing Techniques," *IEEE Transactions on Emerging Topics in Computing*, p. 12, 2024, ISSN: 2168-6750. DOI: 10.1109/TETC.2024.3403757

[21] R. Konig, R. Renner, and C. Schaffner, "The Operational Meaning of Min- and Max-Entropy," *IEEE Transactions on Information Theory*, vol. 55, no. 9, pp. 4337–4347, 2009, ISSN: 1557-9654. DOI: 10.1109/TIT.2009.2025545

[22] M. Paul, S. Ganguli, and G. K. Dziugaite, "Deep Learning on a Data Diet: Finding Important Examples Early in Training," in *Advances in Neural Information Processing Systems*, vol. 34, Curran Associates, Inc., 2021, pp. 20 596–20 607.

[23] A. Mao, M. Mohri, and Y. Zhong, "Cross-Entropy Loss Functions: Theoretical Analysis and Applications," en, in *Proceedings of the 40th International Conference on Machine Learning*, PMLR, 2023, pp. 23 803–23 828.

[24] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," en, in *Proceedings of the 50th Annual Design Automation Conference on - DAC '13*, Austin, Texas: ACM Press, 2013, p. 1, ISBN: 978-1-4503-2071-9. DOI: 10.1145/2463209.2488873

[25] I. J. Goodfellow, J. Shlens, and C. Szegedy, *Explaining and Harnessing Adversarial Examples*, en, 2015. arXiv: 1412.6572 [cs, stat].

[26] A. Ruospo et al., "Assessing Convolutional Neural Networks Reliability through Statistical Fault Injections," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10136998

[27] M. H. Ahmadilivani et al., "Special Session: Approximation and Fault Resiliency of DNN Accelerators," in *2023 IEEE 41st VLSI Test Symposium (VTS)*, IEEE, 2023, pp. 1–10.

[28] S. Pappalardo, A. Ruospo, I. O'Connor, B. Deveautour, E. Sanchez, and A. Bosio, "A Fault Injection Framework for AI Hardware Accelerators," in *2023 IEEE 24th Latin American Test Symposium (LATS)*, Veracruz, Mexico, Mexico, 2023. DOI: 10.1109/LATS58125.2023.10154505

[29] A. Ruospo et al., "Image Test Libraries for the on-line self-test of functional units in GPUs running CNNs," in *2023 IEEE European Test Symposium (ETS)*, 2023, pp. 1–6. DOI: 10.1109/ETS56758.2023.10174176

[30] I. Miro-Panades et al., "A 772uj/frame imagenet feature extractor accelerator on hd images at 30fps," in *2024 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, 2024, pp. 25–29. DOI: 10.1109/APCCAS62602.2024.10808841

[31] Y. LeCun et al., "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[32] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, Dec. 10, 2015. DOI: 10.48550/arXiv.1512.03385 arXiv: 1512.03385[cs]. Accessed: Oct. 5, 2025. [Online]. Available: http://arxiv.org/abs/1512.03385

[33] Y. LeCun, C. Cortes, and C. Burges, *MNIST Handwritten digit database*, http://yann.lecun.com/exdb/mnist/, 1998.

[34] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," en, Tech. Rep., 2009, p. 60.

[35] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, *Mobilenetv2: Inverted residuals and linear bottlenecks*, 2019. arXiv: 1801.04381 [cs.CV]. [Online]. Available: https://arxiv.org/abs/1801.04381

[36] C. Yang et al., *Gated Convolutional Networks with Hybrid Connectivity for Image Classification*, 2019. DOI: 10.48550/arXiv.1908.09699 arXiv: 1908.09699 [cs].

[37] J. Howard, *Imagenette: A smaller subset of 10 easily classified classes from the imagenet dataset*, https://github.com/fastai/imagenette, Accessed: 2025-06-09, 2019.

[38] C. Spearman, "The Proof and Measurement of Association between Two Things," *The American Journal of Psychology*, vol. 15, no. 1, pp. 72–101, 1904, ISSN: 00029556. DOI: 10.2307/1412159 JSTOR: 1412159.

[39] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506. DOI: 10.1109/DATE.2009.5090716

[40] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a "Kneedle" in a Haystack: Detecting Knee Points in System Behavior," in *2011 31st International Conference on Distributed Computing Systems Workshops*, 2011, ISBN: 978-1-4577-0384-3. DOI: 10.1109/ICDCSW.2011.20

CHAPTER 5

Approximation

Modern Deep Neural Networks (DNNs) have revolutionized the field of artificial intelligence and machine learning, due to their capacity to learn from vast amounts of data, making very complex predictions. They are very powerful, able to learn from a vast amount of data and make complex predictions. In specific tasks, such as image classification, they have demonstrated to be able to surpass human performance [1]. Nowadays, the possibility of employing them in many safety-critical is appetizing. Nevertheless, the presence of faults in hardware can affect significantly degrade the performances. To overcome this problem, traditional techniques resort to redundancy. In this chapter, we will explore an approximation-based technique to achieve fault tolerance, essentially exploiting the intrinsic redundancy of the models, employing approximate multipliers. Other than achieving fault tolerance, this technique also improves power usage.

5.1 Motivation

One of the challenges associated with DNNs is their high energy consumption, due to enormous computational required to complete an inference. Convolutional Neural Network (CNN) models are composed by many convolutional layers, and convolution is an expensive computation [2]. As said previously, to address this issue, recently, the new the architecture of Systolic Array (SA) gain popularity. To further push the energy and performance efficiency, Approximate Computing (AxC) seems to be a very promising approach [3]. The idea behind AxC allows some degree of error during computations. Some applications do not really need to be "precise", as it is possible to cope with perceptual imperfections without impacting on the usability of the application. To this extent, low-power hardware can be used. AxC aims to reduce the precision of the hardware in order to save energy consumption. A trivial example, to properly explain AxC consists in using smaller data types: instead of computing a 32bit floating-point operation, one could be interested in performing it with 8bit integer values. Nevertheless, there are more sophisticated techniques than this. Interestingly, reduced precision leads to applications that produce less (but still sufficiently) accurate results while reducing power consumption. Such applications are characterized as intrinsically resilient to noise and errors that affect computation. Inherent resilience is closely related to the application domain and clearly DNNs are perfect candidates thanks to their inherent resilience to noise [4]. The employment of approximate multipliers, the main focus of this chapter, in DNNs is not a new. Many studies focused their attention on the topic. To cite some:

design of AxC multipliers for Neural Network (NN) [5] in this study, the authors design three mul-

tipliers and use them in two DNNs without retraining; they found that using their multipliers the DNNs would lose at most 15pt% and at best, only 1pt% when using MNIST [6] and a simple MultiLayer Perceptrons (MLP). When scaling up in complexity both the model and dataset, they found that and 24pt% in the worst case and only 1pt% in the best case, using a LeNet-5 (cf. subsection 2.1.4) and SVHN[18]. As a recurrent theme in this chapter, we will see how both complexity of the dataset and the neural architecture play an essential role in the resilience of NNs

[18] Street View House Numbers is a dataset developed by google, showcasing pictures of house numbers from their service *street view*

important error features of AxC multipliers [7] in this paper, the authors perform some inferences using approximate multipliers to understand which error features are the most significant for their DNNs, which consist of 2 ResNets (18 and 34) and a VGG-16. They find that, among all the possible metrics, the most relevant are the error bias (whether the multiplier is more prone to overestimate or underestimate the value), the mean error distance and the error rate (the probability of giving a wrong error).

improving the accuracy of NN through AxC multipliers [8] here the authors try Cartesian Genetic Programming to generate approximate multipliers and the use them to improve the accuracy of their NNs. They find out that such multipliers can be beneficial to the NN, as they saw (i) increase in the accuracy for some of the multipliers and (ii) the noise introduced by the multipliers works as an implicit dropout, helping to reduce overfitting.

Nevertheless, there is no work in the literature that investigates the use of AxC multipliers to improve resilience.

Given the marvelous results obtained with DNNs, it is interesting to use them in safety-critical systems. For example, [9] used a neural network to determine the flight regime of an aircraft using temporal segmentation, achieving more than 90% accuracy. Another example can be found in [10], in which the authors proposed a novel approach for lane detection in the context of autonomous driving.

Although DNNs come with such inherent resilience to noise, recent studies in the literature have shown that the application is susceptible to faults in the hardware. Thus, inference can be significantly affected, leading to DNN prediction failures that are likely to lead to a detrimental effect on the application [11], [12], [13], [14], [15], [16]. Therefore, ensuring the resilience of hardware accelerators is crucial, especially when they are deployed on real-time safety-critical systems [17].

The classical technique used to improve the resilience is to introduce fault tolerance mechanisms by adding redundancy in the design [18]. The problem of using redundancy, even in the case of selective fault tolerance [19], is the cost in terms of area, power and latency that can be prohibitive, especially for edge applications. Given this context, there is a need to reduce the cost of fault tolerance for DNN hardware accelerators.

In order to improve the system resilience, many fault tolerance mechanisms can be adopted. A typical solution consists on adding redundancy to the original design [20]. However, the problem of using redundancy, even in the case of selective fault tolerance techniques, is the increasing costs in terms of area, power and latency that can be prohibitive, especially for edge applications. Therefore, there is a need to further reduce the fault tolerance costs for DNN hardware accelerators.

This chapter proposes a novel approach entirely based on the use of AxC to increase the resilience of DNNs without using redundancy and thus avoiding extra costs. In the first section, we show a case study to understand the impact of AxC multipliers on the resilience of a systolic array architecture used to accelerate the DNN execution. We conducted a Fault Injection (FI) campaign to study the impact of single bit-flips on the resilience of the network. Preliminary results have shown that using an AxC multiplier, it is possible to improve the resilience of almost 10% with better efficiency than the accurate implementation. In the second section we repeat the experiments with more networks to find out that training has an essential role in the deployment of this technique.

5.2 Approximate Computing

AxC is the technique of performing computing in an *approximate* way. The significance of the word *approximate* changes based on the context.

A survey from 2016 [21] explains that approximation is done in three steps. Firstly, it is important to find the approximable code and/or data. Secondly, it is needed to choose some approximation techniques. The landscape is varied, going from precision scaling to loop perforation to voltage scaling and refresh rate reductions. Finally, after the implementation is performed, it is necessary to evaluate its correctness and performance. In the context of NNs, we can approximate many things due to the very nature of the processing. In the context of this manuscript, we chose to approximate the multiplications that happen during the processing. This choice is motivated by two main factors: (i) because multiplication is by far the most frequent operation in NN and (ii) it is reasonable to assume that NN is similar to human perception.

Several works have been published so far concerning the use of AxC in the context of fault tolerance. This research area is generally known as Approximate Triple Modular Redundancy.

This approach implements Triple Modular Redundancy (TMR) [22] which employs three replicas of the same circuit at the output is then connected to a majority voter. This structure allows for increased resilience, as any fault is prevented to have any effect, since the two other replicas ensure a correct result. Approximate TMR uses three approximate replicas, instead of three fully-precise replicas. For a given input, only one circuit can provide an incorrect (or approximate) answer. However, the fault tolerance capability mainly depends on the voter that has to be modified [20].

A similar approach referred to as Quadruple Approximate Modular Redundancy was presented [23]. It is based on the idea of selectively approximating a subset of the circuit. The final goal is to have four approximate replicas in a manner that ensures that, out of the four, at least three will provide precise results for a given output. The benefit is that the voter does not have to be modified. However, the design cost is higher since the approximation must be carefully identified.

The described works target fault tolerance through masking. Concerning fault detection only, in [24], the authors exploit approximate computing for image processing applications through duplication with comparison.

The above techniques summarize the use of AxC to reduce the cost of testing and fault detection/tolerance. Another interesting branch of research is investigating how AxC impacts a system's intrinsic reliability, and whether an approximate application is more or less resilient to hardware faults than the precise application. Indeed, AxC exploits the inherent resilience of an application to noise and computing errors. Therefore, an application executed on approximated hardware is less resilient to hardware faults.

In [25], the authors studied the impact of AxC on the reliability of DNN application. As AxC technique, they used the data type and bit-width reduction. In particular, they compared 32-bit floating-point versus 16-bit and 8-bit integer data for storing synaptic weights. The DNN was an image classifier based on LeNet-5 topology. The reliability assessment has been done through radiation experiments: the system running the DNN was exposed to neutron beam. In particular, the target of the beam was the DRAM memory storing the DNN weights. Interestingly, the 32 bit floating point (i.e. the precise) is the less reliable implementation while the approximate DNN (i.e. the one using int16) shows the higher resilience. The reason is on the fact that half of the memory is used thanks to AxC therefore the probability to have a fault is drastically reduced. This example proves that AxC can have a positive side effect in increasing the reliability of an application.

In [14], [15], the authors proposed a specific fault-tolerant technique for systolic arrays used to deploy NNs. In [14], permanent faults are targeted, and the idea consists in skipping the computation of a faulty PE. Contrarily, [15] targets transient faults and the approach consists in computing the same operation in two Processing Elements (PEs) and comparing the results. If they differ, the result is not fed to the

next PE. Both techniques correspond to prune the DNN.

This work exploits two different AxC techniques in systolic arrays: bit-width reduction (concerning weights) and approximate multipliers. It will be shown that it is possible to increase resilience of DNNs without leveraging redundancy.

5.2.1 Metrics and Multipliers

The selected metrics are relative Mean Absolute Error (MAE) and the relative Worst Case Error (WCE). They are defined in the following equations:

$$MAE = \frac{1}{N^2}\sum_{i=1}^{N}\sum_{j=1}^{N}(\|\hat{y}_i - y_j)\|) \tag{5.1}$$

$$WCE = \max\sum_{i=1}^{N}\sum_{j=1}^{N}(\|\hat{y}_i - y_j\|) \tag{5.2}$$

where N is the number of possible combination given $n-$bits (in our case $n = 8$, which implies $N = 256$), y is the output of the precise multiplier and $\hat{y}$ is the output of the approximate multiplier.

5.2.2 Functional approximation

Functional approximation [3] aims at modifying the circuit structure in order to obtain a simpler (thus cheaper) implementation with a reduced precision compared to the precise implementation. This technique has been widely adopted with arithmetic circuits (i.e. adders and multipliers), obtaining benefits in terms of area, power, and latency reduction.

5.3 A Case Study using a Systolic Array

Given this background, a natural question that arose was the following: what is the effect of approximation on NNs reliability when using a Systolic Array? To answer this question, we performed FI on a LeNet-5 using 6 different types of multipliers. The following subsections show the experimental setup and the results.

5.3.1 Neural network

The LeNet-5 CNN architecture has been extensively discussed in the background, in subsection 2.1.4, trained and tested on MNIST dataset, has been exploited. It is composed of three convolutional layers and four fully connected. The training process was performed by using [26] without any approximation: learning rate started at 0.05, with the decay of 5×10^{-4} every $375(\times 128)$ iterations, the momentum was set to 0.9. The final accuracy of the model was equal to 99.05%. We then performed 8-bit quantization through following steps [26].

1. All weights are re-scaled in the range $[-1.0, 1.0]$ and activations at each layer are re-scaled in the range $[-1.0, 1.0]$ for signed outputs and $[0.0, 1.0]$ for unsigned outputs;

2. inputs, weights, biases and activations are quantized to the desired n_{bits} by converting $[-1.0, 1.0]$ and $[0.0, 1.0]$ to $[-2^{n_{bits}-1} - 1, 2^{n_{bits}-1} - 1]$ and $[0, 2^{n_{bits}-1} - 1]$.

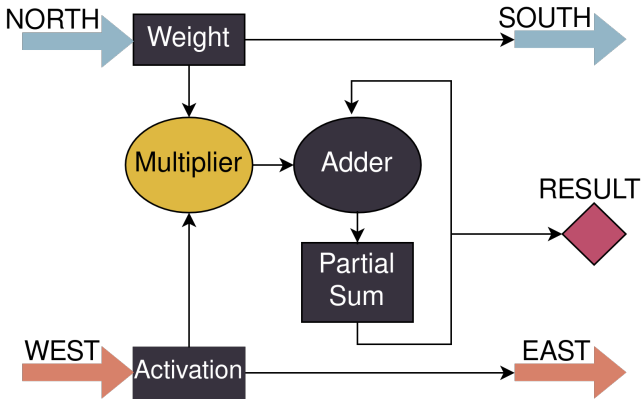


Figure 5.1: Functional diagram of a processing element.

5.3.2 Approximate multipliers

To quantify the effect of inaccuracy on reliability of our systolic array, we replaced the accurate multipliers in each PE with the approximate multipliers presented in EvoApproxLiteLITE [27]. These multipliers are generated as combinations of smaller multipliers, which handle a sub-problem, effectively tiling the multiplication. The smaller multipliers belong to a library of general-purpose approximate multipliers designed for combinational circuits based on a multi-objective Cartesian genetic programming [28]. It represents candidate circuits as directed acyclic graphs and tries to minimize delay, error, and power consumption simultaneously to find a set of approximate circuits along a Pareto front.

It is possible to find the used multiplier at the link https://ehw.fit.vutbr.cz/evoapproxlib/?folder=multiplers/12x12_signed

The LeNet-5 we used performs a multiplication between a signed weight and an unsigned activation, thus we needed a mixed-sign multiplier, which is not available in the aforementioned library. As a consequence, we set up the experiments using a 12-bit signed multiplier and simply extending the sign on the operands. Please note that this method has the effect of making the approximation even worse: since the multipliers are designed to operate on 12-bits values, using only the least significant 8 bits means reducing the performance of the multipliers. To quantify this effect, we computed the Mean Absolute Error (MAE) and the Worst Case Error (WCE) with the mixed-sign range the networks actually uses. Table 5.1 shows the *original* [27] metrics of the multiplier (columns MAE, WCE) and the newly computed metrics that only use our 8bit values ranges (columns MAE-8, WCE-8). Note that the network was trained completely using accurate logic. This means that the weights are not tuned *just right* for the errors that are introduced using an approximate multiplier.

Table 5.1: Characterization of the approximate multipliers. Every value is a percentage.

Name	MAE	WCE	MAE-8	WCE-8
mul12s_2PT	0.000073	0.00029	0.019	0.0748
mul12s_2QH	0.0031	0.013	0.134	0.514
mul12s_2R5	0.0092	0.037	0.315	1.234
mul12s_34P	0.032	0.17	0.785	3.920
mul12s_2TE	0.19	0.77	6.080	24.417

Note: in order to identify the multipliers, we will use the full name or three-character ID (omitting the strings “mul12s” or “mul8s”) interchangeably. This means, for example, that “mul12s_2PP” and “2PP” identify the same multiplier.

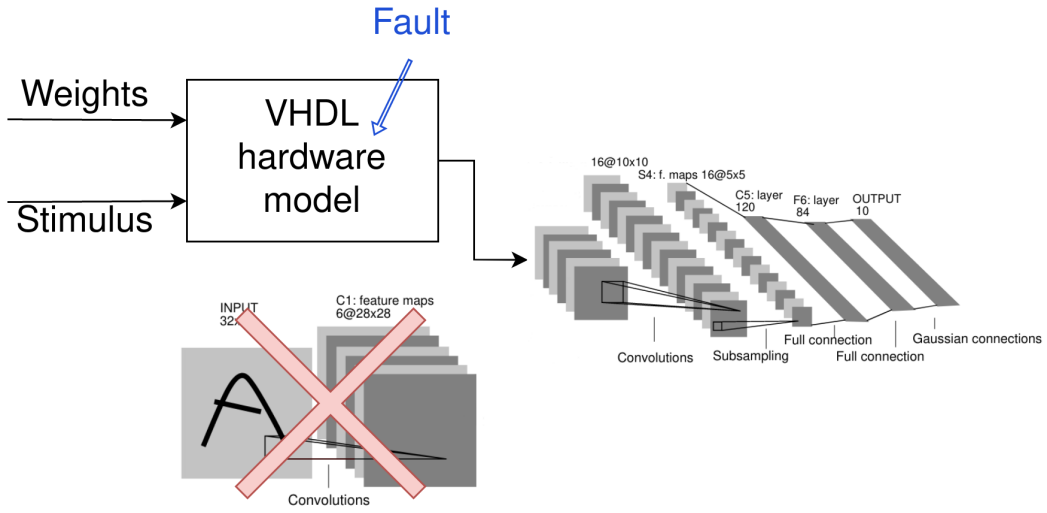


Figure 5.2: Fault injection process. The processing of the first layer is done using a VHDL model of a SA. Being the only faulty processing, we can then just "forward" the faulty data to a software implementation of the network, assuming fault-free conditions for the rest of the network.

5.3.3 Experimental setup

The implementation of LeNet-5 was derived using N2D2 [26] and exported in `C`, the programming language. The network was quantized in order to use int8 values. Specifically, the weights are represented as signed 8bit integers values, while the activations where represented as unsigned 8bit integers. The injection was performed by passing the data to hardware simulations, then gathered back and fed to the other layers of the NN. Figure 5.2 shows this setup.

The hardware model describes a systolic coprocessor composed by 6 systolic arrays, each of size 28×28. The size is chosen based on the neural application. In the case of the LeNet-5 previously described, the output of the first layer is composed 6 channels, each with 28x28 pixels. The mapping used for the systolic array is able to process one convolution at a time, thus for computing the 6 channels (i.e. 6 convolutions) 6 systolic arrays are necessary, are previously described. Alternatively, it is possible to see this model as a single SA executing the convolution sequentially.

According to [29], the most critical layer when neurons inputs are faulty is the first one; so, in this work, the experimental efforts were carried out by injecting this layer, while the others are executed in software using the software implementation of the network. More in details, we injected only the weight register of the PEs (cf. Figure 5.1). The injection was aimed at representing a transient fault affecting the registers.

The space of possible faults is then 3-dimensional, and its size depends on both the array size and the bit-width: $\text{FaultSpace} = 2 \times K \times M$. We have a total of M PEs. Each PE has K bits for the input of the weights. Each fault corresponds to an alteration of one bit, forcing its value to either 1 or 0. We call the forced value **polarity**, and we have only two possible values.

Given the size of the fault space, it is possible to compute n (cf. 2.2.4). In this case, we assumed a 1% margin of error, 50% probability of a fault resulting in a failure, a cut-off point of 2.58 which corresponds to 99% confidence level. This setup gives us $n = 15'000$, thus we injected n single-bit

faults using 100 random images (10 images for each class). In order to have a fair evaluation and exclude the bias of the dataset, the 100 images are different for each fault.

The evaluation of the fault is done using the fault classification (cf. subsection 2.2.3). Basically, we check whether the top-1 probability of the injected network $max(\tilde{Y})$ is greater or smaller than the golden top-1 probability $max(\hat{Y})$ and classify the fault accordingly. Note that **masked** faults do not produce any difference in the output, while only **critical** faults involve misclassification. Finally, we classify as **benign** the faults that fall in the first two categories (masked and good), since not only there is no misclassification, but the confidence of the prediction is equal or higher than the fault-free run. The other three categories are defined as malignant.

In order to fairly evaluate the performances of the FI campaigns, the accuracy of the NN was evaluated without injecting any fault, substituting the accurate multipliers with the approximate ones. We evaluated the classification accuracy of the network on the entire validation set. To speed up the FI process, i.e. without running architectural-level simulations we used the `C` code of each multiplier, available in [27].

5.3.4 Intrinsic NN robustness

We established the baseline of the experiments by computing the accuracy of the network using the different multipliers, without injecting any fault. The approximate multiplier were used to replace the accurate logic, but this was done only for the convolution. Consequently, every convolution of the network was computed with approximate logic, while the rest of the computations were accurate.

Multiplier	Accuracy (%)
accurate	99.05
mul12s_2PT	99.08
mul12s_2QH	**99.10**
mul12s_2R5	99.06
mul12s_34P	98.24
mul12s_2TE	9.80

Table 5.2: Fault-free classification accuracy after the replacement of the multiplier in the convolution. Bold text shows the best performance.

Table 5.2 shows baseline accuracy of the NN. The accuracy is computed using the validation set of 10'000 input images.

Interestingly, the NN withstands the approximation gracefully. Noteworthy, the accuracy slightly increases when using approximated multipliers whose error is quite small. This result shows that the introduced error, due to approximation, might even be beneficial for the NN, although the reason remains unclear. Furthermore, this result is similar to what is shown in [8], in which the authors show that the accuracy increases when using an approximate multiplier and then performs some retraining steps. Similarly, we find an improvement (although minimal) without retraining the network.

While the best case improvement is only 0.05%, it is a non-negligible result. This is because it corresponds to 500 more input images correctly classified, when compared with the accurate counterpart. Nevertheless, introducing major approximations produces nefarious effects. Expectedly, the worst multiplier introduced an error so big that the response is random, with an accuracy of about 10% (random over 10 classes). The confusion matrix shows that the NN classified every input as a 0.

5.3.5 General resilience

Figure 5.3a shows the general behavior of the different configurations. Furthermore, Table 5.3 reports the percentage of masked, benign and critical faults per each FI campaign. It is possible to notice that the 2PT produces the same effect as the accurate multiplier. Increasing the error, multipliers 2QH and 2R5 have the effect of decreasing the resilience, since the number of masked and benign faults decrease significantly. On the other hand, 34P shows a completely different behavior: the majority of injected faults are masked. Compared to the accurate network, there is a 9.5% increase in masked faults, even though the classification accuracy dropped by about 1%*pt*. This outcome shows the benefit obtained by using this multiplier in this NN. Finally, multiplier 2TE is not reliable in any measure, misclassifying the majority of the inputs; actually, about 50% of the inputs were correctly classified when compared with the ground truth, but its performance is unacceptable anyway.

#	Multiplier	Masked(%)	Benign(%)	Critical(%)
1	accurate	64.65	84.18	**0.15**
2	mul12s_2PT	63.90	84.20	**0.15**
3	mul12s_2QH	38.92	79.79	0.20
4	mul12s_2R5	26.96	76.31	0.30
5	mul12s_34P	**74.16**	**88.08**	0.37
6	mul12s_2TE	3.94	14.44	48.59

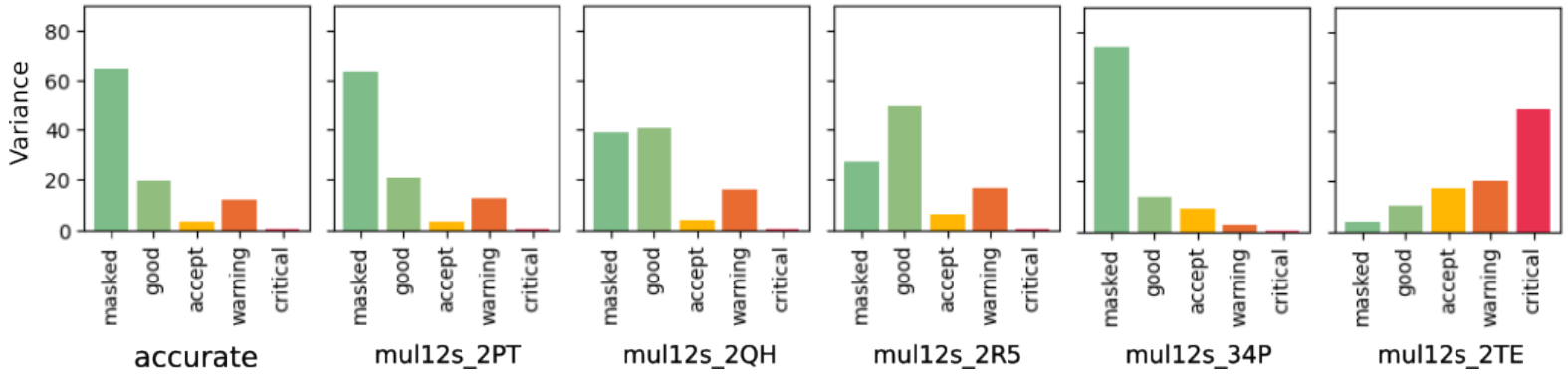
Table 5.3: Percentage of masked and benign faults per each campaign. Bold text shows the best performance. Bold text identify the best in column.

5.3.6 Correlation between injected bit and resilience

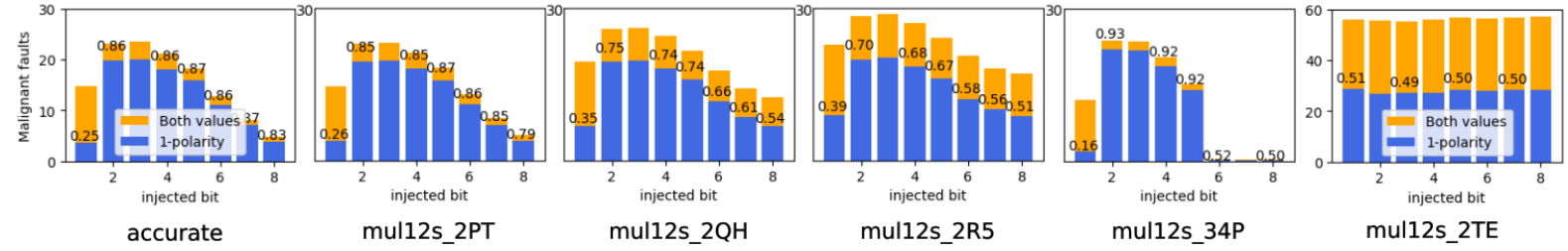
Figure 5.3b shows the number of malignant faults (normalized by the number of injections) with respect to the injected bit. The yellow bar represents the total number of FIs, while the blue bar is only referred to 1-polarity injections. The superimposed text indicates the percentage of 1-polarity faults. Bit 1 is the most significant, while bit 8 is the least significant. The weights are encoded as signed 8-bit integers.

The first four (from left to right) graphs show the same trend. The first bit is the sign bit, and it is less critical than the second one. The second to the last bits show a decreasing criticality, which is expected since the least significant bits have smaller values. Furthermore, 1-polarity injections are more critical for every bit but the first. This behavior is due to the regularization of the weights, which squeezes all the weights around zero, making it less common to have ones in the higher bits. Finally, the first bit is most critical to 0-polarity injections. This depends on the data distribution: the majority of injections did not change the actual value, writing a 1 where there already was a 1, but in general, we see that 0-polarity faults are more critical, representing the 73% of malignant outcomes. The last graph is related to multiplier 2TE, which (as seen before) is the worst multiplier. In this case, we can see that the approximation noise introduced by the multiplier is so much that is indistinguishable from actual information. It is clear that about 50% of the injection produced malignant outcomes and about half of those were produced by 0-polarity faults.

The most interesting results are produced by the multiplier 34P. In this case, FIs in the three Least Significant Bits (LSBs) seldom resulted in malignant faults (less than 1% of the injections per each bit). Furthermore, 90% of the malignant faults were produced by 1-polarity injections in bits 2 to 5. This datum can be used to deploy reliable hardware by simply masking such a fault with a 0 value.



(a) General resilience for each experiment. Each figure describes the distribution of the fault categories per experiment.



(b) Number of malignant faults per bit normalized to the number of injections. Bit 1 is the Most Significant Bit (MSB).

Figure 5.3: On the top row: experimental results showing fault classification for each multiplier. On the bottom row: malignant faults in for each bit and multiplier.

5.3.7 Resilient approximate multiplier

Multiplier 34P produced a very interesting result, since more than 74% of the injections produced masked faults. This is due to the small variation of the output of the multiplier when changing the three LSBs. In order to show this concept, we introduced a new local metric. We gathered the approximate multipliers, and we computed all the possible combinations storing both inputs and outputs. Then, we grouped the data so that for each bucket, only the three LSBs of the input of the weight varied. Finally, for each bucket, we computed the mean m and the variance σ of the outputs o_i of the bucket.

$$m = \frac{1}{N} \sum_{i}^{N} o_i \tag{5.3}$$

$$\sigma = \frac{1}{N} \sum_{i}^{N} (o_i - m)^2 \tag{5.4}$$

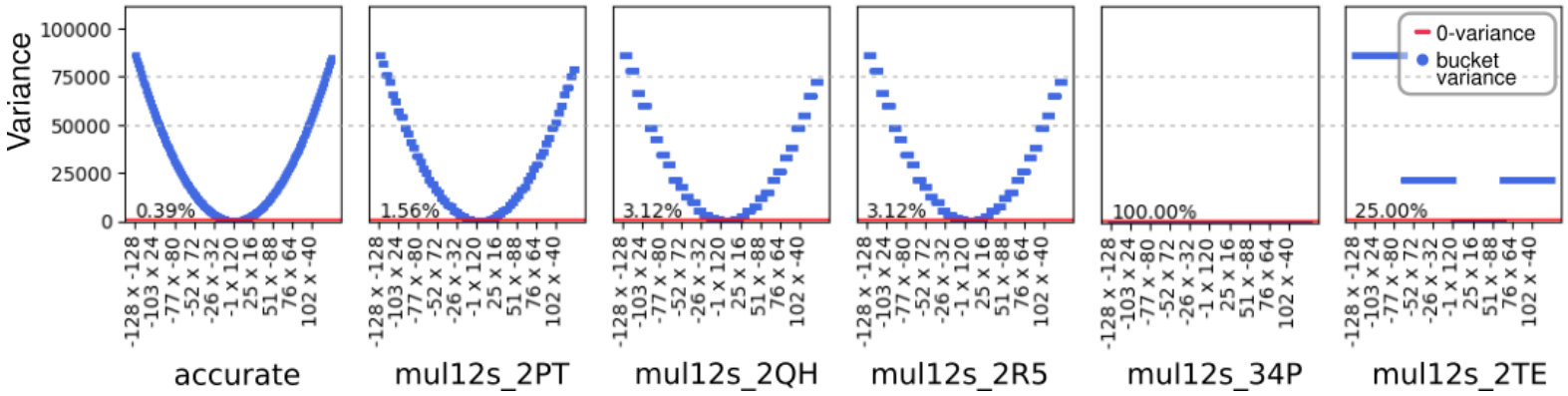
Figure 5.4a shows the variance plotted for each multiplier for each input. On the x axis there are the different combinations of inputs allowed (excluding the three LSBs obviously), while on the y axis there is the variance. The red lines and the superimposed text show how many points are strictly under variance 1, that is, inputs that produce the same output when varying the three LSBs.

The first multiplier is the accurate one. It shows quadratic behavior as it is supposed to be: the bigger the operands, the greater the distance among the outputs, thus the higher the variance. The graph should be 3-dimensional, as the two inputs a and b reside in orthogonal spaces. Nevertheless, in this figure, they are squeezed together, and adjacent points actually have the same variance, for the reason explained later. This implies that the parabola shown in the figure is composed of many horizontal segments.

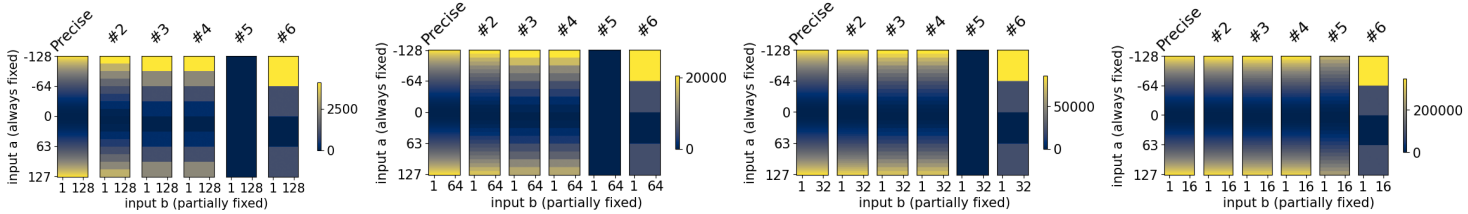
The multiplier has two inputs a and b. The output is computed as $o = b \cdot a$. If we *name* the outputs based on input b (thus fixing the value of a), we have $o_i = b_i \cdot a$. For each pair o_i and o_j there is a difference which can be computed with the formula $d_{ij} = |o_i - o_j| = |i - j| \cdot a$, simply because a is fixed. With this concept in mind, is simple to understand that the variance of successive buckets with the same value of a has the same variance. Furthermore, the variance decreases as a decreases, since the spreading among values is proportional to a.

As is to be expected, multipliers 2PT, 2QH and 2R5 show the same quadratic behavior although noisier. Multiplier 2TE shows a completely different behavior: the outputs vary very little in really big ranges. This datum is a confirmation of MAE and MRE values. Multiplier 34P shows a very interesting behavior since all the points have 0 variance. Later on, we found out that the three LSBs were completely disregarded in the computation, leading to the results at hand.

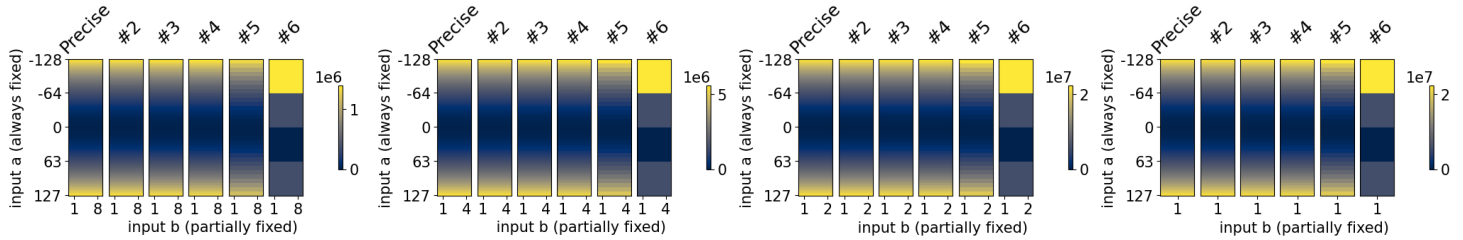
To study how the output (c) of a multiplier ($a \times b = c$) changes according to the possible input combinations (a, b), the following experiments were performed. Specifically, we quantified how much the output varies when the LSBs of b change. To carry out this analysis, we fixed all the bits of a and $(8 - n)$ of the MSBs of b. The remaining n bits of b were varied from 0 to 2^n while storing the output of the multiplier. Finally, the mean and variance of the gathered outputs were computed. This process is repeated $2^{8+(8-n)}$ times, once for each combination of the fixed inputs, thus generating the heatmaps of figures 5.4b to 5.4i. Figures 5.4b to 5.4i show the same process for different values of n. For instance, in Figure 5.4b, we vary only one bit (thus $n = 1$), in Figure 5.4c, we vary two LSBs (thus $n = 2$) and so on until Figure 5.4i in which all bits are considered ($n = 8$). Each heatmap reports the values of a's bits on the y-axis, the values of b's fixed bits on the x-axis, and the value of the variance is represented using colors: blue means no variation, yellow high variation. Zero-variance means that, given a's and b's fixed input bits, the output is always the same, regardless of the n LSBs of b. Please note that in our experiments, **the input b corresponds to the 8-bit input of the DNN weights** (i.e. the injected inputs).



(a) Variance of the output per each input combination varying on the three LSBs. The y-axis is the value of the variance. The red-line and the superimposed percentage show how many points have variance strictly less than 1.



(b) 1-bit variation (c) 2-bit variation (d) 3-bit variation (e) 4-bit variation



(f) 5-bit variation (g) 6-bit variation (h) 7-bit variation (i) 8-bit variation

Figure 5.4: On top: variance of the output as explained in subsection 5.3.7. Below: outputs variance of multipliers for a specific combination of fault-free 8-bit inputs (y-axis) and faulty 8-bit inputs (x-axis), where a growing number of bits varies.

For example, figure 5.4d shows the variance of the outputs when varying the three LSBs of b. This means that each point of the heatmap is associated with 13 fixed bits (8 bits for input a and 5 bits for input b). The remaining three are varied, the outputs gathered and variance computed, converted to a color, and plotted in the corresponding spot. Please note that NN's weights are mapped always on the input b of the multiplier, which is the same input we injected in the previously explained experiments.

Firstly, the simple use of bit-width reduction reduces by a factor of 2 the amount of critical faults. Second, the approximate multiplier significantly impacts the resilience and the impact depends on the adopted multiplier.

Interested by the result, we tried to expand the experiments to other NNs. Such experiments are explained in the next section.

5.4 The Effect of Approximate Multipliers on NNs

The effect of multiplier 34P, shown in the past section, tickled our curiosity and got us wonder: "maybe the approximate multipliers have beneficial effects on the NN?" Spoiler: no. At least, not always. And this section demonstrates this concept by describing a new set of experiments similar to the case study of the past section. For the sake of time, the experiments were carried out with a different type of fault: a memory fault, affecting the weights, without taking into account the SA.

5.4.1 Neural Networks

In order to properly evaluate the effectiveness of the technique, we employed a few pretrained NNs. More information on the architectures were given in subsection 2.1.4. Table 5.4, reports all the data.

Network	Dataset	Accuracy (%)
LeNet-5 v2	MNIST	97.09
AlexNet	Cifar10	82.73
ResNet-14	Cifar10	85.71
ResNet-20	Cifar10	83.88

Table 5.4: List of all the NNs used for the experiments

Unfortunately, the training (hyper-)parameters (included the quantization's) went lost, as the training was done by different people from different laboratories and the trained NNs were sent to us. Although it did not seem relevant at that time, we believe training is one of the most important aspect of the resilience, but this aspect is discussed more in detail in section 5.4.4.

As a final note to this section, we want to point out that, although LeNet-5 is the same architecture as the one in section 5.3.1, we superadd the wording "v2" to clarify that the training is different from the one presented on the last section. This fact is also corroborated the accuracy which is different.

5.4.2 Approximate Multipliers

https://ehw.fit.vutbr.cz/evoapproxlib/?folder=multiplers/12x12_signed

Expanding on the experiments of the section 5.3, we use the same library EvoApproxLiteLITE library [27]. Nevertheless, in this case the quantization has been performed differently, so that all the multiplications are signed, also those on 8 bits. Differently than before, in this case we can use 8bit multipliers, as there is no mixed-sign operation. Per contra, we will repeat the experiments with the 12bit multipliers, for comparison.

Tables 5.5 and 5.6 show all the multipliers used for the experiments, alongside with their metrics (MAE, WCE). Please note that the metrics are reported both with the suffix "-8", as per explained in

section 5.3.2, nevertheless such metrics coincide with the *raw* metrics for the 8 bits multipliers, and the suffix was maintained coherence. It should be also noted that that multipliers `mul8s_1KV6` and `mul12s_2PP` actually corresponds to the exact multipliers, thus the error is always 0. The tables also report the power and the area of the circuit, estimated with Synopsys DC (45nm PDK, 1V, 25°C).

The terms "accurate", "precise" and "exact" are used interchangeably and have all the same meaning.

Name	MAE-8 %	WCE-8 %	Power (mW)	Area (μm^2)
mul8s_1KV6	0.00	0.00	0.425	729.7
mul8s_1KV8	0.0019	0.0076	0.422	711.0
mul8s_1KV9	0.0065	0.026	0.410	685.2
mul8s_1KVP	0.051	0.21	0.363	635.0
mul8s_1L2J	0.081	0.39	0.301	558.9
mul8s_1L2L	0.23	1.16	0.200	411.6
mul8s_1L2N	0.52	2.66	0.126	284.9
mul8s_1L12	3.08	12.30	0.052	172.2

Table 5.5: Characterization of the approximate multipliers. Columns MAE-8 and WCE-8 are Mean Absolute Error and Worst Case Error respectively.

Name	MAE-8 %	WCE-8 %	Power (mW)	Area (μm^2)
mul12s_2PP	0.0	0.0	1.210	1650.5
mul12s_2PQ	0.0004	0.0015	1.205	1644.4
mul12s_2PR	0.0019	0.0076	1.200	1631.8
mul12s_2PS	0.0065	0.0259	1.189	1605.9
mul12s_2PT	0.0191	0.0748	1.168	1564.2
mul12s_2QH	0.0706	0.343	1.055	1443.6
mul12s_2R5	0.1550	0.685	0.948	1395.2
mul12s_34P	0.5190	2.66	0.579	952.0
mul12s_2TE	3.0800	12.3	0.497	874.8

Table 5.6: Characterization of the approximate multipliers. Columns MAE-8 and WCE-8 report the values of the metrics computed when restricting the multiplier at 8 bits.

Note: also in this case we will omit the strings "mul12s" or "mul8s", using just the three (or four) character ID to identify the multipliers. This means, for example, that "mul12s_2PP" and "2PP" identify the same multiplier. Similarly, "mul8s_1L12" becomes "1L12".

Inherent NN resilience

Approximating the multipliers means introducing errors to the computation To quantify such an error, we computed the classification accuracy of the NNs when the approximate multipliers are used. Figure 5.5 shows the baseline accuracy of the networks when the approximated multipliers are used on convolutional and linear layers. The red dash-dotted line shows the 90% of the baseline accuracy (i.e. the one computed with the accurate multiplier). The same data is reported also in table 5.7, to have precise figures. This will be used later since we consider unacceptable an accuracy drop higher than 10% and no fault injection is conducted on NNs which such a bad performance. The figure shows that if the noise introduced by

the approximation is not too much, the NN tolerates it, impacting little or nothing on the accuracy. In facts, the first four multipliers (for both the 8bit and the 12bit sets) impact the NN very slightly. Starting with 1L2J and 2PT the two sets exhibits different behaviors: in general, the 12bit multipliers make the network more resilient to faults, while 8bit multipliers degrade quite faster in comparison. Please note that this behavior solely depends on some aspect of the circuit that is not captured by our metrics. To better illustrate this point, look at figure 5.6. The figure shows, in a black solid line, the accuracy of AlexNet against the different multipliers, in a red dotted line the MAE of each multiplier, in increasing order, regardless of the group (8bit or 12bit). It is visible that, even though some multipliers have the same MAE value, they affect the NNs in different ways. We found similar results also for the other NNs and metrics. The lack of correlation between these two quantities highlights the unsuitability of the classical error metrics to the problem at hand. We suspect the local behavior of the multiplier should be taken into account to provide better metrics, correlating the accuracy drop.

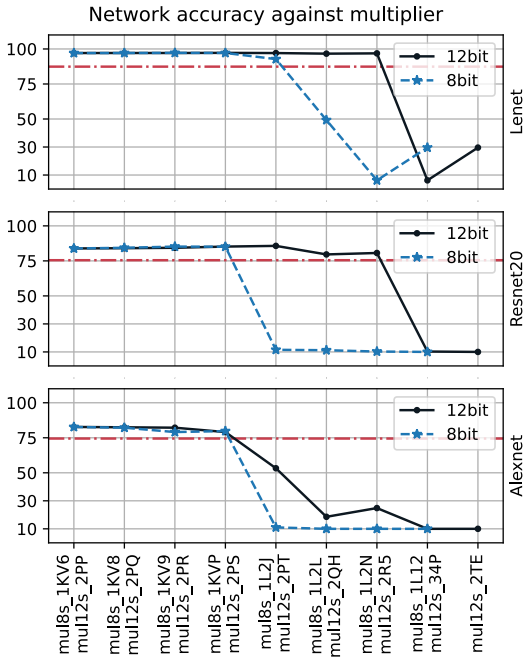


Figure 5.5: Neural Network accuracy variation against the different approximate multipliers. The x-axis shows the multipliers ordered in increasing approximation from left to right. The y-axis shows the classification accuracy over the test dataset. The red line indicates the 90% threshold below which the performance is considered not acceptable anymore. The figure only reports three NNs, since the two ResNets present almost the same performance.

Network	2PP	2PT	2QH	2R5	34P	2TE
LeNet5	97.09	92.24	**97.35**	97.03	79.83	34.81
ResNet14	85.71	**86.60**	82.67	83.01	9.71	10.00
ResNet20	83.88	**85.72**	79.55	80.65	9.73	10.00

Table 5.7: Baseline classification accuracy (%) with different approximate multipliers. Bold is best in row.

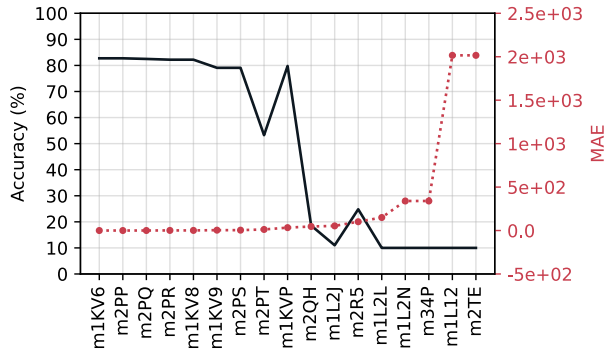


Figure 5.6: AlexNet accuracy and multiplier Mean Absolute Error. In black: accuracy of AlexNet. In red: Mean Absolute Error of the multiplier in logarithmic scale.

5.4.3 Fault Injection

Each fault injection was constituted by a number of faults big enough to obtain statistical significance. We employed the technique explained in [30], using the following formula:

$$n_{i,l} = \frac{N_{i,l}}{1 + e^2 \cdot \frac{N_{i,l}-1}{t^2 p_i (1-p_i)}} \Rightarrow n = \sum_l \sum_i n_{i,l},$$

where $N_{i,l}$ is the population of faults that can affect the bit i in layer l, p_i is the probability of bit i to result in a failure when flipped, $n_{i,l}$ is the number of faults to inject for each subpopulation and n is the number of faults during the whole fault injection campaign.

For each DNN the number of fault to inject is computed by using the following parameters:

- e is the desired error margin (5% in our experiments);
- t correspond to the confidence level ($t = 2.58$ for us, implying a 99% confidence level);
- N is the population size (for us, since we only consider bit-flips, it corresponds to the number of weights in the layer);
- p_i correspond to the probability that a fault in the i-th bit results in an error.

Network	N	n
LeNet5	44.2k	3668
ResNet14	1.2M	3702
ResNet20	19.6M	3704
AlexNet	34.1M	3704

Table 5.8: Injected faults (n) per each network. N is the number of weights of the network.

The number of faults is determined using a variation of the method explained in section 2.2.4, based on [30]. We computed the failure probability p_i using the average distance, as explained in [30]. The average distance is a measure of the effect the bit-flip has on the weight. In fact, it corresponds to the distance between the golden weight and the faulty weight after the i-th bit has flipped. Since we are dealing with 8-bit integers, the average distance per bit simply corresponds to 2^i, where $i = 0, \dots 7$ describes the bit. Bit 0 corresponds to the MSB, while bit 7 is the LSB. Normalizing the distance with $a = 0.01$ and $b = 0.5$ gives us the following probabilities:

$$p = [0.50, \quad 0.25, \quad 0.13, \quad 0.07, \quad 0.04, \quad 0.02, \quad 0.01, \quad 0.01]$$

Using this datum, we computed the number of fault to injected in each network. As a good measure, we injected double the amount of faults computed like this. Table 5.8 reports this figures, alongside with N. Each fault is evaluated running 100 inferences from a random sample of the dataset. Each fault has a different random subset composed of 100 images.

5.4.4 General Resilience

Figure 5.7 reports the results of the Fault injection campaign for each of the networks. On the x-axis there is the approximate multiplier, in order of increasing error from left to right; on the y-axis there is the percentage of faults in each category. The three points for each multiplier add up to 100%. As said earlier, we performed fault injection only on those configurations whose accuracy would drop less than 10% with respect to the baseline (computed with the accurate multiplier).

The first notable result is that, even tough different approximation are given, the resilience of the network is mostly unchanged, decreasing slightly with the most approximated multipliers. This is true for both sets. Moving on, we can differentiate between applications, implying that there is a difference in resilience depending on the neural architecture: in fact, for LeNet and the two ResNets the majority of faults are tolerable; for AlexNet, the majority are masked. On top of that, as also shown in the figure, the three multipliers 2PT, 2QH and 2R5 have a different effect on AlexNet, making the unsuitable for an injection campaign.

5.4.5 Layer-specific resilience

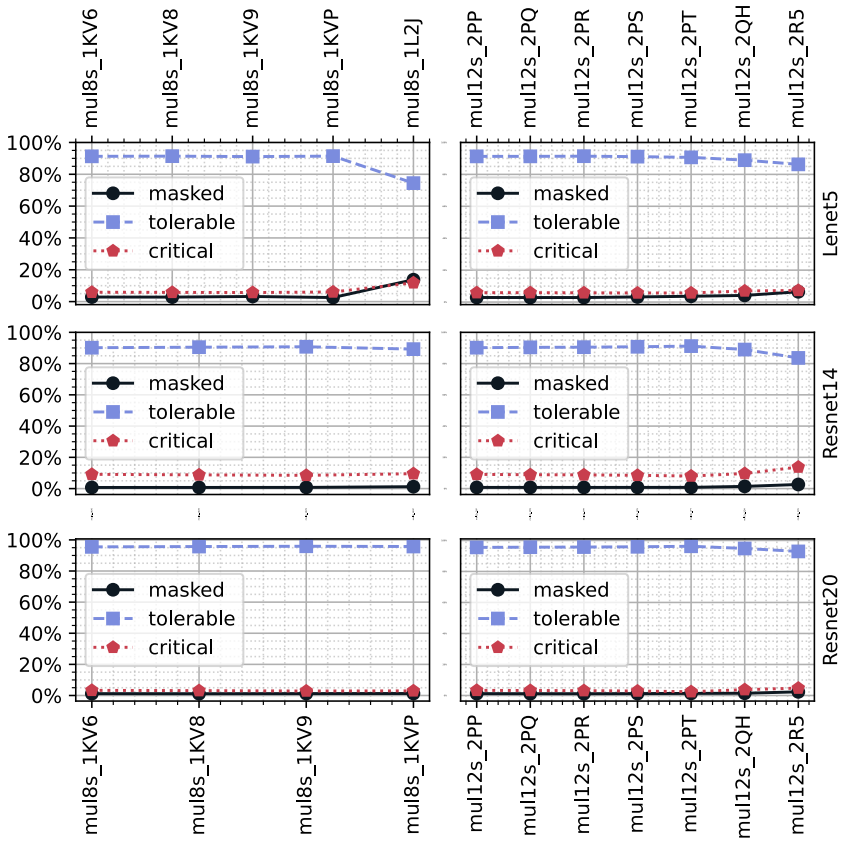
It is well known that different layers have different resilience [29]. However, our results show that it might not be always the case. Figure 5.8 shows the resilience with respect to each of the injected layers. It is possible to see that there is no statistical difference among the layers. The only network that shows a different behavior is ResNet20, but also in this case, the difference is minimal and could be due to the statistical method and its implementation. Similar results are shown for every fault injection campaign, regardless of the multiplier employed. This result suggests the necessity to investigate a DNN result by taking into account also its training method. Most of the literature work of today, does not take into account this aspect of the resilience, focusing solely on the hardware aspect of the problem.

5.4.6 Discussion

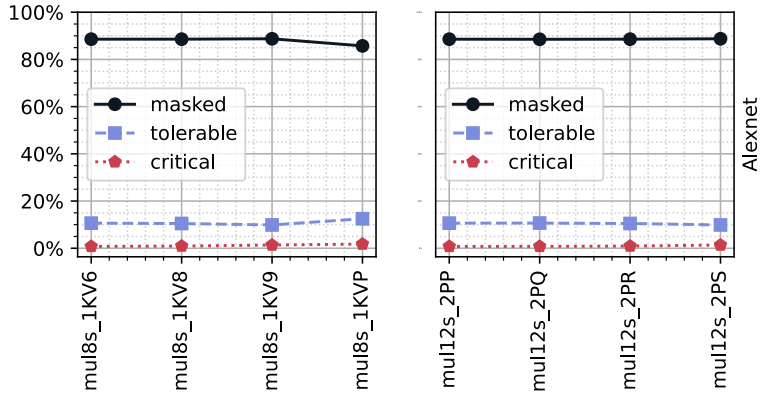
These results were quite unsatisfying, as the case study presented in section 5.3 seemed very promising. The feeling we have, is that most of the resilience actually comes from the training of the NN. Nevertheless, the space of the training hyperparameters has many axes and is not obvious in what direction to go. Some preliminary experiments on data augmentation suggest that augmentation is not the way to go. Maybe better results can be obtained through regularization, but this is an aspect to explore in the future.

Bit-width reduction

The bit-width reduction is a classical approximation technique [20] for which the size of the data manipulated by the application is reduced with respect to the precise implementation. In the context of DNNs, the bit-width reduction is better known as quantization since the DNN parameters (i.e. weights, bias, ...) are "converted" from the 32-bit floating-point representation to $n - bit$ integer representations. From the energy point of view, the bit-width reduction allows reducing memory footprint and data movement. The systolic array has been implemented with two data representations: the INT 16-bit as the reference (i.e. precise) and the INT 8-bit as the approximated precision.



(a) General resilience for LeNet, ResNet14 and ResNet20.



(b) General resilience for ResNet14.

Figure 5.7: General resilience of the NNs. On the abscissa there are the approximate multipliers, on the ordinate there is the percentage of each class of fault.

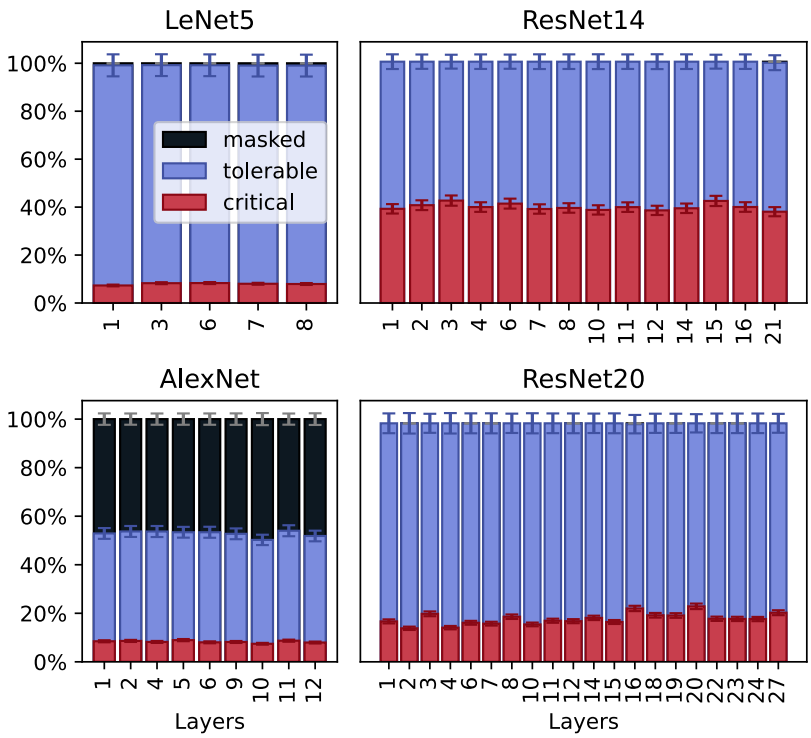


Figure 5.8: Resilience with respect to each injected layer for the accurate multiplier.

5.5 Conclusions

The results obtained in these work FI campaigns suggest that an approximate multiplier can be used for NN computations making the whole NN more resilient. Furthermore, we can identify two properties that the multiplier should have in order to increase the resilience of the NN:

- the approximate output must be always somewhat similar to the accurate computation (MAE and MRE are good metrics in this regard),
- the output value should vary very little (or nothing) when changing the LSBs of the inputs.

The first property is fundamental for approximate computing. In general, the smaller the error of the approximate multiplier, the better the multiplier is. The second property is more interesting and can be used to build new metrics for quantifying the resilience of a NN. Nevertheless, our method does not investigate the effect of all the input bits of the multiplier, so more effort has to be put in the investigation of such phenomena. This method could also be used, in principle, for a general circuit that is put in a NN. For this reason, the next direction of investigation will be building a proper metrics to quantify the resilience of circuits.

In general, approximate multipliers can be really useful in the context of NNs. They use less energy than accurate circuits and, as seen, may provide more accuracy. They can also be used to increase the resilience at the cost of some accuracy loss (which may or may not be acceptable depending on the application). We showed that some multipliers are best suited for increasing the resilience of the NN. Specifically, one of the multipliers increased the number of masked faults by an outstanding 9.3% of masked faults. Furthermore, we provided a simple method to understand why this specific multiplier boosted the performance so much, which also gave some useful insights for the design of resilient circuits.

References

[1] S. Sabour, N. Frosst, and G. E. Hinton, *Dynamic routing between capsules*, Nov. 7, 2017. DOI: 10.48550/arXiv.1710.09829 arXiv: 1710.09829[cs]. Accessed: Oct. 5, 2025. [Online]. Available: http://arxiv.org/abs/1710.09829

[2] G. Desoli et al., "14.1 a 2.9 tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," in *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, IEEE, 2017, pp. 238–239.

[3] A. Bosio, D. Ménard, and O. Sentieys, Eds., *Approximate Computing Techniques*. Springer International Publishing, 2022. DOI: 10.1007/978-3-030-94705-7 [Online]. Available: https://doi.org/10.1007/978-3-030-94705-7

[4] E. Dupuis, S. Filip, O. Sentieys, D. Novo, I. O'Connor, and A. Bosio, "Approximations in deep learning," in *Approximate Computing Techniques*, Springer International Publishing, 2022, pp. 467–512. DOI: 10.1007/978-3-030-94705-7_15 [Online]. Available: https://doi.org/10.1007/978-3-030-94705-7_15

[5] M. Ahmadinejad and M. H. Moaiyeri, "Energy- and quality-efficient approximate multipliers for neural network and image processing applications," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 2, pp. 1105–1116, Apr. 2022, ISSN: 2168-6750. DOI: 10.1109/TETC.2021.3072666 Accessed: Oct. 12, 2025. [Online]. Available: https://undefined/abstract/document/9403977

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791

[7] H. Mo et al., "Learning the error features of approximate multipliers for neural network applications," *IEEE Transactions on Computers*, vol. 73, no. 3, pp. 842–856, Mar. 2024, ISSN: 1557-9956. DOI: 10.1109/TC.2023.3345163 Accessed: Oct. 12, 2025. [Online]. Available: https://undefined/abstract/document/10372208

[8] M. S. Ansari, V. Mrazek, B. F. Cockburn, L. Sekanina, Z. Vasicek, and J. Han, "Improving the accuracy and hardware efficiency of neural networks using approximate multipliers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 317–328, Feb. 2020, ISSN: 1557-9999. DOI: 10.1109/TVLSI.2019.2940943 Accessed: Oct. 12, 2025. [Online]. Available: https://undefined/abstract/document/8863138

[9] J. Wu et al., "Aircraft flight regime recognition with deep temporal segmentation neural network," *Engineering Applications of Artificial Intelligence*, vol. 120, p. 105 840, 2023.

[10] S. Ghanem, P. Kanungo, G. Panda, and P. Parwekar, "An improved and low-complexity neural network model for curved lane detection of autonomous driving system," *Soft Computing*, vol. 27, no. 1, pp. 493–504, 2023.

[11] A. Lotfi et al., "Resiliency of automotive object detection networks on gpu architectures," in *2019 IEEE International Test Conference (ITC)*, IEEE, 2019, pp. 1–9.

[12] B. Salami, O. S. Unsal, and A. C. Kestelman, "On the resilience of rtl nn accelerators: Fault characterization and mitigation," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, IEEE, 2018, pp. 322–329.

[13] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A reliability analysis of a deep neural network," in *2019 IEEE Latin American Test Symposium (LATS)*, IEEE, 2019, pp. 1–6.

[14] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021. DOI: 10.1109/TVLSI.2020.3048829

[15] S. Burel, A. Evans, and L. Anghel, "Mozart+: Masking outputs with zeros for improved architectural robustness and testing of dnn accelerators," *IEEE Transactions on Device and Materials Reliability*, vol. 22, no. 2, pp. 120–128, 2022. DOI: 10.1109/TDMR.2022.3159089

[16] A. Ruospo, E. Sanchez, L. M. Luza, L. Dilillo, M. Traiola, and A. Bosio, "A survey on deep learning resilience assessment methodologies," *Computer*, vol. 56, no. 2, pp. 57–66, 2023. DOI: 10.1109/MC.2022.3217841

[17] J. Knight, "Safety critical systems: Challenges and directions," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, May 2002, pp. 547–550. Accessed: Sep. 21, 2025. [Online]. Available: https://ieeexplore.iee.org/document/1007998

[18] G. Di Natale et al., *Cross-layer reliability of computing systems*. iet-the institution of engineering and technology, 2020.

[19] A. Ruospo et al., "Selective hardening of critical neurons in deep neural networks," in *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, IEEE, 2022, pp. 136–141.

[20] G. S. Rodrigues, F. L. Kastensmidt, and A. Bosio, *Approximate Computing and its Impact on Accuracy, Reliability and Fault-Tolerance*. Springer International Publishing, 2022. DOI: 10.1007/978-3-031-15717-2 [Online]. Available: https://doi.org/10.1007/978-3-031-15717-2

[21] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, 62:1–62:33, Mar. 18, 2016, ISSN: 0360-0300. DOI: 10.1145/2893356 Accessed: Jun. 29, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/2893356

[22] P. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *IEEE Transactions on Nuclear Science*, vol. 51, no. 5, pp. 2957–2969, Oct. 2004, ISSN: 1558-1578. DOI: 10.1109/TNS.2004.834955 Accessed: Oct. 6, 2025. [Online]. Available: https://undefined/abstract/document/1344451

[23] M. Traiola, B. Deveautour, A. Bosio, P. Girard, and A. Virazel, "Test and reliability of approximate hardware," in *Approximate Computing*, Springer International Publishing, 2022, pp. 233–266. DOI: 10.1007/978-3-030-98347-5_10 [Online]. Available: https://doi.org/10.1007/978-3-030-98347-5_10

[24] M. Biasielli et al., "Approximation-based fault tolerance in image processing applications," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 2, pp. 648–661, 2021.

[25] L. M. Luza et al., "Emulating the effects of radiation-induced soft-errors for the reliability assessment of neural networks," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, no. 4, pp. 1867–1882, 2021.

[26] CEA-LIST, *N2D2*, [Online]. Available: https://github.com/CEA-LIST/N2D2.

[27] V. Mrazek et al., "Scalable construction of approximate multipliers with formally guaranteed worst case error," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 11, pp. 2572–2576, 2018.

[28] R. Hrbacek, V. Mrazek, and Z. Vasicek, "Automatic design of approximate circuits by means of multi-objective evolutionary algorithms," in *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era (DTIS)*, 2016, pp. 1–6. DOI: 10.1109/DTIS.2016.7483885

[29] A. Ruospo et al., "Investigating data representation for efficient and reliable convolutional neural networks," *Microprocessors and Microsystems*, vol. 86, p. 104 318, 2021.

[30] A. Ruospo et al., "Assessing convolutional neural networks reliability through statistical fault injections," in *2023 Design, Automation Test in Europe Confe rence Exhibition (DATE)*, 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10136998

CHAPTER 6

Fault Propagation Prevention through Monitors

This chapter is dedicated to a technique that can be employed in Neural Network (NN) to prevent fault propagation. The basic idea is to compute some metrics in the output feature maps of the first layers and characterize them. After that, the inference can be halted and re-executed if a fault is detected.

6.1 Preventing Fault Propagation

To safely and reliably adopt such architectures in safety-critical systems, it is needed to implement resilience mechanisms to safeguard against predictive failures. Indeed, several research works have shown that, with the advance of technological nodes, also SA models are susceptible to random-hardware faults [1]. In specific domains, such as autonomous driving, medical healthcare devices, humanoids and, more in general, robots, the safety and reliability reduction cannot be tolerated during runtime.

In this light, it is compelling to develop new strategies to in-field detect the occurrence of potential failures, so that the faults' impact and the vulnerabilities of deep Deep Neural Network (DNN) models are reduced during their operational lifetime.

Controlling the fault-free execution of a DNN inference and then repeating it has been also defined as Inference-level Parallelism (ILP). Reference [2] proposes a combination of techniques including ILP to increase the reliability of a Convolutional Neural Network (CNN) model. In detail, they propose to selectively rerun vulnerable CNN inferences by analyzing their output (specifically, the top-1 and the difference between the top-2 prediction scores). If these values fall below an acceptable level, the CNN inference is re-executed to mitigate the occurrence of transient hardware faults.

A fine-grained technique exploits tensor-related metrics, first introduced in [3], that are computed over the output feature maps (OFMAPs) of CNNs. Specifically, the *warning* metrics can be computed after any convolutional layer to indicate the presence of critical faults, and based on that, selectively rerun vulnerable inferences. The sooner (in the inference) the metrics are calculated, the quicker CNN inferences that are propagating incorrect values can be halted and executed again. In very deep CNNs, anticipating this check may be notably advantageous, as it will save many useless operations that would be discarded and re-executed if only the output is controlled in the field (by looking for instance to the

top-1 score).

This section proposes a low-cost technique to halt and rerun suspicious CNN inferences by leveraging simple tensor-related metrics as a warning mechanism; decreasing, in this way, the impact of critical faults (also known as Silent Data Corruption (SDC)-1 in image classification tasks) and saving execution time by stopping inferences preemptively.

In particular, the metrics are computed immediately after the first layer of a CNN running on a Systolic Array (SA), to avoid continuing the inference and propagating wrong values. Clearly, not all detected faults (after the first layer) turn into failures. So, it is highly important to develop an early detection mechanism that is capable of halting and re-executing only those faulty inferences that lead to an application failure. The experimental investigations have been performed on two different CNNs, a ResNet-20 and a MobileNetV2 models, trained with the CIFAR10 dataset, and running on a SA architecture [4]. The proposed technique, computed after the first CNN layer, can interrupt on average more than 95% of DNN inferences propagating critical faults (leading to failures). The number of inferences that are uselessly halted and re-executed is always lower than 0.4%. Moreover, analysis in terms of computational costs and FLoating-points Operations Per Second (FLOPS) operations needed to compute the metrics are provided.

6.2 Preliminaries

Systolic array architectures are rapidly gaining popularity within the research community for their efficient design and the ability to introduce a high degree of parallelism, enhancing computational throughput [5]. Among the most common and popular architectures suitable for DNN processing are [6], [7], [8], [9]. A SA is an array of highly regular interconnected Processing Elements (PEs) that exploit spatial locality to process data: the PEs are connected together to forward data in a pipelined fashion. Multiple types of SA exist, as explained in [5]. On top of that, SAs can be used to solve a variety of problems. The exploited SA is a matrix-multiplication engine configured in an output stationary configuration: the accumulation is directly performed on the single PE, while the operands (i.e. weights and activations) flow from top to bottom and from left to right. The designed SA is composed of $32 \times 32 = 1024$ PEs. Each single PE is made by three registers (activations, weights and accumulation), a multiplier and an adder, i.e. a Multiply And Accumlate (MAC) unit.

Recently, systolic arrays have been the target for several reliability studies. Multiple works have proposed the design of fault-tolerant hardware accelerators (e.g., [10]) or techniques to increase the fault tolerance of SA architectures (e.g., [1], [11]). As reported in the literature, SAs can produce catastrophic outcomes depending on the injected fault. Additionally, the authors in [12] experimentally demonstrate that the faults' criticality in systolic arrays also depends on the faulty PE. The authors in [11] present a strategy to on-line test the hardware to prevent functional failures. A further mitigation strategy has been presented in [10], where the authors proposed a small modification to the systolic array architecture, so that fault tests can be scheduled concurrently with the inference. If any element is found to be faulty, it is excluded from the computation.

6.3 Proposed Approach

This research work proposes an early stopping technique to in-field detect random-hardware faults and selectively rerun vulnerable inferences. The approach relies on the computation of tensor-related metrics, applied to OFMAPs following convolutional layers. The overall idea is shown in Figure 6.1. The approach proposes to compute low-cost metrics that act as decision thresholds to rerun a *suspicious* CNN inference. If the SA is affected by a transient fault, during the second inference, the new check may return a *pass* (meaning that the transient fault has disappeared) or a *critical warning* (the fault is still there).

Hence, the SA system enters a halt state (the end-user of the application can decide the safety strategy to adopt). The advantage of computing the metrics without concluding the entire faulty inference (to check the output vector) is obtained in terms of saved FLOPSs and saved energy-power consumption.

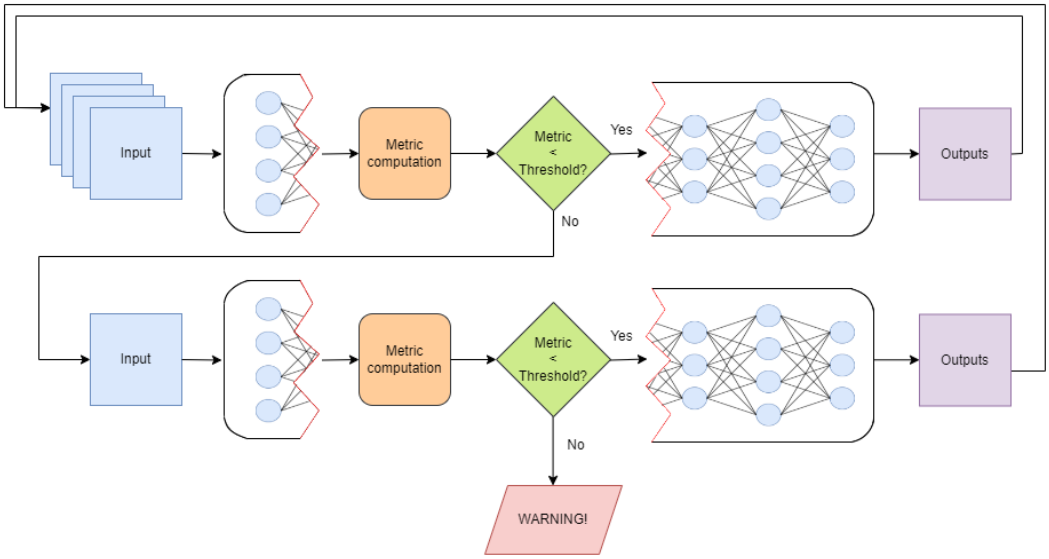
In line with the single tensor metrics proposed in [3], the investigated metrics are the following:

- **tensorMax**: consists in identifying the maximum value of the OFMAP;
- **tensorMean**: it is the arithmetic average of the values of the OFMAP;
- **tensorSum**: computes the sum of all the values of the OFMAP tensor.

In this work, the metrics have been computed to the output of the first layer, before the activation function. To assess the proposed early stopping method, FIs are performed on a SA simulator executing the first layer of the targeted CNNs. The faulty outputs are then used to determine whether an early stopping should be activated. The entire process is executed in two parts: (i) a systolic array simulation takes care of the computation of the first layer, (ii) then the metrics are computed, and the inference is completed using PyTorch from the second layer until the CNN's output. At the same time, by means of FIs, the effects of faults are computed to validate the effectiveness and the accuracy of the method.

It is worth to underline that, to execute an online early stopping and fault classification, the metrics computation must be done before the end of the inference. Specifically, it is possible to compute such metrics in two ways: (i) via hardware, modifying the SA to compute them while performing the convolution, or (ii) via software allowing the host processor to compute them using general matrix multiplication routines. This work covers the second scenario, as the first one requires a redesign of the SA, as it will be the goal of a next research activity.

Figure 6.1: Proposed Flow



6.4 Experimental results

The proposed technique has been experimentally validated on two CNN: a ResNet-20 and a MobileNetV2, trained for image classification on CIFAR10, and achieving 91.48% and 91.73% classification accuracy, respectively. Experiments have been executed exploiting the flauers (cf. 3). The designed

SA can accommodate the first layer of both CNN, as it is composed of $32 \times 32 = 1024$ PEs. A single matrix multiplication takes $n + 32 + 32$ clock cycles to complete, where n is the length of a row of the input matrix. In order to run the experiment efficiently, we used the same strategy of the one described in [4]. This allowed the inference to be executed in $\sim$1 minute per 100 input images per injected fault.

Transient faults have been randomly injected on the weight register of a random PE of the considered systolic array. Specifically, each fault affects a single bit of the targeted register storing the CNN weight. A statistical subset of 38,500 faulty inferences has been selected (100 random images for each fault), obtaining estimates with an error of 5% and a confidence of 95%. Depending on their effects, injected faults have been classified as critical and non-critical. Among the total faulty inferences (38,500), a total of 2.52% and 2.29% of critical faults have been identified in ResNet20 and MobileNetV2, respectively. The remaining ones have been classified as non-critical.

Each register is 32-bit wide and data representation follows the IEEE 754 floating-point standard. The injected fault only affects the computation of the first OFMAP, so the rest of the inference is assumed to be fault-free. For setting the thresholds, we computed the metrics at the output of the first layer for each input image of the CIFAR10 test-set, without injecting the faults. The obtained values form a distribution of the fault-free inference values. Thresholds are then set on the 100-th percentile on the right side. In other words, we used the *maximum value* as a threshold for the three metrics.

Figures 6.2a and 6.2b provide a compact view of the obtained results: first, they plot the probability density function (PDF) of the metrics computed in a golden scenario for all the testset (10k images). The analysis has been computed on all the three metrics, but for sake of space, only the *tensorMax* is illustrated. It is important to underline that the general behavior is confirmed.

In both figures, the y-axis represents the PDF of the metric's golden distribution, while the x-axis represents a range of values of the chosen metric.

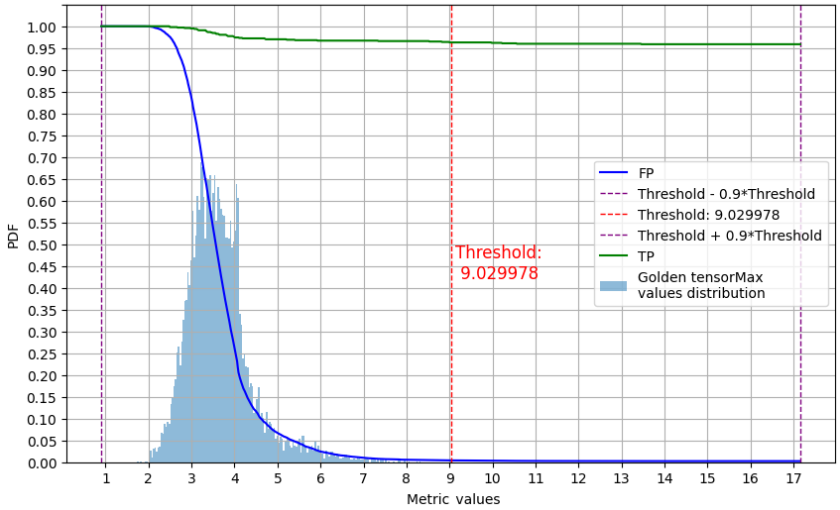
The vertical dashed red lines indicate the maximum values for the tensorMax metric, 9.03 and 8.16 for ResNet-20 and MobileNetV2, respectively. As above-mentioned, those maximum values are used as threshold to activate the early stopping mechanism and to rerun suspicious CNN inferences. The second information provided in the same graphs is the variation of the True Positive (TP) and False Positive (FP) values, showed with the green and the blue lines. The two curves describe the trend of DNN inferences propagating critical faults that have been correctly re-executed (TP) and DNN inferences propagating non-critical faults that have been unnecessarily re-executed (FP), given a specific threshold variation of $\pm$90%. This second parameter (FP) is an index of the computational cost sacrificed to apply the fault-tolerance strategy. Table 6.2 contains a comprehensive description of the four confusion matrix indexes.

DNN inferences that propagates critical faults		**DNN inferences that propagate non-critical faults**	
TP	**FN**	**FP**	**TN**
Early detection correctly activated	Early detection wrongly not activated	Unnecessary early detection activated	Early detection correctly not activated

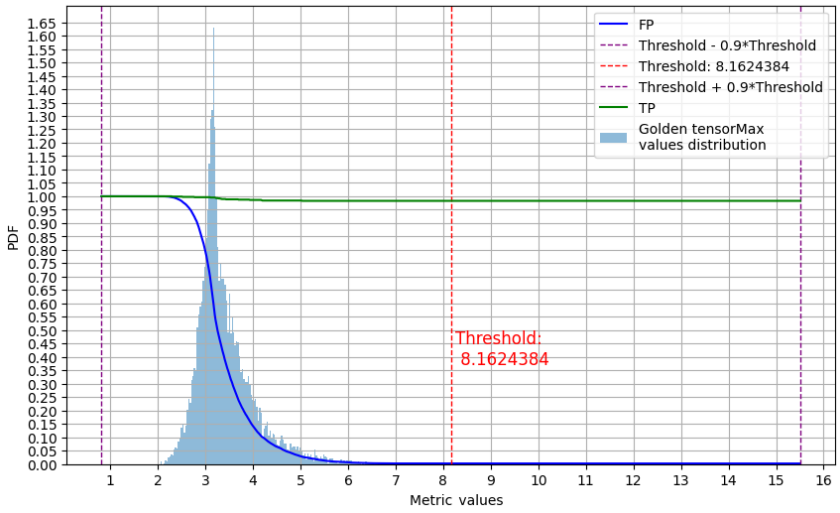
Table 6.1: Definitions of the confusion matrix elements.

Focusing on the trend of the TP and FP curves on the plots, it emerges that with the two selected thresholds, very high TP values of around 96% are achieved, while maintaining relatively low FP values that discard between 0.25% and 0.43% of non-critical inferences. Increasing the thresholds, the changes in terms of TPs are minimal. Conversely, by lowering the thresholds, both curves show a slope, in the case of TPs, more critical inferences can be identified, but at the same time, the number of FPs increases exponentially, making the threshold computationally costly and inefficient.

Table 6.2 reports complete results for all the three considered metrics; in all cases, the threshold has



(a) ResNet20



(b) MobileNetV2

Figure 6.2: These graphs illustrate the distribution of the tensorMax values computed over the golden model inferences of the testset for both CNNs. It overlaps information of the TP and FP values for a given threshold (vertical dashed lines).

been fixed at the maximum value of the metric among the golden inferences executed to collect the metrics.

It is crucial to emphasize that in this context, the overhead cost is represented by the FP, the critical inferences that are not interrupted. Conversely, the cost in terms of safety and reliability is represented by the FN, the non-critical inferences wrongly interrupted. The results indicate that the TP values are notably high while the FP values remain low, which is a highly positive outcome. Notably, there are minor differences in the metrics for ResNet-20, whereas the metrics for MobileNetV2 exhibit near-identical behavior. Interestingly, the strategy employed for MobileNetV2 results in a TP inference coverage that surpasses that of ResNet-20 by approximately 2.4%, while maintaining similar FP rates. This suggests that MobileNetV2 achieves higher TP rates without a significant increase in FP, demonstrating superior efficiency and reliability.

			Critical Faults		**Non-critical faults**	
Model	**Metric**	**Threshold**	**TP**	**FN**	**FP**	**TN**
ResNet-20	tensorMax	9.03	96.28%	3.72%	0.43%	99.57%
ResNet-20	tensorMean	0.83	95.87%	4.13%	0.26%	99.74%
ResNet-20	tensorSum	13662.50	95.87%	4.13%	0.25%	99.75%
MobileNetV2	tensorMax	8.16	98.28%	1.72%	0.23%	99.77%
MobileNetV2	tensorMean	0.56	98.30%	1.70%	0.25%	99.75%
MobileNetV2	tensorSum	18387.30	98.33%	1.67%	0.24%	99.76%

Table 6.2: Confusion matrix given the selected threshold.

6.5 Computational Costs

Computing the metrics has an overhead over the cost of the inference itself. We can compute the computational cost of the entire CNN in terms of MACs. Please note, that the computation costs mentioned corresponds effectively to the overhead needed to employ the technique. Nevertheless, roughly speaking, such overhead is negligible in comparison to the requirements of the convolution. A standard depth-wise convolutional (as the one implemented in ResNet) layer needs a total of:

$$M \cdot N \cdot D_F^2 \cdot D_K^2 \quad \text{MACs},$$

where N and M are respectively the number of input and output channels of the layer, D_F corresponds to the spatial size of the OFMAP and D_K is the spatial size of the kernel (both assumed squared). The cost of the other non-convolutional layers is negligible, as they occupy a small portion of our CNNs. As for MobileNetV2, the implemented convolution is a point-wise separable convolution, which has a different cost. Specifically, its cost is:

$$M \cdot D_K^2 \cdot D_F^2 + N \cdot M \cdot D_F^2.$$

Please note that the cost of a point-wise convolution is much smaller than the cost of the standard depth-wise one. As Table 6.3 reports, computing the first OFMAP only occupies about 1% of the total MAC operations of the CNN (1.1% and 0.83% for ResNet20 and MobileNetV2, respectively). This means that when a faulty inference is correctly stopped (our TP values always higher than 95%), the proposed technique can save ∼99% of the MACs required to complete that faulty inference.

To compute the metrics, it is necessary to iterate over all elements of the output tensor of the OFMAP, and the cost of computing that metrics is roughly equal to the shape of the tensor. Specifically, tensorMax and tensorSum metrics have the same computational cost. The tensorMean metric has an additional

division, accounting for an additional single MAC operation. The table also shows that the overhead produced by computing any of the three metrics is negligible, being in the order of 10^{-4} or 10^{-5} of extra MACs (with respect to the whole CNNs), as given in the last row of the column.

Future works will study the trade-off between number of metrics computations included in more layers of the CNNs and true positive rates of critical faults that are detected and not propagated.

Table 6.3: Computational costs of the proposed approach. The first row shows the total MACs needed for a single inference. The second row is the MACs required for the computation of the first layer only – in terms of the total. The fourth row shows the absolute number of MAC needed to compute the metric, the next row shows the same value with respect to the total inference, while the last row shows the overhead of computing the metric with respect to the cost of the first layer.

Details of computational costs	**ResNet20**	**MobileNetV2**
Total MACs	40.56e9	15.19e9
MACs for computing the first OFMAP	0.44%	0.13%
Shape of the first OFMAP	(32,32,16)	(32,32,32)
MACs for computing the metrics	16.38e3	32.77e3
Total Overhead	0.00004%	0.0002%
Layer Overhead	0.00009%	0.0017%

6.6 Conclusions

In this chapter, we explored a novel approach to prevent fault propagation in NNs by leveraging monitors that compute metrics in the output feature maps of the first layers. The core idea is to detect faults during inference and halt execution if anomalies are identified, thereby enhancing the reliability of NN operations. Our analysis demonstrated that the proposed monitoring technique introduces minimal computational overhead, as evidenced by the metrics computed for ResNet20 and MobileNetV2 architectures. The overhead is negligible compared to the total computational cost of the networks. This makes the approach feasible for real-world applications where reliability is critical, without significantly impacting performance. The results also highlighted the importance of selecting appropriate metrics for fault detection. By characterizing the output feature maps, we established a robust mechanism to identify and mitigate faults, ensuring that the neural network's inference remains accurate and reliable. While this work lays a strong foundation for fault prevention in NNs, several avenues for future research remain open. Further investigation is needed to explore the scalability of this approach across different network architectures and to refine the metrics for even more precise fault detection. Additionally, integrating this technique with other fault-tolerant mechanisms could provide a comprehensive solution for ensuring the reliability of neural networks in safety-critical applications.

References

[1] J. J. Zhang, K. Basu, and S. Garg, "Fault-tolerant systolic array based accelerators for deep neural network execution," *IEEE Design & Test*, vol. 36, no. 5, pp. 44–53, 2019. DOI: 10.1109/MDAT.2019.2915656

[2] A. Mahmoud et al., "Optimizing selective protection for cnn resilience.," in *ISSRE*, 2021, pp. 127–138.

[3] V. Turco, A. Ruospo, E. Sanchez, and M. S. Reorda, "Early detection of permanent faults in dnns through the application of tensor-related metrics," in *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, 2024, pp. 13–18. DOI: 10.1109/DDECS60919.2024.10508918

[4] M. Taheri et al., "Saffira: A framework for assessing the reliability of systolic-array-based dnn accelerators," in *2024 27th International Symposium on Design & Diagnostics of Electronic Circuits & Systems (DDECS)*, 2024, pp. 19–24. DOI: 10.1109/DDECS60919.2024.10508925

[5] V. Sze et al., "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[6] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017. DOI: 10.1109/JSSC.2016.2616357

[7] nvidia, *Nvdla*, 2021. [Online]. Available: https://nvdla.org/

[8] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 1–12, Jun. 2017, ISSN: 0163-5964. DOI: 10.1145/3140659.3080246 [Online]. Available: https://doi.org/10.1145/3140659.3080246

[9] H. Genc et al., "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[10] S. Burel, A. Evans, and L. Anghel, "Mozart+: Masking outputs with zeros for improved architectural robustness and testing of dnn accelerators," *IEEE Transactions on Device and Materials Reliability*, vol. 22, no. 2, pp. 120–128, 2022. DOI: 10.1109/TDMR.2022.3159089

[11] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021. DOI: 10.1109/TVLSI.2020.3048829

[12] S. Pappalardo, A. Ruospo, I. O'Connor, B. Deveautour, E. Sanchez, and A. Bosio, "Resilience-performance tradeoff analysis of a deep neural network accelerator," in *2023 26th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, IEEE, 2023, pp. 181–186.

CHAPTER 7

Conclusions and Future Perspectives

AI has already revolutionized our world. From Large Language Models (like ChatGPT) to object detection models (like YOLO) Artificial Intelligence (AI) is present more than ever in our lives. As the technology is so powerful, the possible applications become virtually endless. Some sectors are most interesting for applications, like autonomous driving or air traffic management might hugely benefit its employment. Nevertheless, there is the unmissable need of having reliable and safe systems. To this extent, this manuscript introduced novel techniques to solve this problems. The main contributions are summarized in the next section, while the following will explore future directions.

7.1 Main Contributions

Systolic array simulation and Fault Injection (FI) In chapter 3 we presented an innovative approach to FI, a method used to evaluate how well systems can withstand and recover from unexpected errors or malfunctions. We introduced a novel strategy designed specifically for Systolic Arrays (SAs). Our approach is tailored to address the unique challenges and vulnerabilities of matrix-multiplicator SAs. We implemented the strategy in flauers, a free and open source tool available for the research community. What makes flauers particularly notable is the simulation strategy: rather than using an hardware description language, the tool is able to map a normal matrix multiplication on Processing Element (PE) of the SA, making it particularly fast. Using flauers, we also explored the relationship between resilience and different dataflows. The results are presented in section 3.4

Fault injection optimization We delved into a critical, yet often overlooked, aspect of FI: the role of the *input workload* in shaping the effectiveness of FI campaigns. The core innovation lies in focusing on *challenging inputs* those that push Neural Networks (NNs) and the systems under evaluation to their limits. Instead of randomly or uniformly sampling inputs from the dataset, our approach identifies and prioritizes the most difficult cases, where faults are more likely to expose vulnerabilities or trigger catastrophic failures. This targeted selection not only streamlines the FI process but also amplifies its impact, ensuring that resources are concentrated where they matter most. This strategy serves as an upper-bound on the consequences of faults. On top of that, this approach is adaptable and scalable, making it suitable for a wide range of applications. By refining how we conduct FI, this work does not just optimize a process; it redefines how we evaluate and enhance the robustness of modern systems, setting a new standard for reliability engineering. Finally, this strategy can also be used to perform online testing, and assuring a reliable service.

The effect of approximation on NNs We examined the emerging paradigm of Approximate Computing (AxC) as a means to improve the resilience and efficiency of Deep Neural Networks (DNNs). DNNs have undeniably revolutionized fields ranging from image classification to autonomous systems, often surpassing human performance in complex tasks. However, their deployment in safety-critical applications, such as autonomous driving, demands not only high accuracy but also robustness against hardware imperfections, environmental noise, and computational constraints. This work posits that AxC, a technique that trades off computational precision for gains in efficiency, speed, and energy consumption, can be strategically leveraged to enhance the resilience of DNNs. By relaxing the strict requirements of exact computation, we can mitigate the impact of hardware-induced faults, reduce resource consumption, and even improve system adaptability in dynamic environments. The findings underscore the potential of approximation as a dual-purpose tool: it not only optimizes computational efficiency but also fortifies DNNs against the uncertainties inherent in real-world deployment, although the technique works only in some cases.

Fault Propagation Countermeasure We presented a novel and proactive approach to mitigate the impact of hardware-induced faults in NNs. As NNs, especially Convolutional Neural Networks (CNNs), become increasingly integrated into safety-critical embedded systems, their resilience to random hardware faults has emerged as a critical concern. Faults arising from latent defects, aging, or environmental perturbations can propagate through the network, leading to degraded performance or, worse, incorrect inference results, particularly problematic in applications such as autonomous driving, medical imaging, and industrial automation. The core contribution is the development of a monitor-based fault detection and prevention mechanism. The fundamental idea is to compute and analyze metrics from the output feature maps of early network layers during inference. By characterizing these metrics in real time, the system can detect anomalies indicative of fault propagation and halt inference, allowing for re-execution or corrective action before errors cascade through the network. This approach not only prevents faulty outputs but also enhances the overall reliability and trustworthiness of NN-based systems.

7.2 Perspectives and future directions

Achieving perfect reliability in AI is not an easy task. This manuscript focused on the hardware-side of things. Nevertheless, as the results showed throughout the manuscript, the biggest role is played by the set of weights available for the model. The weights are so important in fact that some results are only available with a specific sets and do not apply to the broad spectrum of NNs we have today. To this extent, studying the weights (possibly framing the problem as an explainability problem) is an interesting proposition, potentially allowing AI to be used in safety-critical applications. Furthermore, we showed some degree of redundancy in the NNs themselves, but it is unsure how this result is achieved. For this reason, we believe that another interesting point of view would be about minimizing the weights of a model, to make it simpler and thus less energy-hungry. Such an understanding could be then exploited to build intentional redundancy in the model itself.

Coming back on the hardware side of things, we should point out that our tool, flauers, is only able to simulate a subset of what the community offers. Some of the architectures presented in the past years are not available yet, namely NVDLA and MAERI. To this extent, it would be beneficial to implement a new strategy to simulate those architectures in the same spirit as flauers. On top of that, there are many lowering strategies that can be used to perform a convolution as a matrix multiplication. This direction was not explored, but has some potential: with a dynamic lowering strategy, it might be possible to assign a specific PE to a specific operation, enabling dynamic scheduling. This idea offers an opportunity to exclude (or limit) the use of faulty PE to compatible operations.

Nevertheless, using FI is expensive, even though the optimization technique we showed. To this

extent, following what said about explainability, there is the possibility to use analytical tools to test the reliability of NNs. In other words, the authors of this manuscript believe that FI is a valid tool that can be used to analyze the inner-workings of modern NN models.

A similar argument can be done using AxC as a means to inject noise in the model. To this extent, the work in this manuscript offers a peek into the effect of such noise. However, in the scope of this manuscript it was not used in a systematic way. Yet, the possibility of approximating other components than the multipliers is appealing. To this extent, another possible direction to take is regarding the reliability of the components of a NN. More specifically, quantized NN are ubiquitous and this opens the possibility to study each component they have: from the scaling and zero point of the tensors, to the approximate activation functions.

Finally, using the novel knowledge given by these possibilities, more countermeasures can be developed. The method showed in this manuscript has great potential, but it shows a relatively simple strategy. In order to build on this concept we can propose two main directions: (i) on one side, we could go one level lower explore the constant factors at the single PE level. (ii) on the other side, we could go one level higher and explore redundancy at the network level, exploring the opportunities for extending the neurons to detect adversarial noise. Finally, it would be interesting to model the counter-measures as part of the Regular Iterative Algorithms (RIAs) shown in subsection 3.2.3. This would give (i) on one side the possibility to automatically generate the Hardware Description Language (HDL) through the method seen in 12 and (ii) analytically manipulate the expressions. Another promising direction is in the employment of stochastic computing to perform multiplications in SA. Stochastic computing is so interesting because any error in a stochastic bitstream is an error on the Least Significant Bit (LSB), contributing only in a minor way in the multiplication. To our knowledge, this technique remains unexplored in SAs.

APPENDIX A

Synthèse en Français

Titre traduit: Étude du matériel pour une intelligence artificielle testable et rée

A.1 Résumé (Abstract)

C'est en 1943 que l'idée des réseaux de neurones (NNs) a été introduite pour la première fois dans la communauté académique en tant que paradigme computationnel. À cette époque, Warren McCulloch et Walter Pitts publiaient un article intitulé «The Logical Calculus of the Ideas Immanent in Nervous Activity, »dans lequel ils formalisaient le concept de réseau nerveux à travers une nouvelle théorie. Les auteurs y affirmaient que «pour la psychologie, quelle que soit sa définition, la spécification du réseau contribuerait à tout ce qui pourrait être accompli dans ce domaine «, soulignant ainsi l'espoir de reproduire une intelligence artificielle (IA) générale, comparable à l'intelligence humaine.

Ce n'est qu'en 2012, avec l'introduction de l'architecture AlexNet pour la classification d'images, que cette vision théorique a véritablement pris son essor. Treize ans plus tard, en 2025, nous avons parcouru un long chemin : nos systèmes sont désormais suffisamment sophistiqués pour passer aisément le test de Turing. L'avènement des grands modèles de langage (LLMs) a complètement transformé le paysage de l'IA.

Cependant, ces progrès s'accompagnent d'un coût élevé : non seulement la puissance requise pour faire fonctionner les systèmes actuels dépasse celle de tout système computationnel du passé, mais il a également été démontré que les modèles d'IA plus volumineux offrent de meilleures capacités, annonçant des besoins énergétiques encore plus importants.

Aujourd'hui, nous disposons des machines les plus puissantes de tous les temps. Pourtant, en observant les avancées récentes, il semble que nous allons bientôt saturer toute la puissance disponible. Une grande partie de ces progrès est due à la loi de Moore, qui stipule que le nombre de transistors dans un circuit intégré double tous les deux ans. Cependant, cette loi approche ses limites en raison des effets quantiques se manifestant à l'échelle actuelle (3 nm), et les défauts de fabrication deviennent de plus en plus problématiques.

De plus, les améliorations potentielles de l'IA semblent limitées par les contraintes computationnelles liées aux murs de la puissance et de la mémoire. Bien que ces problèmes soient connus depuis le début des années 2000, la communauté s'est tournée vers de nouvelles architectures (avec un succès modéré) pour répondre à l'opération la plus courante dans l'IA moderne : la multiplication matricielle. À cet égard, l'architecture la plus populaire est le tableau systolique (SA), introduit dans les années 1960, qui

offre une performance presque dix fois supérieure tout en consommant environ la moitié de l'énergie.

Le succès de cette technologie a poussé de nombreuses entreprises à proposer leur propre version. Cependant, avec des centres de données suffisamment vastes, un nouveau défi émerge : les corruptions silencieuses de données (SDC). Des entreprises comme Intel, Google et Meta ont rapporté que des erreurs quasi aléatoires affectent les calculs de leurs serveurs. Ce problème devient particulièrement crucial lorsque de tels systèmes sont introduits dans des environnements critiques pour la sécurité.

Dans ce contexte, ce manuscrit explore le problème de la fiabilité sous quatre angles :

1. Amélioration de la simulation : La simulation au niveau Register Transfer Level (RTL) est coûteuse et prendrait beaucoup de temps pour simuler un système complexe composé de SA. Pour cette raison, nous proposons un cadre d'injection de fautes (FI) pour étudier la fiabilité de l'IA lorsque le SA est défaillant et que des erreurs sont introduites dans les calculs.

2. Optimisation de la FI : Comme la FI est un processus coûteux, nous proposons une nouvelle technique pour réduire la charge de travail de la FI par sélection de données : en ne prenant que des exemples difficiles, il est possible d'obtenir une estimation pessimiste de la résilience du réseau.

3. Étude de l'effet du bruit dans les NNs : Nous analysons l'impact du bruit sur les NNs par l'utilisation de multiplieurs approximatifs dans le réseau.

4. Développement d'une suite de benchmarks : Pour comparer équitablement les résultats et les contre-mesures.

A.2 Contexte et enjeux

Coût de l'IA

- Les modèles d'IA modernes (comme les LLMs) nécessitent des ressources computationnelles et énergétiques colossales, avec des coûts croissants exponentiellement (ex. : 2,4x/an pour l'entraînement des modèles frontaliers).
- La loi de Moore approche ses limites physiques (effets quantiques, courants de fuite), rendant les architectures traditionnelles (CPU/GPU) insuffisantes.

Fiabilité des NNs

- Les défauts matériels (transitoires ou permanents) peuvent corrompre les calculs, entraînant des Silent Data Corruptions (SDC).
- Les systèmes critiques (ex. : conduite autonome) exigent une résilience accrue face aux erreurs.

Tableaux systoliques (SA)

- Architecture optimisée pour les multiplications matricielles, réduisant la consommation énergétique et améliorant les performances.
- Trois types principaux : Output Stationary, Weight Stationary, et No Local Reuse.

A.3 Contributions clés

Cette section présente la liste des principales contributions du manuscrit.

Cadre d'injection de fautes (FI)

- Développement de flauers (anciennement SAFFIRA), un outil open-source pour simuler les fautes dans les SA.
- Approche hybride : combine la précision des simulations matérielles (RTL) avec l'efficacité des modèles logiciels.
- Modélisation des fautes via des Uniform Recurrent Equations (URE) et des Regular Iterative Algorithms (RIA) pour projeter les erreurs dans l'espace physique des SA.

Optimisation de la FI

- Réduction du coût computationnel en sélectionnant des échantillons de données ”challenging” pour évaluer la résilience des NNs.
- Utilisation de métriques pour classer les fautes (ex. : masquées, critiques).

Étude des multiplieurs approximatifs

- Analyse de l'impact des approximations matérielles (ex. : multiplieurs low-power) sur la robustesse des NNs.
- Résultats : Certaines approximations améliorent la résilience, tandis que d'autres dégradent les performances.

Contre-mesures

- Développement de moniteurs pour prévenir la propagation des fautes.
- Évaluation des coûts computationnels et énergétiques des solutions proposées.

A.4 Perspectives futures

- Intégration des SA dans des systèmes embarqués pour des applications temps réel.
- Amélioration des outils de simulation pour inclure des modèles énergétiques et des latences réalistes.
- Exploration de nouvelles architectures (ex. : in-memory computing) pour dépasser les limites des SA actuels.

A.5 Publications associées

Le manuscrit s'appuie sur 12 publications (conférences et journaux), couvrant des sujets comme :

- L'évaluation de la résilience des accélérateurs DNN ([3], [7]).
- L'impact des multiplieurs approximatifs ([6]).
- Les frameworks pour la FI dans les SA ([1], [8]).

Voici un résumé détaillé et académique de chaque chapitre du manuscrit, rédigé en français.

A.6 Résumé par Chapitre

A.6.1 Chapitre 1 : Introduction

Ce chapitre plonge dans les fondements conceptuels et historiques de l'intelligence artificielle, en commençant par une analyse étymologique des termes « artificiel » et « intelligence ». Il rappelle que l'intelligence, longtemps considérée comme une capacité purement humaine, a été progressivement modélisée par des machines, d'abord à travers des automates mécaniques, puis via des algorithmes inspirés du fonctionnement du cerveau. Les perceptrons de Rosenblatt dans les années 1950, puis les réseaux de neurones multicouches dans les années 1980, ont marqué des étapes clés dans cette quête de reproduction de l'intelligence. Cependant, l'essor fulgurant de l'IA moderne, illustré par des modèles comme AlexNet en 2012 ou les LLMs aujourd'hui, s'accompagne de coûts prohibitifs. Les données présentées montrent que le coût énergétique et financier de l'entraînement des modèles a explosé, avec des besoins en calcul qui doublent presque chaque année. Cette croissance exponentielle n'est pas durable, d'autant plus que les limites physiques des semi-conducteurs se font de plus en plus sentir. La loi de Moore, qui a permis pendant des décennies une augmentation constante de la puissance de calcul, atteint aujourd'hui ses limites. Les transistors, désormais de l'ordre de 3 nm, sont sujets à des effets quantiques et à des courants de fuite qui dégradent leurs performances. Parallèlement, le mur de la mémoire – le décalage entre la vitesse des processeurs et celle des mémoires – devient un goulot d'étranglement majeur. Face à ces défis, les architectures matérielles évoluent. Les Systolic Arrays, par exemple, se distinguent par leur capacité à effectuer des multiplications matricielles de manière extrêmement efficace, avec une complexité temporelle linéaire et une faible consommation d'énergie. Ces caractéristiques en font des candidats idéaux pour les applications embarquées ou les centres de données. Cependant, leur adoption dans des environnements critiques nécessite une évaluation rigoureuse de leur fiabilité. Les fautes matérielles, qu'elles proviennent de défauts de fabrication, de vieillissement, ou de radiations, peuvent en effet compromettre la sécurité des systèmes. Ce chapitre souligne ainsi l'importance de développer des méthodologies pour tester et renforcer la résilience de ces architectures, un thème central qui sera approfondi dans les chapitres suivants.

Contributions clés :

- Problématique de la fiabilité des systèmes d'IA dans des contextes critiques (ex. : véhicules autonomes, aviation).
- Introduction des concepts de résilience, robustesse, et sûreté de fonctionnement, avec une définition originale de la résilience comme « capacité à résister au bruit sans rupture ».
- Présentation des fautes matérielles (défauts de fabrication, vieillissement, radiations) et de leur propagation dans les systèmes (faute erreur défaillance).

A.6.2 Chapitre 2 : Contexte Théorique

Ce chapitre pose les bases techniques nécessaires à la compréhension des enjeux abordés dans le manuscrit. Il est structuré en trois parties distinctes, chacune explorant un aspect fondamental : les réseaux de neurones, les fautes et la fiabilité, et enfin les Systolic Arrays. La première section est consacrée aux réseaux de neurones, depuis les perceptrons jusqu'aux architectures convolutives modernes. Ce chapitre explique le fonctionnement d'un neurone artificiel, modélisé comme une combinaison linéaire de ses entrées suivie d'une fonction d'activation non linéaire. Cette non-linéarité est cruciale, car sans elle, un réseau de neurones ne serait qu'un simple modèle linéaire, incapable de capturer des relations complexes. Les algorithmes d'apprentissage, comme la descente de gradient stochastique, sont également détaillés, avec une emphase sur leur rôle dans l'optimisation des paramètres du réseau. Les réseaux convolutifs (CNNs) sont ensuite présentés comme une avancée majeure, permettant de traiter des données spatiales

comme les images avec une efficacité inégalée. Leur capacité à réduire le nombre de paramètres grâce au partage des poids et à capturer des motifs invariants par translation en fait des outils puissants pour la vision par ordinateur. Des architectures emblématiques comme LeNet-5, AlexNet, ou ResNet sont brièvement décrites, illustrant l'évolution des CNN vers des modèles toujours plus profonds et plus performants. La deuxième section aborde la question des fautes et de la fiabilité. Les fautes matérielles, qu'elles soient transitoires ou permanentes, sont classées et leurs impacts sur les systèmes sont analysés. Les méthodes d'injection de fautes (FI), essentielles pour évaluer la résilience des systèmes, sont passées en revue. On distingue les approches basées sur l'irradiation, l'émulation matérielle, et la simulation logicielle, chacune ayant ses avantages et ses limites. La simulation, bien que moins coûteuse, peut manquer de réalisme, tandis que l'irradiation ou l'émulation offrent une précision accrue mais à un coût prohibitif. Enfin, la troisième section introduit les Systolic Arrays, des architectures matérielles spécialisées dans les calculs matriciels. Leur structure régulière, composée de Processing Elements (PEs) interconnectés, permet une exécution parallèle et efficace des opérations de convolution, centrales dans les DNNs. Le chapitre explique comment ces architectures peuvent être modélisées à l'aide d'Uniform Recurrent Equations (URE) et de Regular Iterative Algorithms (RIA), un cadre mathématique qui sera exploité dans les chapitres suivants pour simuler et analyser leur comportement en présence de fautes. Ce chapitre, bien que technique, reste accessible grâce à des explications claires et des exemples concrets. Il prépare le terrain pour les contributions originales du manuscrit, en particulier le développement du framework flauers pour l'injection de fautes dans les SA.

Ce chapitre fournit les fondements techniques nécessaires à la compréhension du manuscrit, divisé en trois sections :

1. Réseaux de Neurones (NNs) :
    - Perceptrons et Feed-Forward NNs : Modèle mathématique d'un neurone (équation 2.1), architecture en couches, et rôle des fonctions d'activation non linéaires (ex. : sigmoïde, ReLU).
    - Algorithme d'apprentissage : Descente de gradient stochastique (SGD), fonction de coût, et problème de surapprentissage.
    - Réseaux Convolutifs (CNNs) : Opération de convolution (équation 2.6), avantages (réduction des paramètres, invariance par translation), et architectures emblématiques (LeNet-5, AlexNet, VGG-16, ResNet-50).
    - Jeux de données : Importance de la division en ensembles d'entraînement/test/validation pour évaluer la généralisation.
2. Fautes et Fiabilité :
    - Modèles de fautes : Distinction entre fautes transitoires (ex. : bit-flips dus aux radiations) et permanentes (ex. : courts-circuits).
    - Injection de Fautes (FI) : Méthodologies existantes (campagnes d'irradiation, émulation matérielle, simulation logicielle) et leur classification selon [Ahmadilivani et al., 2024].
    - Évaluation Fonctionnelle par FI : Comparaison entre un système Golden (sans faute) et un système Faulty (avec faute), avec métriques de classification (masqué, bon, acceptable, avertissement, critique).
3. Systolic Arrays (SA) :
    - Historique et Principe : Architecture introduite dans les années 1970, composée de Processing Elements (PEs) interconnectés pour des calculs locaux (ex. : multiplication matricielle).

- Avantages : Efficacité énergétique, faible latence, et adaptabilité aux opérations de convolution (via Lowering and Lifting, LoLif).
- Types de SA : Output-Stationary (accumulation dans le PE), Weight-Stationary (poids fixes par PE), et No Local Reuse (flux séparés pour activations/poids/accumulation).

A.6.3 Chapitre 3 : Injection de Fautes dans les Systolic Arrays

Ce chapitre présente flauers (Faulty Logic Assessment Unit for Enhancing Resilience of Systolic Systems), un framework open-source dédié à l'injection de fautes dans les SA. L'objectif est de combler le fossé entre les simulations hardware-aware (précises mais coûteuses) et hardware-agnostic (rapides mais imprécises).

Motivations :

- Limites des outils existants (ex. : PyTorchFI ne modélise pas les fautes matérielles spécifiques aux SA).
- Besoin d'un outil rapide, modulaire, et compatible avec les bibliothèques modernes (PyTorch, TensorFlow).

Ce chapitre est le cœur de la contribution technique du manuscrit. Il présente flauers (Faulty Logic Assessment Unit for Enhancing Resilience of Systolic Systems), un framework innovant dédié à l'injection de fautes dans les Systolic Arrays. L'objectif est de proposer un outil qui combine la précision des simulations hardware-aware avec la rapidité des approches hardware-agnostic, un compromis essentiel pour étudier des systèmes complexes comme les SA. Le chapitre commence par expliquer les motivations derrière le développement de flauers. Les outils existants, comme PyTorchFI ou TensorFI, bien qu'utiles, ne sont pas adaptés aux spécificités des SA. Ils se concentrent soit sur des fautes logicielles (comme des perturbations des poids des réseaux), soit sur des architectures plus traditionnelles comme les GPU ou les CPU. Or, les SA, avec leur structure unique et leur mode de fonctionnement, nécessitent une approche dédiée. flauers se positionne donc comme un outil capable de simuler des fautes matérielles de manière réaliste, tout en restant suffisamment léger pour permettre des campagnes d'injection à grande échelle. Le cadre mathématique sous-jacent à flauers est ensuite détaillé. Les SA sont modélisées à l'aide d'Uniform Recurrent Equations (URE) et de Regular Iterative Algorithms (RIA), qui décrivent les dépendances entre les calculs effectués par les différents Processing Elements (PEs). Cette formalisation permet de projeter l'espace d'itération (où chaque point représente un calcul) vers l'espace physique (où chaque point correspond à un PE à un instant donné). La matrice de projection PPP joue ici un rôle clé, car elle détermine la topologie du SA et la manière dont les données circulent entre les PEs. Le mécanisme d'injection de fautes est au cœur de flauers. L'utilisateur spécifie plusieurs paramètres : la ligne cible (par exemple, les poids, les activations, ou les accumulations), le type de faute, la position physique dans le SA, et l'intervalle temporel pendant lequel la faute doit être injectée. L'algorithme identifie ensuite les points de l'espace d'itération affectés par la faute, en tenant compte des dépendances spatiales et temporelles. Cette approche permet de simuler précisément la propagation des erreurs dans le SA, un aspect crucial pour évaluer la résilience du système. Ce chapitre illustre le fonctionnement de flauers à travers des exemples concrets, comme la génération de SA de type Output-Stationary ou Weight-Stationary. Ces exemples montrent comment différentes matrices de projection PPP conduisent à des topologies distinctes, chacune avec ses propres caractéristiques en termes de flux de données et de performance. Des extraits de code en Python/Migen sont également présentés, démontrant comment flauers peut générer automatiquement une description matérielle (HDL) des SA, une fonctionnalité utile pour des implémentations futures sur FPGA ou ASIC. Enfin, les limites de l'approche sont discutées avec honnêteté. flauers, dans sa version actuelle, ne modélise qu'un sous-ensemble des fautes possibles (principalement des bit-flips), et ne prend pas en compte des aspects comme la consommation énergétique ou les délais de propagation. De plus, l'intégration avec des frameworks comme PyTorch ou TensorFlow reste un défi en raison des

optimisations agressives de ces bibliothèques, qui rendent difficile l'insertion de modules personnalisés comme flauers. Malgré ces limites, le framework représente une avancée significative, offrant un outil open-source et modulaire pour l'étude de la résilience des SA.

Méthodologie

1. Cadre Mathématique :
    - Formalisation des SA via des Uniform Recurrent Equations (URE) et Regular Iterative Algorithms (RIA), permettant de décrire les dépendances entre calculs (équation 3.2).
    - Projection de l'espace d'itération S_i (coordonnées (i, j, k)) vers l'espace physique S_p (coordonnées (x, y, t)) via une matrice de projection P (équation 3.5).
2. Mécanisme d'Injection :
    - Paramètres d'injection : Ligne cible (a, b, ou c), type de faute, position physique (x, y), et intervalle temporel $(t_{\text{start}}, t_{\text{stop}})$.
    - Propagation des fautes : Algorithme 1 pour identifier les points d'injection dans S_i à partir de S_p, en tenant compte des dépendances spatiales et temporelles (vecteurs δx_i et δt_i).
    - Implémentation : Modification des valeurs des registres a, b, ou c pendant la simulation (Algorithme 2).
3. Exemples et Performances :
    - Matrices de Projection : Comparaison des topologies Output-Stationary (figure 3.6), Weight-Stationary (figure 3.7), et No Local Reuse.
    - Génération de HDL : Code Python/Migen pour générer automatiquement le description matérielle (VHDL/Verilog) des SA à partir de P.
    - Limites : Fautes modélisables limitées (ex. : pas de fautes de retard ou de diaphonie), et absence de métriques énergétiques/délais dans la version actuelle.

Intégration Logicielle

- Défis pour coupler flauers avec PyTorch/TensorFlow (optimisations des bibliothèques, génération dynamique des noyaux CUDA).
- Solutions proposées : utilisation de couches personnalisées ou d'outils académiques compatibles ONNX.

A.6.4 Chapitre 4 : Optimisation de l'Injection de Fautes

L'injection de fautes, bien qu'indispensable pour évaluer la résilience des systèmes, est une tâche extrêmement coûteuse en termes de ressources computationnelles. Ce chapitre aborde la question de l'optimisation de ces campagnes, en proposant des méthodes pour réduire leur complexité tout en conservant une estimation fiable de la résilience des réseaux de neurones. Il commence par rappeler que les campagnes d'injection de fautes exhaustives sont souvent irréalisables en pratique, en raison du nombre astronomique de combinaisons possibles entre fautes, entrées, et paramètres du réseau. Une approche statistique est donc nécessaire, mais elle doit être soigneusement conçue pour éviter des biais ou des approximations trop grossières. La sélection de métriques pertinentes est un premier levier d'optimisation. Plutôt que d'injecter des fautes de manière aléatoire et uniforme, on propose de cibler les échantillons les plus « challenging », c'est-à-dire ceux pour lesquels le réseau est le plus sensible aux perturbations. Des métriques

comme la confiance (confidence) ou la marge (margin) sont utilisées pour identifier ces échantillons critiques. Par exemple, un échantillon pour lequel la sortie du réseau est proche du seuil de décision entre deux classes sera plus susceptible de basculer en cas de faute, et devra donc être prioritaire dans les campagnes de test. Des expérimentations matérielles sont ensuite présentées pour valider cette approche. Deux types d'expérimentations sont distingués : les tests hardware-independent, qui évaluent la résilience du réseau indépendamment de l'architecture matérielle sous-jacente, et les tests hardware-aware, qui prennent en compte les spécificités des SA. Les résultats montrent qu'en se concentrant sur les échantillons les plus sensibles, il est possible de réduire significativement le nombre d'injections nécessaires, tout en conservant une estimation pessimiste mais réaliste de la résilience du système. Le chapitre souligne également l'importance de benchmarks standardisés pour comparer les différentes approches. Sans des protocoles communs, il est en effet difficile d'évaluer objectivement l'efficacité des contre-mesures proposées. Ce chapitre ouvre ainsi la voie à des méthodologies plus efficaces pour l'évaluation de la résilience, un enjeu crucial pour le déploiement sécurisé des systèmes d'IA dans des environnements critiques.

A.6.5 Chapitre 5 : Approximation et Résilience

Ce chapitre explore une approche contre-intuitive mais prometteuse pour améliorer la résilience des réseaux de neurones : l'utilisation de multiplieurs approximatifs. Ces composants, qui sacrifient une partie de la précision des calculs pour gagner en efficacité énergétique ou en vitesse, pourraient paradoxalement rendre les systèmes plus robustes face aux fautes matérielles. Ce chapitre commence par rappeler le contexte du Approximate Computing, un domaine qui cherche à exploiter le fait que de nombreuses applications (comme le traitement d'images ou de sons) tolèrent un certain niveau d'imprécision sans perte significative de qualité. Dans les réseaux de neurones, où les calculs sont souvent redondants et où une certaine marge d'erreur est acceptable, cette approche peut être particulièrement pertinente. Les multiplieurs approximatifs, comme EvoApprox8 ou AxMul-8, introduisent des erreurs contrôlées dans les opérations de multiplication, ce qui peut réduire la consommation d'énergie tout en limitant l'impact sur les performances globales du réseau. Une étude de cas est ensuite présentée, dans laquelle des multiplieurs approximatifs sont intégrés dans un Systolic Array exécutant un réseau convolutif comme LeNet-5. Les résultats montrent que certains multiplieurs, bien qu'introduisant des erreurs, peuvent en fait améliorer la résilience du réseau face aux fautes matérielles. Cela s'explique par le fait que les erreurs introduites par les multiplieurs approximatifs peuvent « brouiller » les effets des fautes, rendant le système moins sensible aux perturbations extérieures. On analyse également la corrélation entre les bits injectés et la résilience. Il apparaît que les bits de poids faible ont un impact moindre sur la robustesse du réseau, une observation qui pourrait guider la conception de multiplieurs approximatifs optimisés pour la résilience. Enfin, des multiplieurs résilients sont proposés, conçus spécifiquement pour minimiser la propagation des erreurs tout en conservant les avantages énergétiques de l'approximation. Ce chapitre met en lumière un paradoxe fascinant : en acceptant une certaine imprécision dans les calculs, on peut en fait rendre les systèmes plus fiables. Cette approche, bien que contre-intuitive, ouvre des perspectives intéressantes pour la conception de systèmes d'IA à la fois efficaces et résilients, un enjeu crucial pour les applications embarquées ou critiques. eur conception doit être adaptée au réseau cible.

A.6.6 Chapitre 6 : Prévention de la Propagation des Fautes via des Moniteurs

Alors que les chapitres précédents se concentraient sur l'évaluation et l'atténuation des effets des fautes, ce chapitre propose une approche proactive pour prévenir leur propagation au sein des Systolic Arrays. L'idée est d'intégrer des moniteurs matériels capables de détecter et de corriger les erreurs en temps réel, avant qu'elles n'affectent les sorties du réseau de neurones. Le chapitre commence par rappeler que les fautes, une fois introduites dans un système, peuvent se propager et amplifier leurs effets, conduisant éventuellement à des Silent Data Corruptions (SDC) difficiles à détecter. Dans les environnements critiques, comme les systèmes autonomes ou médicaux, de telles corruptions peuvent avoir des conséquences

catastrophiques. Il est donc essentiel de mettre en place des mécanismes de détection précoce. La solution proposée repose sur l'intégration de moniteurs locaux au sein des SA. Ces moniteurs surveillent en continu les calculs effectués par les Processing Elements (PEs), en vérifiant par exemple que les valeurs des accumulations restent dans des plages attendues. Des seuils dynamiques sont utilisés pour adapter la sensibilité des moniteurs en fonction des couches du réseau. Par exemple, les couches convolutives, souvent plus robustes, peuvent tolérer des seuils de détection plus larges que les couches pleinement connectées, plus sensibles aux perturbations. Les résultats expérimentaux montrent que cette approche permet de réduire significativement le nombre de SDC, avec un surcoût computationnel limité. Les moniteurs, bien qu'ajoutant une certaine complexité au système, se révèlent efficaces pour intercepté les fautes avant qu'elles ne se propagent. Cette méthode est validée sur des architectures comme LeNet-5 et AlexNet, démontrant son applicabilité à différents types de réseaux. Ce chapitre souligne l'importance d'une approche préventive dans la gestion des fautes, complémentaire aux méthodes correctives ou tolérantes aux fautes présentées dans les chapitres précédents. En combinant détection précoce et contre-mesures adaptatives, il devient possible de concevoir des systèmes d'IA non seulement performants, mais aussi fiables et sécurisés, un impératif pour leur adoption dans des domaines critiques.

A.6.7 Chapitre 7 : Conclusions et Perspectives

Le dernier chapitre du manuscrit offre une synthèse des contributions présentées et ouvre des perspectives pour les travaux futurs. On rappelle que l'objectif principal de ce travail était de développer des méthodologies pour évaluer et améliorer la résilience des systèmes d'IA, en particulier ceux basés sur des Systolic Arrays, dans des environnements où la fiabilité est cruciale. Les contributions majeures sont résumées comme suit :

Le développement de flauers, un framework open-source pour l'injection de fautes dans les SA, combinant précision et efficacité. L'optimisation des campagnes d'injection de fautes via des méthodes de sélection de données, réduisant les coûts computationnels tout en conservant une estimation fiable de la résilience. L'étude de l'impact des multiplieurs approximatifs sur la résilience, montrant que l'approximation peut, contre toute attente, améliorer la robustesse des réseaux. La création de bancs d'essai pour comparer objectivement les différentes contre-mesures proposées.

Dans ce chapitre, on souligne que ces contributions s'inscrivent dans une démarche plus large visant à rendre les systèmes d'IA dignes de confiance (trustworthy), un enjeu majeur pour leur adoption dans des applications critiques comme les véhicules autonomes, la médecine, ou l'aérospatiale. Les perspectives futures sont nombreuses et ambitieuses. Parmi elles, on peut citer :

L'intégration matérielle de flauers avec des FPGA ou des ASIC pour des évaluations en temps réel, ce qui nécessitera de relever des défis liés à la compatibilité avec les architectures existantes. La modélisation de fautes plus complexes, comme les fautes dépendantes du temps (delay faults) ou les effets de diaphonie, pour une évaluation encore plus réaliste de la résilience. Le développement de mécanismes de résilience adaptative, où les systèmes pourraient s'auto-réparer ou s'auto-configurer en fonction des fautes détectées. La standardisation des protocoles de test, en collaboration avec l'industrie, pour établir des normes communes d'évaluation de la fiabilité des systèmes d'IA.

Contributions Majeures :

1. Framework flauers : Outil open-source pour l'injection de fautes dans les SA, combinant rapidité et précision.
2. Optimisation des FI : Méthodes de sélection de données pour réduire les coûts de simulation.
3. Étude des Multiplieurs Approximatifs : Analyse de leur impact sur la résilience, avec propositions de designs optimisés.
4. Bancs d'Essai : Développement de benchmarks pour évaluer et comparer les contre-mesures.

Perspectives Futures :

- Intégration Matérielle : Couplage de flauers avec des FPGA/ASIC pour des évaluations en temps réel.
- Fautes Complexes : Modélisation de fautes dépendantes du temps (ex. : delay faults) ou contextuelles (ex. : diaphonie).
- Résilience Adaptative : Conception de NNs auto-réparateurs via des mécanismes de self-healing.
- Standardisation : Collaboration avec l'industrie pour définir des protocoles de test unifiés.

A.7 Conclusion

Ce travail propose une méthodologie complète pour évaluer et améliorer la fiabilité des systèmes d'IA, en combinant des outils de simulation innovants, des analyses théoriques, et des solutions matérielles. Les résultats ouvrent la voie à des applications critiques plus sûres et plus efficaces.

Acknowledgments

Three years are very long. It's about 1000 days. And if you consider the title of that nice song "what a difference a day made", imagine how much of a difference a thousand days can make. And they did make a difference. Many differences indeed.

When I started my PhD I had no clue what I was gonna do. I left Italy, almost rushing, and I ended up in Lyon, in France. I did not know the language. I did not know the people, but I found a welcoming environment. And nice people.

Now, on the off chance of making this section a whole chapter (because yeah, I am definitely capable of that) I will try to concentrate and properly thank the people that were by my side during these long three years.

I want to start by the thanking my Master thesis supervisors: Ernesto Sanchez and Annachiara Ruospo. Without them, I would have never met my PhD supervisor: Alberto Bosio. When Alberto and I met, it was fully blown pandemic. In fact, we did not really *meet*, rather we joined a virtual meeting and at that time he asked me a bunch of questions. I am not sure what he saw in me back then, but he put his trust on me and that is how I got to Lyon. There I met Bastien Deveautour, one of the most kind people I have ever met. And my second supervisor. All of them deserve a special thank you for believing in me, even when I did not.

Then, surely, PhD would be practically impossible without financial support. For this reason I want to thank the project ReTrusting, which made my experiences possible at a practical level. And more broadly, the people who made INL what it is. Among which, I cannot forget to mention Ian O'Connor – who always organized weekly meetings that nobody wanted to attend (but he needed to keep up with our progress) – and other important people for this lab: Sylvie Goncalves and Patricia Dufaut, Cedric Marchand, Bruno Masenelli, Segolene Callard, etc...

I mentioned, a couple of paragraphs up, that I found a welcoming environment when I arrived in Lyon. And this is the case because of the people that were in the office. Now, everyone has a different path to go through and this meant that, even though some people were there when I arrived, they inevitably left before I finished. In the same way, some people are still here, while I am already done. Nevertheless, I will not be able to forget them for the rest of my life. Making a complete list would be practically impossible, so I apologize for those of you who are not in the following list, but be sure that all those who crossed my path will remain in my heart, in a way or another. So, I want to especially thank my office-mates: Raphael, Mohab, Sergio, Ali, Paul, PA, Mauricio, Clèmant, Alban, Mateus, Basile, Antoine, Miqueas, Rosario, Eya, Manar, Leonardo, Antonio and Micka. I surely forgot someone, but those who met me, know I am a pretty forgetful person eheheh. These people are those who I met basically every day for the past three years. We've seen things together. And I learnt a lot (but truly a lot) of lessons thanks to them. Every comment, every laugh, every cough and "bless you" were part of this journey and in one measure or another, they made the difference.

I will dedicate this paragraph to some special (very special) people. I am talking of Anjali, Dahlia, Fryni, Salsabil and Sara. We basically lived together. At the beginning I did not know any of them. One day we went to celebrate Salsabil's birthday. She just wanted to be wished. And there I met Dahlia, who laughed because we live in Lyon. Talking about taxes, I found out that Lyon's postal code is 69. And she just couldn't hold the laugh. We became friends. That night, Sara was sitting next to her. She was embarrassed, but amused nonetheless. The way I met Sara was funny, actually. It was the first of November 2022. I came for the very first time to Lyon to sign the PhD contract. I met the guys in the lab and, while leaving, I wanted to shake everybody's hand. She was the last one, and I was already thinking at "what to do" when it was her turn. You see, I had never met a hijabi girl before. When I outstretched my hand, she put hers on her chest and bowed. I bowed as well. We became friends. Fryni has always been my go-to when I am in *Michael Scott* mood. The day we met was a Monday. Back then both of us were going to French class, to learn the language. And we did not really know each other, but

we were going back to the office together. While we were on the stairs, I made a comment (which shall remain unwritten, but those who know, know) about some potential courtship she might have faced in the future from a mutual acquaintance. When I was done, she looked at me shocked. Then busted in laughs, saying how" my comment was very inopportune, but I was lucky it was her I commented to. We became friends. About Salsabil I remember something different. I do not remember how "we became friends", but I remember what happened at the beginning of Summer 2023. Me, her and Dahlia were hanging out a lot together in that period. That was the last day before summer break. We went all together to the station, to accompany Dahlia, Sara and Chirine to Gare de Part-Dieu. They were going home in Lebanon. And I was gonna go to Sicily. And Salsabil was gonna go to Tunisia. But differently than us, she was the only one who might not have come back. And no tear was spared. They all crowded her usually smiley visage, being sad for the future, uncertain whether she would see us again or not. At that point we had shared many adventures already and we were friends. It was heartbreaking to see her like that... But soon afterwards she graduated and she forgot all the sadness. Did I mention she actually came back? Yeah. She did. The way I met Anjali was actually very funny. I was in a period during which I wanted to meet new people. So that night I was going back home after work. I was taking the elevator and someone passed by. I remember seeing her being too shy to take the elevator with me and for this reason she took the stairs. Little did I know she was my immediate neighbor! As soon as I saw her opening her door I immediately shouted: "Oh wow! So you are my neighbor!" And asked her to exchange our numbers. I texted her only months later. I invited her and another neighbor to take a tea together. He did not show up. We had *mango passion* tea. I'm sure she did not love it, but drank it anyways. The next day I cooked pasta for us. While in the middle of the night, I heard Italian songs being played outside, in the corridor and I convinced her to go with me and knock at the door, and we spent a funny night with a bunch of Italians. The following days we were basically always together. We enjoyed each other company, and we were complaining mutually about our lives and problems. We became friends. I don't know how, but then I made Anjali and Dahlia meet. They liked each other and so we became the *tricycle*[19].

Jokes aside, both Anjali and Dahlia were there for me when I most needed it. And I will never forget that dark period of my life, in which the light was solely dictated by the two of them. For this reason, Thank You. And also we exchanged the services. For example, there was a time that Dahlia needed a little push, just a little nudge, to confront a life situation. That's when we really became friends. Similarly, Anjali was going through a rough patch, due to a certain situationship, so to speak. And we started playing "It takes 2". That's when we really became friends. The rest is history. Once again, Thank you girls.

E poi la signora Anna. Questo è un ringraziamento che avrei dovuto fare parecchi anni fa, già e quindi arriva molto in ritardo. Ma sono sicuro che lei leggendo starà pensando una cosa come "meglio tardi che mai". Grazie a lei ho imparato ad essere estroverso senza vergognarmene. E si, è successo parecchi anni fa, ma mi è stato utile per tutta la vita. E oggi so dove l'ho imparato. E ancora, Erika. Ci conosciamo da tanti anni ormai. Avevamo una decina di anni ciascuno. E ancora oggi ci raccontiamo i nostri gossip, con la stessa intensità di 15 anni fa. Anche se siamo due persone molto diverse, sappiamo comprenderci e capirci. Nonostante la distanza. Nonostante il tempo. Grazie di essere mia amica.

Ovviamente, per ultimi, ma non per importanza, voglio ringraziare la mia famiglia. I miei genitori, i miei nonni, i miei patrozzi[20] di laurea. Sono passati tanti anni da quando sono andato via da Biancavilla. Tante cose sono successe in questi anni. E loro lo sanno. Tutte le avventure e disavventure. Le gioie e i dolori. E finalmente questo lungo e laborioso percorso di studi è terminato. Grazie d'avermi permesso di partire, grazie per i soldi che mi avete mandato durante questi anni (indispensabili), grazie per aver avuto fiducia in me, grazie per essermi stato vicino quando ne avevo bisogno e grazie anche d'essermi stato lontani quando era necessario. Sarebbe stato tutto molto diverso senza il vostro supporto. Grazie per avermi insegnato il potere dell'Amore. Come scrivevo all'inizio di questi riconoscimenti, c'è una canzone che dice "quanta differenza fa un singolo giorno" per dire che in ventiquattro ore può cambiare tutta la

[19] Finding a good name was not easy. They kept insisting that it would be weird to have a name for "the three of us". But in the end we settled on this. I guess. Well, I learned today that there is a funny reference in "How I met Your Mother", mentioning a certain tricycle, but I digress.

[20] Avevo scritto "padrini" all'inizio, ma non mi piaceva. Del resto, non vi ho mai chiamati così. Patrozzi suona più naturale.

tua vita. Ventiquattro misere ore. Immaginate quanto possano fare 8 anni. E si, hanno fatto parecchio. Sono cresciuto tanto. E poi E anche se sono andato via da tanto tempo, ogni volta che ci rincontriamo, sembra che non sia mai partito.

You might think, reading, where is the *science* in these acknowledgments? And my answer is simple: it's not there. Science is made by people and as such, we need to remember the feelings, the love, the emotions and also the hate and despair lived throughout. Science is made by people. And people are... Just people.

27 Feb 2026 - Salvo

List of acronyms

AI Artificial Intelligence 129

ML Machine Learning 53

FI Fault Injection 129

SDC Silent Data Corruption 122

CC Clock Cycle 46

LSB Least Significant Bit 131

MSB Most Significant Bit 107

RTL Register Transfer Level 40

SA Systolic Array 129

PE Processing Element 129

MLP MultiLayer Perceptrons 100

FLOPS FLoating-points Operations Per Second 122

LLM Large Language Models 4

DSA Domain Specific Architectures 41

GPU Graphical Processing Unit 40

CPU Central Processing Unit 40

TPU Tensor Processing Unit 30

NVDLA NVIDIA's Deep Learning Accelerator 10

NN Neural Network 129

CNN Convolutional Neural Network 130

DNN Deep Neural Network 130

SGD Stochastic Gradinet Descent 22

matmul Matrix Multiplication 41

FPGA Field Programmable Gate Array 30

BLAS Basic Linear Algebra Subprograms 41

HPC High-Performance Computing 41

MAC Multiply And Accumlate 122

URE Uniform Recurrent Equation 41

RIA Regular Iterative Algorithm 131

LoLif Lowering and Lifting 41

HDL Hardware Description Language 131

API Application Programming Interface 53

OS Output Stationary 61

RS Row Stationary 61

NLR No Local Reuse 61

WS Weight Stationary 34

NaN Not a Number 64

AxC Approximate Computing 130

valset Validation Set 76

MAE Mean Absolute Error 102

WCE Worst Case Error 102

SET Single Event Transient 74

SFI Statistical Fault Injection 75