Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (34^{th}cycle)

Reliability and Security Assessment of Modern Embedded Devices

By

Annachiara Ruospo

******

Supervisor:
Prof. Ernesto Sanchez

Doctoral Examination Committee:
Prof. Letícia Maria Bolzani Poehls, IDS RWTH Aachen University, Germany
Prof. Alberto Bosio, École Centrale De Lyon, INSA Lyon, CNRS, INL, France
Prof. Giorgio Di Natale , *Referee*, Université Grenoble-Alpes, TIMA, CNRS, France
Prof. Luca Sterpone, Politecnico di Torino, DAUIN, Italy
Prof. Haralampos Stratigopoulos, *Referee*, Sorbonne Université, CNRS, LIP6, France

Politecnico di Torino
2022

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Annachiara Ruospo
2022

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

Dedico tutto questo alla mia famiglia, da sempre e per sempre al mio fianco.

Acknowledgements

And I would like to acknowledge all the people who have shared this journey with me, providing me with incessant support and help, in different forms.

Firstly, I would like to thank my tutor, Prof. Ernesto Sanchez, for guiding me, advising me, helping me over the years of my Ph.D. I will be forever grateful to you for giving me this opportunity. It has been a journey not only professional, but of life. Thank you. I would like to thank Prof. Matteo Sonza Reorda for giving me his trust, in every situation, and for always being attentive to my professional growth. I would also sincerely like to thank Prof. Alberto Bosio for his constant support, for giving me interesting opportunities, and for introducing me to this stimulating and exciting research world.

Then, I would like to acknowledge all the professors in CAD & Reliability Group, and my colleagues for encouraging me to conduct my research in an enjoyable environment. A special thank goes to Professors Paolo Bernardi, Luca Sterpone, and Giovanni Squillero for our very interesting collaborations. Then, to Davide Piumatti, Andrea Floridia, Aleksa Damljanovick, Esteban Rodriguez, and Riccardo Cantoro for sharing with me my best and worst moments, and to all my current colleagues in LAB3+7. Thanks, Gabriele Gavarini for entering our exciting "TheNNReliable" team, thank you for your very valid support and for sharing with me the last important moments of my Ph.D. Next, I would like to thank Professor Luigi Dilillo, Lucas Matana Luza, and Marcello Traiola for our very interesting path together.

Last but not least, special thanks go to my lovely family, to Kevin, and to my friends for their constant and warm presence and, most importantly, for the energy they give me and encourage me every day. My family with unmeasurable love have always protected, supported, and encouraged me. Without them, none of this would have been possible. Thank you dad, mom, Ilaria, grandpa. Thank you, Kevin, for being exactly as you are. Thank you for sharing this journey with me.

To conclude, I would like to thank all the people I have met over these long years, who in different ways and intensities have given me the courage to achieve this goal.

Abstract

The complexity of modern embedded systems has increased rapidly over the past few decades. The integration of many various technologies as well as the adoption of more sophisticated algorithms pose significant challenges in ensuring the robustness and the reliability of these systems. Indeed, together with the shrinking of technology nodes, even more systems leverage artificial intelligence based algorithms to cope with their ever-growing computing requirements. In this light, worthy of note is the emerging trend toward the adoption of brain-inspired artificial models, i.e., Artificial Neural Networks, in different fields, like automotive, robotic, avionic, etc. If from one side this results to be a very interesting solution, given their outstanding and near-human computational capabilities; from the other side this could be dangerous from the safety point of view. Actually, outsourcing important decisions to them (like deciding whether the car should stop to let a pedestrian pass), can be threatening for the following reason. As a matter of fact, being predictive model, they are not 100% accurate: even in fault-free scenarios, they can provide a wrong answer. Then, despite their very interesting capabilities which are appealing for both industry and academic, it is necessary to deeply investigate the reliability and the behaviours of those systems in faulty scenarios, especially because it has been demonstrated that with the shrinking of the technology, emerging devices are more prone to physical defects.

This manuscript is organized in two parts. The primary focus of this Ph.D. thesis is on artificial intelligence-based systems and on the assessment and the improvement of their reliability. This topic has been addressed comprehensively at very different abstraction levels and from different perspectives. Firstly, a characterization of the existing fault models is provided together with the identification of the possible vulnerabilities in ANN-based systems. Then, a key contribution of these research activities is the proposals of different fault injection tools and methodologies to easy and support the reliability assessment process. Specifically, this is done at

different levels: by addressing only the ANN model, or by considering the entire system entailing both the ANN software running on a target hardware system. The advantages and disadvantages of the different categories are detailed in the manuscript. At this level, the principal novelties are the identification of the critical bits in different data type representations, the establishment of critical neurons depending on the importance of a neuron as a class- and as a network-dependent entity; trade-offs analysis on data type representations based on two aspects, i.e., the memory footprint of the application and their reliability. In the end, relying on these analysis and results, strategies to mitigate the effect of faults have been proposed. The first proposed technique aims at redistributing neuronal computations on an AI-based SoC by leveraging integer linear programming. The second mitigation technique is an on-line testing solution based on the adoption of Software Test Libraries coexisting with the requirements of ANN algorithms.

Together with the reliability assessment of neural networks, this Ph.D. thesis covers also a further important topic: the hardware security of modern embedded systems. The complexity of emerging hardware systems led the industry to pursue new development processes: to build a SoC, a recent trend is to rely on third-party IP blocks to keep the cost low and to meet the deadlines. This outsourcing poses increasing concerns regarding the security of modern embedded devices. As an example, these IP blocks might come with malicious circuitry intentionally added by adversaries, namely Hardware Trojans. They are hidden inside the design and become active under certain rare conditions (such as input sequences). Otherwise, they can be always active since the power on of their host device. Their malicious functionalities can vary from leaking secret information, degrading the circuit's performance, creating backdoors for attackers, and many others. In the literature, new advances and progress have been done in this field, especially in proposing specific detection methodologies to discover hidden Trojans inside a given design. However, the existing gap lies in the HT benchmarks used to validate state-of-the-art methodologies. Indeed, they are obsolete and very simple, unable to represent the complexity of current designs and architectures. For example, they are injected on small 8-bit 8051 microprocessors, or simple cryptographic circuits. In this context, one of the main contributions of this Ph.D. thesis is the release of a set of Hardware Trojans benchmarks at the Register Transfer Level for an open-source pipelined RISC core. They have been designed by following the guidelines for creating hard-to-detect Trojans. Furthermore, the second contribution in this field is a pre-silicon

detection methodology for detecting RTL Hardware Trojans. This is build on a deep learning analysis of the dynamic and static properties extracted from the design RTL model. Experimental results show that the proposed technique is highly accurate in highlighting suspicious code sections. By carefully inspecting them, it is possible to unfold all the hardware trojans.

Contents

List of Figures **xiii**

List of Tables **xvi**

Introduction **1**

I Artificial Neural Networks Reliability **6**

1 Background and Related Works **8**

1.1 Fault Models in ANNs 8

1.1.1 Fault Propagation 12

1.2 Related Works 13

1.2.1 Fault Injection Tools and Frameworks 13

1.2.2 Advantages and Disadvantages of Fault Injection Methodologies 19

1.2.3 Related Studies on ANN Weights and Neurons 22

1.2.4 Related Studies on Mitigation Strategies 25

1.2.5 Software Test Library 26

2 Main Contributions **28**

3 Reliability Assessment at the Software Level **32**

3.1 Static Parameters in Artificial Neural Networks 33
3.1.1 Full Precision ANN Weights 34
3.1.1.1 Proposed Approach 34
3.1.1.2 Experimental Results 37
3.1.2 Reduced Precision ANN Weights 42
3.1.2.1 Proposed Approach 43
3.1.2.2 Experimental Results 48
3.2 Dynamic Parameters in Artificial Neural Networks 62
3.2.1 Proposed Approach 63
3.2.2 Experimental Results 67
3.3 Chapter Summary 76

4 Reliability Assessment at the Architectural Level 79
4.1 Proposed Approach 79
4.2 Experimental Results 82
4.3 Chapter Summary 85

5 Reliability Assessment at the Physical Level 86
5.1 Proposed Approach 86
5.1.1 Radiation Experiments 88
5.1.1.1 DUT Characterization 88
5.1.1.2 Radiation Tests on CNNs 89
5.1.2 Software Emulator 92
5.2 Experimental Results 94
5.2.1 Radiation Tests Results 94
5.2.2 Software Emulator Results 96
5.3 Chapter Summary 104

6 Mitigation Strategies 106

6.1 Neurons Redistributions on AI-oriented MPSoCs 106

6.1.1 Proposed Approach 108

6.1.1.1 Integer Linear Programming based Methodology . 109

6.1.2 Experimental Results 111

6.2 Software Test Libraries for ANNs 116

6.2.1 Proposed Approach 117

6.2.2 Experimental Results 120

6.3 Chapter Summary 126

II Security of modern devices 128

7 Background and Related Works 130

8 Main Contributions 134

9 Hardware Trojans 135

9.1 Proposed Benchmarks 135

9.1.1 Experimental Analysis and Implementation 137

9.2 Pre-silicon Detection methodology 141

9.2.1 Proposed Methodology 141

9.2.1.1 Control Flow Graph Extraction 143

9.2.1.2 Data Formatting 145

9.2.1.3 Classification 146

9.2.2 Experimental Results 147

9.3 Chapter Summary 153

10 Conclusions and Achievements 156

10.1 Future Directions 158

Bibliography **162**

List of Figures

1.1 The fundamental concepts of fault, error, and failure applied to an ANN-based system. 10

1.2 Fault Propagation in a full system [1]. 12

1.3 Schematic of Fault Injection approaches for Deep Neural Networks. 14

1.4 Comparing different Fault Injection Techniques. The x-axis reports the specific level: 1-*Low*, 2-*Medium-Low*, 3-*Medium*, 4-*High-Medium*, 5-*High*, 6-*Very High*. 20

2.1 Fault injection approaches for assessing the reliability of ANNs and ANN-based systems at different abstraction levels. In each situation, the lightning bolt icon illustrates where errors or faults can be introduced. 29

3.1 Single Precision IEEE 754 Floating-Point Standard 34

3.2 LeNet Masked Faults 38

3.3 LeNet Safe Observed Faults 38

3.4 LeNet Unsafe Observed Faults 39

3.5 Example of YOLO Predictions. 40

3.6 Tiny YOLO Masked Faults 41

3.7 Tiny YOLO Unsafe Observed Faults 41

3.8 YOLO Workload #4. 42

3.9 Custom Data Type. 43

3.10 On-line Weights Conversions. 44

3.11 Fault Injection Scenario [2]. 46

3.12 Critical Bits 49

3.13 LeNet-5 pre-trained weights distribution. 49

3.14 YOLO pre-trained weights distribution 55

3.15 Basic Scheme for a neuronal computation. 63

3.16 The critical neuron identification process: a practical example with the MNIST dataset. 66

3.17 MNIST LeNet: software fault injection campaigns on *random* and *critical* neurons. 70

3.18 SVHN ConvNet: software fault injection campaigns on *random* and *critical* neurons. 71

3.19 CIFAR-10 All-CNN: Software fault injection campaigns on *random* and *critical* neurons. 72

3.20 Network-oriented Analysis with a growing t percentage of critical neurons from the CoA. 74

3.21 Showing the robustness of the proposed approach based on the contribution of the CoA and the NoA (blue lines). Its effectiveness is compared against [3] (green line) and our proposed methodology without the contribution of the class-oriented analysis (red line). . . 75

3.22 Intersection between the critical neuron ranking obtained using the test set and the rankings obtained using subsets of the validation set. The figure shows how, for ResNet-32, the intersection remains stable for different values of *pNeu*. 78

4.1 Exploiting the pipeline technique to reduce the DNN simulation time. 81

4.2 Proposed Pipelined Multi-Level Fault Injector Tool 81

5.1 Diagram of the Proposed Approach. 87

5.2 Top-level diagram of the system. The DUT is highlighted. 91

5.3 A diagram of the proposed emulator. 93

5.4 Accuracy variation based on the increase of the E-SER with SBUs or stuck-at faults for the three CNNs. 98

5.5 Number of injected stuck-at faults for each single BE for each CNN application (y-axis on the left). Average accuracy value for each single BE affecting a run at a random time (y-axis on the right). . . . 99

5.6 The incidence of the most critical BEs on CNNs weights and biases. 101

5.7 Multiple block errors affecting the three CNNs under assessment. . 103

6.1 Neurons assignment in a AI-oriented SoC exploiting the SIMD configuration. 108

6.2 Fault Detection Time 118

6.3 CNN performance versus FDT 125

6.4 Resnet-18 performance versus FDT 126

9.1 RTL Hardware Trojans benchmarks inserted in the mor1kx CPU. . . 138

9.2 Flow of the proposed methodology 142

9.3 ROC curves for 4 different kernels including different set of extracted attributes (farther from the 45-diagonal, i.e., closer to the upper-left corner, the better) 150

9.4 Set of nodes belonging to Hardware Trojans as TP and FN 153

List of Tables

3.1 LeNet [4] 36

3.2 Tiny YOLO [5] 37

3.3 LeNet-5 Data Type Accuracy Loss [%] 50

3.4 LeNet-5 Fault List for Fault Injection Campaigns 50

3.5 LeNet-5 Fault Injection outcomes with respect to the **Golden Standard** version. 53

3.6 LeNet-5 Fault Injection outcomes with respect to the **Golden Custom** version. 54

3.7 YOLO Data Type Accuracy Loss [%] 55

3.8 YOLO Fault List for Fault Injection Campaigns 56

3.9 YOLO Fault Injection outcomes with respect to the **Golden Standard** version. 58

3.10 YOLO Fault Injection outcomes with respect to the **Golden Custom** version. 61

3.11 ANN Benchmarks 68

4.1 Pipelined Fault Injector timing details. 83

4.2 A comparison between the performances of the sequential fault injector wrt the pipelined. 83

4.3 Fault Injection Results 84

4.4 Sequential Framework vs Pipelined Multi-Level Framework 85

5.1 Estimated event rate for both test scenarios. 92

5.2 Summary of the runs that return a faulty accuracy at the end of the radiation tests on CNNs. 95

5.3 Details of SBUs and stuck-at bits injection for the *Float 32*, *Int 16*, *Int 8* CNNs with an increasing E-SER: 1x, 25x, 50x, 75x, 100x . . . 97

5.4 BE Injection Details 99

5.5 Details of block error injections for the CNNs *Float 32*, *Int 16*, *Int 8* with an increasing E-SER: 1x, 25x, 50x, 75x, 100x 103

6.1 Figures of variance when the chunks of neurons are assigned following a static scheduling. 113

6.2 Figures of variance when the chunks of neurons are assigned following the proposed ILP and Variance-based optimal scheduling. 113

6.3 RTL Fault Injection Results. 116

6.4 CNN timing details 121

6.5 Software Test Library Details 122

6.6 RI5CY stuck at faults details and test coverage 123

7.1 RTL Hardware Trojan benchmarks available on Trust-Hub [6, 7]. . . 132

9.1 Hardware Trojan Benchmarks Description 136

9.2 Synthesis Results 141

9.3 Experimental results of the four SVM classifiers 149

9.4 Experimental results of the Neural Network 152

9.5 Meaning of the confusion matrix in context of HT detection 153

Introduction

Nowadays, safety and security assessments for modern embedded systems are challenging tasks. Indeed, they have become increasingly complex and sophisticated due to the ever-growing computing requirements and capabilities. As a matter of fact, the growing complexity of emerging computing systems has called for enhanced computing paradigms. Among all the existing possibilities, Artificial Intelligence (AI)-based solutions and, specifically, brain-inspired computing models, have gained large interest in the industry and academia for their outstanding and near-human computational capabilities. Inspired by the human brain where billions of neurons process information in parallel [8], researchers have developed artificial models, named Artificial Neural Networks (ANNs) which mimic the functioning of the human brain [9]. The neural network history has its roots in 1943 with the seminar paper by Warren S. McCulloch and Walter Pitts [10]. For the first time, they described a formal neuron with capabilities similar to that of a Turing machine, and created a machine able to implement logical reasoning. The idea of exploring logic in terms of manifestations of a mental process forms the basis for the artificial intelligence world. From that historical time on, a large amount of studies made progress in improving the theory behind brain-inspired computations to build highly complex artificial models. For their superior qualities, they represent now one of the most used solutions for addressing complex computational problems.

AI-based systems surround our daily life, and, among other countless applications, they power our virtual assistants, transcribe our voice messages, recognize people in our phone, or, in our car. Moreover, they have found successful applications in several safety-critical domains, like automotive, robotic, avionic. Their adoption in human environments raises many concerns. For these reasons, assessing their reliability has become a crucial requirement, and, due to their complexity, there is a growing need to develop even more sophisticated methodologies to accomplish

this scope. It is also worth adding that neural networks are notorious for producing inaccurate inference results. They are not 100% accurate due to their nature as predictive models, which provides an essential aspect that must be considered when addressing the reliability of these systems. Adoption in road cars, for example, may lead to the violation of functional safety requirements (e.g., ISO 26262 [11]). As a consequence, recently, the research community has become even more interested in determining aspects such as the reliability and trustworthiness of AI-based systems.

It is worth saying that, ANNs are traditionally considered intrinsically fault-tolerant and tightly robust, being brain-inspired models. Indeed, our brain is known as the best information processing system, with its 10 billion nerve cells (i.e., *biological* neurons) and about 10,000 synapses for each connection starting from a neuron. Because of its plastic capacity to renew, repair, and reconfigure its neuronal activities, it can withstand synapses or neuron errors while still operating correctly [12]. Clearly, this fascinating plastic property can not be extended to *artificial* neurons, which being mathematical models, are not capable of self-repairing from errors. On the other hand, it is necessary to specify *why* it is claimed that they can be considered inherently robust and fault-tolerant models. Firstly, they own a certain degree of robustness because of their redundancy. In fact, such artificial models are composed of more neurons than they really need. This property is known as *over-provisioning* and constitutes a very key principle in neural networks fields [13]. Due to this excessive neurons budget, it has been demonstrated that over-provisioning leads to robustness [14]. In other words, this characteristic means that ANNs can accept a finite number of neurons failing without affecting the computation's outcome; beyond that number, accuracy declines gradually [15]. The second reason behind the claim of the intrinsic robustness is related to their distributed and parallel architecture. In those systems, the unit of failure is one single neuron or synapse. Examples are reported by IBM teams which recently presented neuromorphic implementations of convolutional neural networks where neurons can fail independently [16].

Under these assumptions, i.e., (i) the over provisioning and (ii) the single neuron-s/synapse as units of failures, artificial neural networks can be considered having an intrinsic robustness and reliability.

However, in the real world this is not totally true, because they are not used as pure mathematical models, but they are deployed on silicon. The first examples of digital architectures devised for neural networks have been systolic array structures,

very suitable to express their recurrence and parallelism [17, 18]. In that particular architecture, an individual processing element implemented the function of an associated neuron while the weights were stored in a circular memory. For another perspective, it means that a single physical fault affecting the hardware corrupted a single neuronal computation. Therefore, the single neurons as unit of failure was preserved in that case. Currently, neural networks are typically executed on high-performance GPUs or optimized custom ASIC designs, where a single hardware unit implements many neuronal computations. As a consequence, a single physical fault affects many neurons, tearing down the theory of the intrinsic fault tolerance.

A growing number of AI-based systems are built on ASIC design implementations, especially those exploited on the Internet of Things (IoT) world and for the edge computing (e.g., [19, 20]). The reason lies in their flexibility, which makes them suitable for a wide range of applications requiring low-power and low-cost resource-constrained embedded devices. On a final note, regardless the used architecture (e.g., GPUs, FPGAs, ASICs), new emerging devices at nanoscale dimensions are more prone to manufacturing defects and transient faults [21]. In other words, the probability that parts of the hardware fail due to the occurrence of physical faults increases as the shrinking of semiconductor technologies continues.

In light of these considerations, it starts to be crucial to evaluate the reliability of systems based on artificial neural networks and, generally, artificial intelligence algorithms. This is further motivated by the following considerations:

- Being predictive models, they are not 100% accurate. Even in fault-free scenarios, they can provide a wrong result.
- In many contexts and applications, they make choices for us.
- There is a growing push from the industry to deploy them on safety-critical systems (e.g., self-driving cars).

Part I of the manuscript addresses this topic and describes the principal contribution of my Ph.D. thesis in this field. The primary focus is on the reliability assessment of artificial neural networks and AI-based hardware systems. First, an in-depth investigation has been done to understand the main vulnerabilities of those systems; then, tools and methodologies have been proposed to perform reliability assessments, and then, mitigation solutions are proposed to improve their reliability.

In this manuscript, a detailed background in the area is given in Chapter 1: the principal vulnerabilities and fault models are addressed, and a specific section on related works is provided. The novelties and the main contributions are illustrated in Chapter 2. Next, Chapters 3, 4, and 5 detail the main research works that have been done to assess the reliability of ANNs at very different abstraction levels: respectively, at the *software* by considering only the neural network model, at the *hardware* by presenting the architecture of a RTL fault injector, at the *physical* by describing the architecture of a software emulator which is capable of injecting real faults retrieved from radiation test campaigns. Finally, leveraging the outcomes coming from the previously described assessment phase, mitigation strategies are proposed in Chapter 6 to improve the reliability of ANNs running on hardware embedded systems. An Integer Linear Programming (ILP)-based method is proposed to redistribute neuronal computations on a resource-constrained AI-oriented ASIC device to improve its reliability. Finally, the adoptions of Software Test Libraries (STLs) for the on-line testing and their coexistence with the requirements of artificial neural networks are investigated in a study.

ANNs Reliability is not the only issue covered in this manuscript. Part of the research in these years focused also on the security of modern embedded devices. In the last years, the growing complexity of modern devices and the fabrication costs led the Integrated Circuit (IC) industry to pursue a new global business model. In that regard, even more companies around the world are deeply involved in all phases of the IC supply chain. The outsourcing of part of the process to untrusted third-party entities raises growing concerns about the hardware security of the final products. Particularly, Hardware Trojans (HTs) are gaining worldwide attention not only from academia and industries, but also from government bodies [22]. Hardware Trojans are malicious and intended alteration of a circuit, that endangers the trustworthiness and the security of the hardware, leading to unwanted behaviour. As an example, a HT may leak secret information, change the circuit functionality or degrade the performance. In the literature, the vast majority of the existing detection techniques are applied at gate-level [23][24]. However, a growing examples of HTs inserted at the RTL are coming out, due to the flexibility for implementing various malicious functions. Hence, more RTL HTs detection techniques are needed.

Part II of the manuscript deals with this issue: the security of modern embedded systems. Chapter 7 provides background knowledge in the field together with an accurate analysis of state-of-the-art research works. Chapter 8 introduces the main

novelties in the field. Then, in Chapter 9, the two main contributions in this topic are described: first, the description of newly released RTL Hardware Trojan benchmarks for pipelined RISC microprocessor cores. Second, a machine learning-based strategy to detect RTL Hardware Trojans at the pre-silicon phase of the supply chain. The analysis is based on dynamic and static properties extracted from the RTL model of the design under assessment.

To conclude, Chapter 10 includes a summary of this research, together with useful information on future research directions and recommendations.

Part I

Artificial Neural Networks Reliability

Chapter 1

Background and Related Works

The origin of Artificial Neural Networks (ANNs) dates back to 1943 with the work published by W. Pitt and W. McCulloch [25], where the concept of a neuron as a computing unit was formalized for the first time. They are computing models composed of computing nodes, i.e., neurons, connected to one another through communication links, i.e., synapses. The role of an ANN is essentially to mimic the behaviour of a function. For this purpose, ANNs are first trained by observing the function input-output correspondence. Then, they can reproduce the learned function behaviour by propagating the computed values from the input neuron(s) to the output neuron(s) [9]. The propagation is regulated by weights and biases parameters, which are specific to each synapse and neuron, respectively. Neurons are arranged in layers, at least one input layer, one intermediate (or hidden), and one output layer. An ANN composed of more than three layers, i.e. an input layer, an output layer and multiple hidden layers, is named a *deep* neural network.

The intent of this chapter is to provide the reader with basic definitions and concepts in the field. Section 1.1 overviews the different fault models existing in ANNs. In Section 1.2, related works in the topic are described.

1.1 Fault Models in ANNs

It is known that Artificial Neural Networks have built-in fault-tolerance properties due to their distributed and parallel structure, as well as for their redundancy due to over-provisioning [13]. However, the evaluation of their fault tolerance is not

a trivial task because there are no common systematic methods or tools for the assessment [21]. The intent of this section is to introduce and clarify basic definitions related to fault types and fault models in ANNs and ANN-based systems.

Fault Injections (FIs) have long been acknowledged as appealing techniques for assessing the dependability of systems under test among all available testing approaches. Such procedure consists of introducing faults/errors into the system under test and checking its behaviour in response to them. To address reliability issues in ANNs, it is first necessary to clarify *where* faults can be injected and at which abstraction level. Dealing with different abstraction levels, before delving into the distinct reliability assessment approaches, it is worth reminding the fundamental concepts of defect, fault, error, and failure [26]. A *defect* in an electronic system is the difference between the intended design and the implemented hardware. Typical defects are due to the process, the material, the age of the device, and the package. The representation of a defect is a *fault*, an anomalous physical condition that may lead to an error. An *error* is the exhibition of a fault in a system that might not or might be propagated and, in this last case, give rise to *failures*. An example of their application to the neural network field is reported in Figure 1.1, where the two principal levels are illustrated: the architectural and the behavioural one. The former includes the target hardware device, the latter the artificial neural network application. If the injections are performed at the behavioural level, we can talk about errors and failures, whereas if they address the physical abstraction level, also referred to as architectural, we will refer to faults. For the system represented in Figure 1.1, the defect is the short circuit to the ground, while the fault is the signal b stuck-at logic 0.

In this light, reliability assessments in the neural network domain can focus on: (i) the neural network model as a technology-independent software application, (ii) the system comprising both the neural network model and the final hardware architecture running it. In the first case (i), a designer might be interested in evaluating the robustness of the neural network, regardless of the target device on which it will be deployed. As a consequence, a fault injection process can target only the entities belonging to the ANN model, i.e., neurons and synapses. According to [15], each neuron must be considered as a single entity that can fail independently of the failure of any other. This is also true for synapses. In line with the classification proposed in [21], errors in artificial neural networks may occur in the following elements:

- **Communication channels**: The communication link between two neuron cells can be broken due to faulty interconnections or noise.
- **Synaptic weights**: Weights represent the strength of the connection between two neuron cells.
- **Neuron body**: It constitutes the core of the neuron cell and includes both the summation and the activation function. An error affecting the neuron body can be distinguished in two categories: *crash* and *byzantine*. In the first, the neuron completely stops its activities and saturates to positive/negative values. In the second, it transmits arbitrary values.

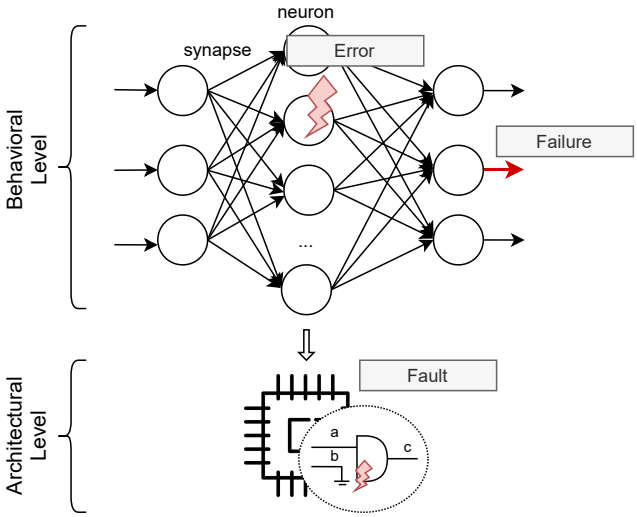


Figure 1.1 The fundamental concepts of fault, error, and failure applied to an ANN-based system.

Therefore, a FI campaign can mimic the occurrence of such error types. When running the FI campaigns, all these sources of errors can be considered to assess the ANN resilience. Other similar fault model classifications for ANNs are proposed by Sequin and Clay in [27], Chandra and Singh in [28], and Bolt in [29]. Specifically, in [29], the author presents a method for developing fault models at the abstract level by taking into account several characteristics, such as the fault location and the time of occurrence. In practice, having the high-level model of the ANN under test, these errors can be injected in the following faulty locations: weights, biases, communication links, and input or output neurons.

As regards the second case (ii), a designer might be interested in evaluating the robustness of the entire system before deploying a given ANN on a final device. Therefore, together with the error types affecting the ANN software model, all the physical faults affecting the hardware model should be taken into account. Faults affecting electronic devices can be classified for their temporal characteristics as permanent or transient. The former is stable with time and represents irreversible physical damage. The latter is active only for a short period of time and arises as a result of external disturbance or abnormal conditions or events (e.g., particle strikes). Starting from this broad fault types classification, the following fault models have been proposed over the years as abstractions of physical defects in electronics devices:

1. **Stuck-at**: Individual elements of the electronic device are tied to a state. For example, in a memory array, the bit can be stuck at a logical '1' or '0', and regardless of the operation, the read result will be the same.

2. **Bit-Flip**: Individual memory elements of the electronic device had their logical state changed. This change in the logical state can be recoverable by resetting the value on the element.

The stuck-at fault is a very common fault model. Indeed, it has been shown that many transistor and interconnection defects can be modelled with fair accuracy as permanent defects at the logic level. On the other hand, a random bit-flip model can represent the occurrence of transient faults, usually affecting registers or memory regions. Transient faults (i.e., soft errors) may be caused by different sources of interference phenomena such as electrical noise, electromagnetic interference and impinging ionizing particles. However, it is fair to say that today these two fault models are not able to cover the newer fault mechanisms of the deep-submicrometer technologies: new fault models are needed to deal with delays, stuck-opens, open-lines, bridgings, and transient pulses.

Nevertheless, it has been demonstrated that the stuck-at and bit-flip models allow a good investigation of the fault tolerance also at behavioural level [21] and, for this reason, they have been widely used for reliability studies. For instance, an error affecting the communication channels of a neural network can be modelled as a single or multiple stuck-at-0 or stuck-at-1 faults affecting one or more bits of the channel. Similarly, an error in the synaptic weights can be represented with a stuck-at

fault (o bit-flip) impacting one or more bits of the weight parameter. For example, if adopting a 32-bit floating-point representation, an error in a weight means that one or more bits of the 32 can be faulty. The same reasoning can be applied for representing a crash or byzantine neuron. A neuron can be considered dead if it is no longer transmitting values: this error can be modelled with a stuck-at-0 at its output. Contrarily, a Byzantine neuron can be modelled as a stuck-at-*value*.

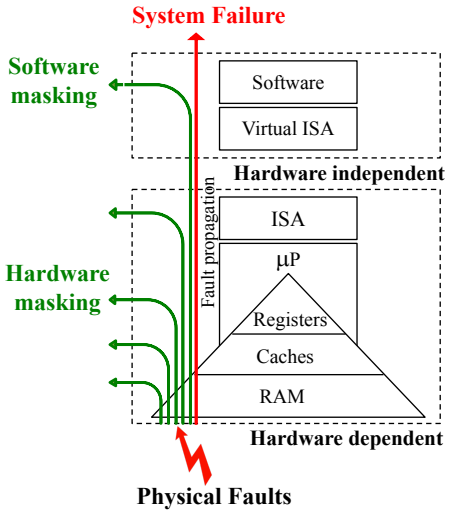


Figure 1.2 Fault Propagation in a full system [1].

1.1.1 Fault Propagation

So far, all the possible fault models existing in neural network based systems have been detailed. Nevertheless, it is worthy of note to discuss *how* hardware faults can be propagated in a system. The hardware system can be affected by faults due to physical manufacturing defects. As shown in Figure 1.2, faults can propagate through the various hardware structures that encompasses the entire system. However, it is likely that they will be masked during the propagation, either at the technological or at the architectural level [1]. When a hardware fault reaches the software layer, it might damage data, instructions, or control flow. These errors can cause incorrect program execution, resulting in erroneous results, or they might block the application from running, resulting in unexpected termination or application hangs. At the same time, the software stack can hide mistakes, avoiding failures at the output of the application. These phenomena are both inherently crucial for system reliability

and a difficult challenge for engineers who must verify the safety of their systems. Additionally, it must be said that the scenario illustrated in Figure 1.2 is valid for *every* kind of application. It has been pointed out for an interesting reason: apart from the above-described system (hardware or software) masking ability, neural network applications possess an intrinsic masking ability, by their own. It is therefore very interesting to understand how these properties coexist and combine.

1.2 Related Works

In the literature, several research studies have been done in this field. As stated, the research community has been even more interested in understanding the reliability of artificial neural networks and systems based on them. This section first cover the principal fault injection techniques and tools proposed at very different levels, and discusses their main advantages and disadvantages (Sections 1.2.1 and 1.2.2). Next, it presents related works in the literature: techniques to study the resilience of weights and neurons in neural networks are presented in Section 1.2.3, and to improve their robustness in Section 1.2.4. Finally, an overview of the Sofware Test Libraries (STLs) is given in Section 1.2.5.

1.2.1 Fault Injection Tools and Frameworks

In [30], we propose a preliminary classification for classifying the state-of-the-art FI methodologies for the assessment of ANNs reliability. Depending on the abstraction level, they can be ranked in the following way.

- **Simulation-based:** The injection process is carried out without relying on the physical device finally running the ANN. Moreover, depending on the abstraction level, they can be further ranked.
  - **Software Level:** The injections are performed on a high-level model of the ANN, not considering any details of the actual hardware architecture.
  - **Hardware Level:** The injections are performed on a more accurate model of the ANN that simulates the target hardware architecture. This can be described at the register transfer level (RTL) or gate level, for example.

- **Platform-based:** The measurements and the analyses are performed directly on a physical device that emulates the final implementation of a design using FPGAs or on physical platforms running the ANN under assessment, e.g., CPUs and GPUs.
- **Radiation-based:** The reliability assessment is performed in the actual platform running the ANN under assessment by means of external electromagnetic interference, such as ionizing particle incidence through accelerated radiation test campaigns.

A schematic illustration is given in Figure 1.3.

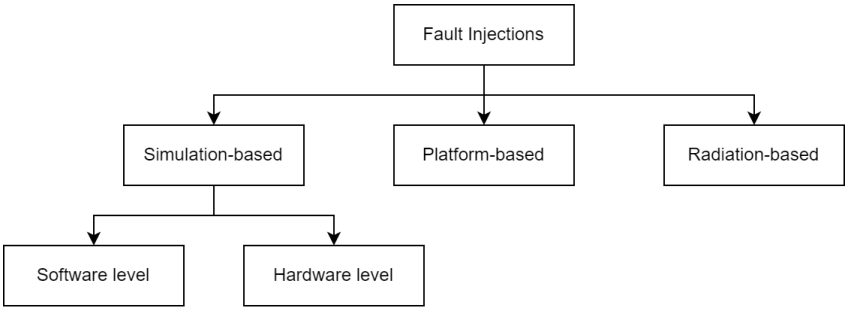


Figure 1.3 Schematic of Fault Injection approaches for Deep Neural Networks.

The great majority of published works in this topic consists in **simulation-based software-level** methodologies: they are the most frequently used for having lower costs, for being faster, more controllable, and easier to deploy. To investigate the weaknesses of the neural network, and to be independent of any potential hardware architecture, a Fault Injection (FI) framework is typically developed to inject errors in a high-level model of the DNN under assessment. Two principal ones are publicly available to run neural network applications: TensorFlow [31] and PyTorch [32]. They are open-source machine learning frameworks for DNN-based applications that in the last years enjoyed enormous popularity among engineers and developers. Clearly, these represent the most popular solutions, but many others are available in the research community: N2D2 [33], Darknet [34], or even custom tools have been created for the purpose. In [35], Chen *et al.* present TensorFI, a high-level FI framework for DNN. It is a flexible tool that can be used to inject faults at the TensorFlow graph level, particularly at the output of the TensorFlow operators. NN models are represented as a sequence of operations (nodes) in TensorFlow, which

are connected in a computational graph. It means that faults can be injected at the TensorFlow graph level (in the high-level programming logic) by corrupting the outputs of the most common operators (e.g., *Add, Sub, Mul, ReLU, Conv2D*). Since TensorFlow does not expose the operators and most of the execution occurs "behind the scenes", TensorFI is created by duplicating the original TensorFlow graph. Then, at inference time, it is possible to choose between the golden operators and the faulty ones. Only transient faults are injected in one of the following formats: by flipping a single bit in one (or all) the data item(s) of the target operator; by shuffling one (or all) the data item(s) of the target operator into random values; by changing the output of the target operator into all zeros. A further TensorFlow-based software-level FI framework is described in [36]. It is written in Python and, similarly to TensorFI, it uses the TensorFlow graph to inject errors into the most common mathematical operations *Add, Sub*, and *Mul*. Three types of errors are introduced at the output of the faulty operator: a single bit-flip, a random value instead of the computed one, all zeros. On the base of [35], an efficient fault injector for finding the safety-critical bits in DNN-based application is proposed in [37], namely BinFI. The goal of the framework is to identify the safety-critical bits that are most vulnerable to hardware transient faults (soft errors) and protect them at low cost. Since the purpose is to pinpoint *all* the safety-critical bits of the program to measure its overall resilience, a random FI based on statistical sampling would not have covered all the cases, and thus, was not suited for the scope. An exhaustive FI would have resulted in substantial performance overheads. Indeed, depending on the number of bits in the program, the time would have been directly proportional. Therefore, a binary search FI approach (BinFI) was used by exploiting the monotonicity feature of the functions used in ML applications, which is extremely useful to prune the FI space and efficiently detect the safety-critical bits. The framework is built on the top of TensorFI. To identify the critical bits of the DNN, faults are injected in the processor datapath (pipeline registers and ALU) and propagates to the software level (the ML application). Note that they consider faults that are not masked before reaching the software layer (as described in Section 1.1.1, as masked faults do not affect the execution of the application. It is assumed that the main memory, cache and register file are protected by error correction code (ECC) or parity, and thus, not considered as faulty locations. In line with prior works [35, 36] and studies on how hardware faults lead to erroneous values (instruction-level errors in [38]), errors in BinFI are injected directly into the output value of TensorFlow operators.

On the other side, based on PyTorch, Mahmoud *et al.* propose in [39] a runtime perturbation tool for DNNs, named PyTorchFI. It allows performing FIs in weights and/or neurons in convolutional operations. Perturbations in weights are performed offline by modifying the weight tensor, while neuron values are modified during the forward pass of a computational model by exploiting the `hook` functionality. Even though PyTorchFI operates at the application level of DNNs, it can also model lower level faults (such as register-level) by mapping them to single or multiple bit-flips in single or multiple neurons. FI campaigns are performed to study the reliability of six neural networks adopting an INT8 quantization: AlexNet, GoogleNet, ResNet-50, ShuffleNet, SqueezeNet, VGG-19. Moreover, interestingly, they show that PytorchFI runs at the native speed of silicon, as it requires no code instrumentation for error modeling. The emulations on both CPUs and GPUs show that all the inferences (with and without PyTorchFI) typically take less than 0.2 seconds.

Software methodologies are relevant for assessing the DNN resilience to errors and, in particular, for characterizing the criticality of the layers and the various DNN topologies. However, these analyses might not be complete to represent the real behaviour when deployed on a physical hardware device. It is evident that software-level simulations and theoretical analyses may lack the information of the underlying hardware platform and are relatively less accurate [40]. As outlined in the introduction, the choice of the hardware architecture hosting the DNN plays a key role in the final reliability of the system. Therefore, it might be needed to study the overall system resilience to faults *before* reaching the fabrication phase. Working at the software level allows the injection of errors only on neural networks parameters (such as synaptic weights, activations values) or on high-level programming logic representing the basic operations of DNN models. On the other hand, **simulations at the hardware-level** enable a wider spectrum of possibilities. The lower the abstraction level, the higher the degree of freedom. For these reasons, it is required to run the fault injections on a more accurate Hardware Description Level (HDL) model of the target hardware architecture running the DNN. HDL models give the opportunity to perform such studies comprehensively with accuracy and exactness close to the real hardware. However, it is also worth saying that the lower the abstraction level, the higher the simulation time.

A good compromise between a pure software approach and a pure hardware one is presented in [41], where a software technique meets the precision and accuracy of the hardware abstraction level. FIdelity [41] is a resilience analysis framework that

has been developed to study the behaviour of hardware errors in DL accelerators. The framework is able to model a class of hardware errors (transient) in software with high fidelity, only leveraging on high-level design information obtained from architectural descriptions. Indeed, using just the information coming from design plans, block diagrams, architectural descriptions, and estimated values, FIdelity is able to generate accurate software fault models without the need to access the RTL description or the presence of a RTL description at all. Note that RTL is not likely to be available during the early phases of the design process. By defining a Reuse Factor Analysis, they are able to obtain the set of faulty output neurons, i.e., output neurons that are affected by a specific hardware fault. According to the hardware-level faults derived from the architectural analysis, random bit-flips are injected at specific input values, weights, output neurons of the DNN under assessment. To demonstrate that FIdelity software fault models are accurate, the authors use NVDLA [42], an open-source accelerator provided by NVIDIA. RTL injections are performed to compare the software fault models generated using the Reuse Factor Analysis with the information obtained from the RTL simulations. FIdelity is built on TensorFlow, and the DNN models used for its validation are: Inception, ResNet-50, and MobileNet which are trained on ImageNet and Cifar-10; Transformer trained on IWSLT14, a dataset containing about 160K sentence pairs, english-german and german-english; YOLO, trained on COCO dataset. Interestingly, this paper also analyses the effects of the quantization to 16- and 8-bit integers (INT16-INT8) as well as the half-precision floating-point (16-bit FP), thanks to the TensorFlow support for playing with reduced-precision data representations.

Driven by analogous motivation of [41], Li *et al.* [43] propose a framework for studying the propagation of transient faults (i.e., soft errors) in DNN-based systems. The study addressed the reliability of DNN accelerators, in particular a recently proposed one, Eyeriss [44]. Due to the lack of the RTL implementation, they modified an open-source DNN simulator framework, Tiny-CNN, to map each line of code in the simulator to the corresponding hardware component of the DNN accelerators under assessment. In this way, they are able to randomly inject faults into the data-path of accelerators and into the buffers of the Eyeriss DNN accelerator. The error propagation behaviours are classified according to the structure of the neural networks, the data types, the position of layers, and the types of layers. Four DNNs are investigated: AlexNet, CaffeNet, NiN, and ConvNet.

Salami *et al.* in [40] present a simulation-based hardware-level FI framework for performing an in-depth vulnerability analysis of a hardware accelerator described at the RTL. The assessment is performed by considering both the application-level specifications (the DNN weights, inputs, and the intermediate values) and the architectural-level ones (the specific data representation and the amount of computational resources, i.e., the PEs). In this paper, they propose an HLS approach to characterize the effects of both permanent and transient faults. Faults are injected during the inference cycles on a sub-set of registers: those that are in charge to store the neural network parameters, i.e., the weights, input values, and intermediate ones. To assess the sensitivity of the neural network layers, permanent and transient faults are also placed in the layer registers (individually). Also, the activation values are studied, particularly in two state-of-the-art activation functions: positive saturating linear and logarithmic sigmoid. So far, a similarity could be found with the injections made at the software level. However, having the HDL model running the DNN under assessment, Salami *et al.* performed also architectural-level analyses. First, they played with the architecture of the accelerator to study how the amount of processing elements can impact the fault propagation behaviour. FIs are performed for different numbers of PEs, i.e., 64, 256, 1024, revealing a direct proportionality between the number of PEs and the reliability of the accelerator against permanent and transient faults. Second, they studied the sensitivity of sign, digit, and fraction sections of the FxP data representation.

Next, some reliability assessment methodologies are performed by running FIs directly on an **emulation platform**. In [45], the authors present Ares, a framework for quantifying the resilience of DNNs. It enables the DNNs execution directly on the GPUs and targets permanent faults occurring in the memory, which is the unit hosting the weights. At the application level, errors are injected in the weights, the activations and the hidden states through bit-flips. Particularly, they are injected at construction time (static) and evaluation time (dynamic). The former are injected off-line, before the inference is executed. The latter are injected during the inference phase, introducing a minimum performance overhead. Ares is built on the top of Keras [46], which takes high-level DNN descriptions specified in Python and executes them using either Theano [47] or TensorFlow back ends.

Concerning the emulation on FPGAs, several tools for the reliability assessment of DNNs have been introduced. De Sio *et al.* proposed FireNN [48]: an emulation platform for evaluating the reliability of DNNs. The methodology exploits the

reconfigurability of FPGAs to mimic faults affecting the hardware running DNNs (e.g., stuck-at faults, delays, conflicting connections, and others). Particularly, the framework emulates the effects of single event upsets with random bit-flip injections in the configuration memory of the programmable logic of Zynq All Programmable SoC, running specific layers of the targeted DNN. The FI framework combines software and hardware levels through the extension of PyTorch and PyNQ[49] frameworks and exploits Vivado HLS to develop a custom IP core for performing convolutional operations. In the paper, the evaluation is performed on the fifth convolutional layer of the AlexNet network.

The **radiation-based** FI category relies on the exposure of the system to an accelerated radiation source, e.g., atmospheric-like neutrons. Being radiation a source of perturbations in electronics devices, in the literature, several radiation-based approaches have been proposed. For example, in [50], the authors evaluated the impact of neutron-induced SEUs on a CNN (LeNet-5) implemented with three different levels of approximation on the data representation. In this study, the target hardware is the memory device that hosts the network parameters and input images. A similar approach is presented in [51], where a 2- and 3-D Flash memory storing the weights of an ANN is exposed to X-ray irradiation. Also, in [52], three different NVIDIA GPU architectures are exposed to a neutron beam targeting the study of error propagation in computing resources. Finally, targeting an FPGA-based architecture, Libano *et al.* in [53] analyse the SEUs influence on three versions of a MNIST CNN implemented in an SRAM-based FPGA. Most of the works that assess the reliability of ANN applications via radiation-based approaches use atmospheric-like neutrons as a radiation source. However, as mentioned above, for a specific environment, specific radiation sources like protons, heavy-ions, electrons, among others, should be considered.

1.2.2 Advantages and Disadvantages of Fault Injection Methodologies

We introduce three different metrics for presenting the trade-offs between the state-of-the-art fault injection approaches:

- **Costs:** It refers to the costs needed to carry out the reliability assessment, including both resources and time.

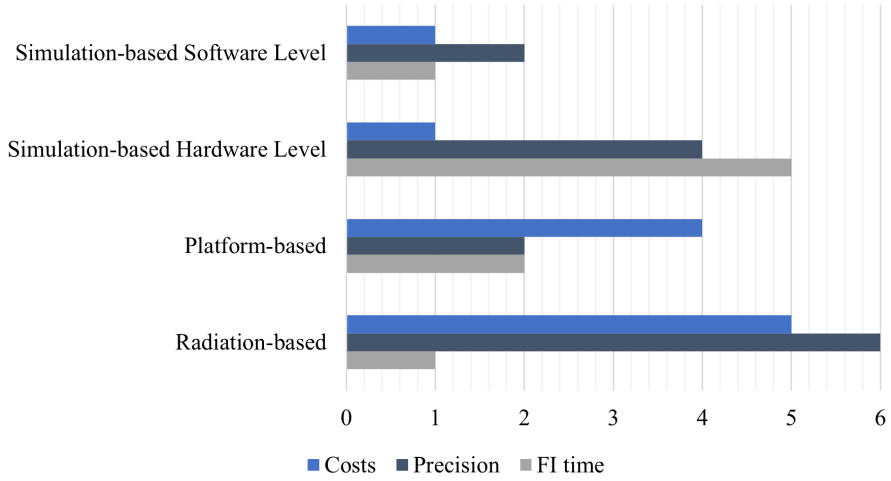


Figure 1.4 Comparing different Fault Injection Techniques. The x-axis reports the specific level: 1-*Low*, 2-*Medium-Low*, 3-*Medium*, 4-*High-Medium*, 5-*High*, 6-*Very High*.

- **Precision:** It means how much the FI procedure is close to reality, and the obtained results are accurate and realistic.
- **Fault Injection Time:** The amount of time that the injection process takes to complete a single injection cycle.

In Figure 1.4, we assign a specific level to these metrics and grade them as *Low* (1), *Medium-Low* (2), *Medium* (3), *High-Medium* (4), *High* (5), and *Very High* (6). As for the costs, because simulation-based approaches do not need the development and purchase of specific electronic devices to conduct the assessments, they are the most cost-effective. Moreover, when the HDL model is available, the costs are low and, anyhow, reduced compared to other FI techniques. When working with platform-based techniques, economic costs increase since they rely on the purchase and usage of specific validation or emulation devices (e.g., GPUs, CPUs, and FPGAs). A further benefit is also due to the fact that (i) they can be utilized again once the FI campaigns are completed, and (ii) they can be parallelized to increase the performance. The radiation-based procedures are the most expensive ones for three principal reasons: access to an irradiation facility, hardware setup development, and the low possibility of reusing the irradiated devices.

The precision with which the four FI procedures provide findings varies and is dependent on how well these approaches simulate the incidence of realistic system

faults and how near they are to real. Radiation-based FI techniques achieve the maximum level of precision, as radiation-induced faults directly impact the silicon implementation of the device under test. This enables the DNN model to be accurately characterized. On the other hand, simulation-based hardware-level FIs have a good level of precision. Due to the adoption of the HDL model (either RTL or gate-level), their injection procedure can be considered close to the actual silicon implementation. For this reason, they are credited with a medium-high precision level. Contrarily, platform-based and simulation-based software-level FIs both have a low-medium precision level for the following reasons. The incidence of realistic hardware faults is mimicked using sophisticated software fault models, where errors are injected at the software or algorithmic level. Specifically, when a DNN model is written in C or C++, it can be compiled and executed directly on a physical hardware device; hence, the injected software errors can be close to the faults they attempt to reproduce. In higher-level programming languages or tools, such as Python, PyTorch, and TensorFlow, FI frameworks introducing errors at the algorithmic level are exposed to a more elaborate compilation chain. Consequently, the lower the programming language level adopted for the DNN application, the higher the precision. One of the advantages of conducting simulation-based reliability assessments at the software level is the possibility of characterizing the vulnerability of neural networks independently of the target hardware device and, in particular, driving analyses on layers, data types, weights, and network's parameters.

However, when a more thorough reliability evaluation is necessary, the injection campaigns should additionally encompass the target hardware that will ultimately execute the DNN under test, clearly when the device's HDL model, whether RTL or gate-level, is provided. In this second scenario (such as in [30, 40]), hardware-level FIs can achieve better accuracy of the results, closer to the silicon implementation.

In a recent paper [54], we propose using realistic fault models (retrieved from radiation test campaigns) to inject at the software level in a CNN application, with the aim of enhancing the precision level of simulation-based FIs at the software level. The methodology is presented in Chapter 5.

Regarding the last metric (i.e., the fault injection time), it is worth considering that it is very difficult to exactly compare the time required to run a single FI among the existing fault injection approaches. Indeed, many variables are involved and

responsible for determining the fault injection time, such as the parallelization of the experiments, the adopted tools, the specific radiation source.

The problem associated with simulation-based FIs at the hardware level is that HDL simulations are extremely time-consuming. Clearly, it depends on the complexity of the neural networks under assessment and their HDL description. For example, a small CNN with only seven layers can take about 25 minutes to run a single inference [30]. Furthermore, existing commercial fault simulation tools are not tuned and neither optimized to face the complexity of the state-of-the-art DNN applications [55] (with billions of neuronal computations). This means that a FI at the hardware level is accurate but very costly in terms of simulation time. Therefore, reliability assessments at the hardware level typically consider only neural networks of limited size: a 6-layer fully connected classifier in [40] and a 7-layer CNN in [30]. By contrast, simulation-based FIs at the software level are not concerned with this non-negligible limitation. Actually, the neural network under consideration can range from 2-layer neural networks to more complex and deep networks, such as VGGNet and ResNet.

1.2.3 Related Studies on ANN Weights and Neurons

In [56, 57], the authors investigate the reliability of CNNs when 32-bit **floating-point** values are used as data type for representing their weights. Results show that the most significant bit of the exponent part (i.e., bit 30^{th}) is the most critical one. Interestingly, we found out the same result with our methodology, as described in the next sections. In [52, 58], the authors evaluate the reliability of one CNN executed on three different GPU architectures (Kepler, Maxwell, and Pascal). The soft error injection has been done by exposing the GPUs running the CNN under controlled neutron beams. A similar but wider approach is detailed in [59], where the authors assess the reliability of a 54-layers DNN (NVIDIA DriveWorks) through FI experiments and accelerated neutron beam testing for permanent and transient faults, respectively. Faults are injected on the DNN weights and on the input images. All inferences are executed on the Volta GPU and only target 32-bit floating-point values. Moving forward, Li *et al.* present in [43] a different analysis. They characterize the propagation of soft errors from the hardware to the application software of different CNNs. The injections are performed by using a DNN simulator based on open-source simulator framework, namely Tiny-CNN. Thanks to the flexibility of the

simulator, it is possible to characterize each layer for a more precise analysis. In this article, CNNs with six different data types are considered: 64-bit double precision floating-point, 32-bit single precision floating-point, 16-bit half precision floating-point, 32-bit fixed-point (with two different radix points), and 16-bit fixed-point. Overall, they conclude that, the larger the dynamic value range of the network's data type, the higher the likelihood of having large deviations in values in the event of faults leading to wrong predictions. Furthermore, a different framework is shown in [45]: Ares, a light-weight DNN fault injection framework. The authors present an empirical study on the resilience of three prominent types of DNNs (fully connected, CNNs and Gated Recurrent Unit). In particular, they focus on two **fixed-point** data types for each network: $Q_{3,13}$ i.e, 3 integers and 13 fractional bits, and $Q_{2,6}$. Their experiments demonstrate that the optimized $Q_{2,6}$ data type is 10x more fault-tolerant. The reason lies in the fact that the unnecessary larger range of integer values increases the chance of failure happening. It is worth noting that this result is in line with ours, presented in Section 3.1.2. It is a common trend to explore fixed-point computations for ultra-low power embedded systems with a limited power budget, e.g., [60]. Finally, in [40], the authors analyse the reliability of a DNN accelerator by following a HLS approach. They characterize the effects of both permanent and transient faults by exploiting a fault injector framework embedded into the RTL design of the accelerator. Faults are injected during the inference cycle only on a subset of registers: those that are in charge to store weights, input values, and intermediate ones used throughout the inference job, without considering the effects of faults in the other data-path units. As for the used data representation, they perform the experiments by only adopting a 16-bits fixed-point low precision model, claiming a negligible accuracy loss with respect to a full-precision data model. Additionally, it is worth also mentioning the research contribution of [41] and [53], where the authors investigate the reliability of convolutional neural networks by exploiting both the 16-bit and 8-bit integer data representation. Then, moving towards an even smaller data dimension, related works in [61] and [62] exploit reduced bit-widths. The former uses 5-bit and 3-bit fixed-point data types and a binary representation. The latter performs reliability assessment analysis on a binary neural network, where only 1 bit is used to represent the parameters (weights and biases).

Understanding the importance of individual neurons is currently a relevant topic to deal with the problem of complex DNN models running on resource-constrained devices. Indeed, neural networks are both computationally and memory intensive,

making them difficult to deploy on embedded systems. Many researchers have provided pruning techniques to remove either redundant neurons or connections from over-parameterized neural network models. The first pruning algorithm was proposed by LeCun in the 1990s [63], paving the way for several similar solutions. In [64], a three-step method is described to cut redundant connections (i.e., weights) by learning the important ones and retraining the remaining sparse network. Without any loss of accuracy, they can reduce the number of connections by 9x and 13x. A second paper provides an algorithm to remove neurons whose importance is below an optimal threshold [65]. To understand if neurons or connections can be removed, it is a common approach to use tuned thresholds or explore machine learning approaches [66], albeit rarer. The importance of neurons in a neural network is also addressed by Venkataramani *et al.* [67] to design energy-efficient hardware implementations of large-scale neural networks. To characterize the importance and the resilience of each neuron, the back propagation of error gradients is used to discover those that impact output quality the least. Neurons that contribute the least to the global error are more resilient and can be approximated with energy-efficient neurons. The process implies that, for each input in the training set, the error at the output is computed using forward propagation. Then, the errors are back propagated to the outputs of individual neurons to get their average error contribution over all inputs in the training set. Finally, the errors are ordered based on the magnitude of their average error contribution. On this base, the same methodology is exploited by Liu *et al.* in [68] to determine the fault tolerance capability of each neuron, albeit for a different scope. This measure for the *i-th* neuron is computed with the derivative of cost function E concerning output node y_i. A low δ_{i} corresponds to a more resilient neuron, and vice versa.

$$\delta_{\mathrm{i}} = \frac{\partial E}{\partial y_{\mathrm{i}}} \tag{1.1}$$

Two principal problems may arise from the adoption of the two above-mentioned techniques ([67, 68]). The first is related to computational costs: to compute the average error contribution (the neuron measure of resilience), it is required to perform both the forward propagation and back propagation for each instance of the training set. Second, the derivative of the cost function implies that the golden output must be available; in other words, the training set must be labelled. It means that the method can be used only with supervised learning neural networks.

A further contribution in this direction is given by Schorn *et al.* in [3] where the authors propose a methodology to assign resilience values to individual neurons. It is based on the deep Taylor decomposition of neural networks described in [69] which computes the contribution of each neuron to the output function value of a neural network. For each input, the Taylor decomposition and layerwise relevance propagation (LRP) algorithm computes the value $R_{i,j}$ for each neuron j belonging to the layer i, as described in [3]. This rule is used with the intent of calculating the average contribution of each neuron (with a score between 0 and 1) over a set of M training images. In more details, based on the training set, the resilience score r of each neuron $y_{i,j}$ is computed as follows:

$$ri,j = \frac{M}{\sum_{m=1}^{M-1} R_{\mathrm{i,j}}(y_{0,\mathrm{m}}, t_{\mathrm{m}})} \tag{1.2}$$

Where t is the output label vector related to the input image y_0. Similarly to [67, 68], this methodology requires the output labels to be available, and thus, it restricts the applicability of the technique to neural networks that are trained with a supervised learning procedure. Also concerning the computational costs, the back propagation phase must be repeated twice, first to compute the contribution of each neuron to the output function value with the Taylor decomposition and LRP ($R_{i,j}$). Next, to compute the contribution of each neuron to the output function value $r_{i,j}$.

Although the above-mentioned problems can be considered of relative importance, the real problem is that all these approaches can be classified as *network-oriented*: they do not consider the importance of neurons as entities linked to the single output classes. As it will be described in Section 3.2, *neurons that are critical for individual output classes may take a low resilience value in network-oriented approaches*. The different per-class information flow made by a few class-specific neurons with higher contribution is also addressed in [70] to reduce the size of convolutional neural networks.

1.2.4 Related Studies on Mitigation Strategies

Shorn *et al.* in [71] propose a methodology to predict the error resilience of neurons in DNNs and to map them in a reconfigurable neural network accelerator based on their criticality. They propose a Neural Compute Engine made of two areas: a

safe one composed of protected hardware elements and a normal region composed of unprotected hardware units. Neurons with the lowest resilience are assigned to protected processing elements (PPEs), and their intermediate and final computation results are given to protected local memory buffers or external memories. All other computations and their results are assigned to regular PEs, which are less power consuming and occupy less silicon area. PPEs are obtained by adding spatial or temporal redundancy, along with ECC. To demonstrate the validity of their proposal, they perform fault injections by exploiting the dropout fault model, in which a fraction of neuron outputs is set to zero. By means of their methodology, they are able to mitigate the effect of neuron crashes by keeping the accuracy degradation of the network comparatively small. A similar work was recently proposed by Hanif et al. in [72]. The authors described SalvageDNN, a fault-aware mapping methodology that permutes neurons and weights in a neural network such that the least critical weights are mapped to faulty PEs. In this way, they are bypassed by the Fault Aware Pruning (FAP) without impacting the accuracy of the DNN.

To improve ANN reliability, [52] propose an Algorithm-Based Fault Tolerance (ABFT) strategy to detect and correct single and multiple errors in matrix multiplication operations. The idea stems from the fact that the 67% of operations in their systems are matrix multiplication related. The framework is thus modified to call the ABFT-protected kernel at each matrix multiplication call. Experimental results demonstrate that the proposed ABFT can detect and correct about 60% of radiation-induced errors in Kepler GPUs and 50% in Pascal GPUs.

1.2.5 Software Test Library

The usage of Software Test Libraries (STLs) is a widely adopted solution for testing microcontrollers to perform periodic on-line tests during the system mission [73]. A Software Test Library (STL) is a set of software procedures or test programs executed by the processor core, and their main target is to test it, and eventually the surrounding peripherals. This methodology was initially proposed by [74]. The adoption of STLs for the on-line testing is a functional in-field self-test mechanism where exclusively functional stimuli are applied. This differs from structural ones, that often produce non-functional stimuli. Although being a functional test strategy, the STL effectiveness is evaluated via fault simulation. Therefore, it is possible to compute a fault coverage with respect to existing fault models (e.g., stuck-at faults).

Additionally, when knowing the workload (mission software) of the system, the fault coverage can be further increased removing the Safe Faults of the processor [75]. Safe faults do not cause any failure of the processor during its mission. However, it is worth saying that these faults are totally dependent on the application. Contrarily to hardware-based strategies, e.g., Logic BIST [76], Software Based Self Test (SBST) approaches do not require additional hardware. However, they need space in memory to be stored, and time for running, without ignoring the effort required for the development [77].

As detailed in [78], the test programs (or self-test routines) composing the STLs are developed following different approaches and can be distinguished in two main categories: the boot-time and the run-time. Boot time tests are executed during the boot operations before the mission application software is executed. These self-test routines have full access to processor and system resources. Furthermore, they do not have particular constraints on the execution time. On the other hand, the run-time test programs are periodically executed while the system is fully on-line with the mission application software already running. Unlike the boot-time self-test programs, they are developed taking into account different restrictive constraints [79]. Such constraints are intended for avoiding any interference of the STL mechanism with the mission software, as discussed in [79].

Chapter 2

Main Contributions

The main intent of this thesis is to investigate the reliability of modern embedded systems and to find out solutions to improve their safety. The major effort was put on those exploiting artificial intelligence and, specifically, predictive models such as Artificial Neural Networks.

In this field, the principal contributions of this thesis are:

- The investigation of the principal vulnerabilities and fault models existing in ANNs and AI-based devices.
- The proposal of reliability assessment methodologies and the release of specific tools to assist the analysis at different abstraction levels.
- The proposal of mitigation strategies to improve the safety and the reliability of AI-based devices.

Initially, in-depth studies have been carried out to understand their fault tolerance and to classify the different sources of error of such predictive models. Based on that, numerous fault injection campaigns have been performed at different abstraction levels to cover specific scenarios and to broaden the spectrum of analysis. Indeed, the robustness evaluation of ANN models and ANN-based systems can be pursued for various purposes and at different levels: from a software abstraction layer to a hardware-specific one, up to silicon measurements on Application-Specific Integrated Circuits (ASIC), Field Programmable Gate Arrays (FPGA), and Graphics Processing Units (GPUs) resorting to radiation campaigns. Clearly, depending on the adopted

methodology, parameters such as costs, precision, may considerably vary. As illustrated in Figure 2.1, this thesis presents reliability assessments methodologies at three levels of abstraction: at the software, at the hardware, and at the physical level.

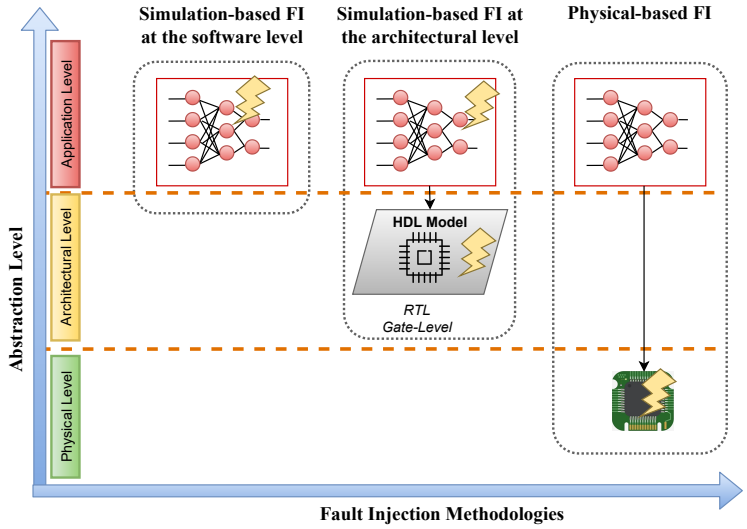


Figure 2.1 Fault injection approaches for assessing the reliability of ANNs and ANN-based systems at different abstraction levels. In each situation, the lightning bolt icon illustrates where errors or faults can be introduced.

As for the first one, two principal approaches are proposed to carry out reliability analysis on a neural network, without considering any hardware architecture running the model. They both rely on the adoption of software fault injector tools, which are able to simulate the occurrence of (i) hardware faults on weights/biases and (ii) errors on neurons. Specifically, a comprehensive analysis on different data types (e.g., floating-point numbers versus fixed-point numbers) was performed with the aim of finding out the best data type representations leading to the best trade-off between reliability and memory footprint. Compared to the state-of-the-art analyses (detailed in Section 1.2.3), a wider spectrum of floating- and fixed-point representations is given (five typologies ranging from 32 bits up to 8 bits). Next, a technique to rank neurons based on their criticality is presented. To the best of our knowledge, this is the first time that the importance of the neuron as related to the single output class is taken into account. In Chapter 3, the proposed approaches, the principal novelties as well as the experimental results are presented. It is worth to underline that the proposed technique differ from [72] in the way we assign criticality scores. While Hanif et al. consider only the *static* parameters (i.e., weights, bias, filters), in our work we consider also the contribution of the inputs (it is a *dynamic* approach).

Though resilience studies on neural network models are relevant for characterizing their major weaknesses and criticalities, they may not be comprehensive enough to accurately replicate actual behaviour when installed on hardware. It is recognized that simulations and theoretical investigations at the software level may be missing information about the underlying hardware platform. Therefore, this thesis proposes a reliability assessment technique at architectural level that considers both the ANN model and the final hardware device. A multi-level pipelined fault injector is proposed to run fault injection campaigns in simulation at the Register Transfer Level (RTL). Despite the higher costs in terms of RTL simulation time and computational resources, the advantage of conducting FIs at the architectural level is the following. Contrarily to the application/software level where faults can only be placed on the network weights, activators or input images, at a lower level, i.e., hardware, fault injections can be done anywhere in the system. This leads to a more faithful reliability assessment, where the designer could be able to co-shape the hardware platform with the DNN application, to pursue a desired reliability level. The novelty of the work is that, by mimicking the flow of the pipeline in processor cores, the proposed tool can drastically reduce the fault injection time by more than 60%. It will be detailed in Chapter 4. Finally, Chapter 5 presents a reliability assessment methodology at the physical level. This work introduces the architecture of a software emulator for reproducing the incidence of real faults retrieved from radiation campaigns on CNN-based applications. From the physical characterization of a DRAM memory (i.e., HyperRAM) under radiation tests, fault models and failure rates are extracted and used to configure the software emulator. The consistency between the actual radiation tests results and the software ones is established by using three different CNNs with different data representations (i.e., 32-bit floating point, 16-bit integer, 8-bit integer).

Based on the outcomes obtained through the different reliability assessment analysis, in this thesis two mitigation strategies are proposed. Described in Chapter 6, the first one builds on the above-mentioned study on critical neurons. An approach to evenly distribute critical neurons among the available processing element (PE) of a multiprocessor SoC (MPSoC) is described. It exploits Integer Linear Programming (ILP) to find out the optimal scheduling solution which is able to distribute critical elaborations (i.e., critical neurons) on the different computing resources (e.g., PEs). The results provide evidence that the proposed ILP scheduling can mask the effects of more faults and predict fewer wrong predictions. A reduction of 24.74% of

wrong predictions and an improvement of 97.80% and 59.53% of masked faults was obtained. Compared to [71], the proposed mitigation technique do not require additional hardware or spatial redundancy. The second work provides a comprehensive analysis of the use and the integration of Software Test Libraries (STLs) for the on-line testing of embedded systems running ANN-based applications. Specifically, a fault detection technique based on the adoption of an STL that must coexists with the requirements and the limitations of these resource-constrained microcontrollers is illustrated.

The contribution of the thesis in this topic is not limited to the mentioned studies, but they represent the most significant outcomes of my research activities. To conclude, the most prominent contributions are going to be briefly presented in the following chapters.

Chapter 3

Reliability Assessment at the Software Level

Analysing the reliability of artificial neural networks requires, foremost, shedding some light on two fundamental concepts. The main entities responsible for their proper functioning are nodes (i.e., artificial neurons) and connections (i.e., synaptic weights). Although highly interconnected, their role and their sensitivity differ within the neural network. For their major difference, we can classify them in two main categories:

- **Static Entities:** Weights and biases are fixed and do not depend on input stimuli. After the training phase, they are treated as constant data (read-only variables) and keep the value for all the inferences.
- **Dynamic Entities:** Neurons' outputs vary according to the inputs. They are considered as dynamic entities because their values depend on the current input stimuli.

To evaluate the trustworthiness and the reliability of such predictive models, in this thesis several studies have been done in both directions: first, a comprehensive analysis on weights and static data is presented in Section 3.1, and then, a study on critical neurons is outlined in Section 3.2.

3.1 Static Parameters in Artificial Neural Networks

Evaluating the sensitivity of ANN weights to the occurrence of hardware faults is crucial for the following reasons. Recent studies have shown that hardware faults due to silicon wear out and ageing effects, as well as those induced by an external perturbation (i.e., in a harsh environment) can significantly impact the inference leading to neural networks prediction failures [21, 59]. Moreover, when ANNs are deployed on hardware devices, being read-only variables, weights and static data are stored in memories, which are the highest contributor of soft errors in the system [80–82]. Being constant data, weights are never rewritten in the memory, and therefore, even for transient faults, once the fault is triggered, it behaves exactly like a permanent one since the flipped memory cell is normally not rewritten.

It is true that memories can be protected with Error Correction Code (ECC), but it also true that ECC comes at the cost of extra circuitry with power/performance overheads that may be too costly for an embedded application. Additionally, since the target applications are ANNs, it is claimed that they are intrinsic resilient to faults, and thus it justifies the absence of an error correction mechanism. Moreover, as the characterization based on radiation static/dynamic test shows [54], multiple errors occur, and for that, ECC is simply not effective.

This means that the assessment of the reliability of neural networks starts from the study of their static data, i.e., the weights and their data type representations.

A recent trend is to reduce the ANNs memory and energy footprint by leveraging on the adoption of reduced bit-width data types. Indeed, one important limitation about the usage of the newer version of ANNs is the memory required for storing the static parameters (e.g., weights). Approximate Computing (AxC) is considered as a good solution to deal with this complexity: by relaxing the need for fully precise or completely deterministic operations, AxC substantially improves energy efficiency and reduces the memory requirement. In the neural networks field, a common approximation approach is to reduce the precision and data type of weights and activation's values. Indeed, neural networks lend themselves well to AxC methods, especially with fixed-point arithmetic or low-precision floating-point implementations, which expose large fine-grain parallelism. If from one hand this is a sound solution to reduce the memory footprint of ANN-based applications, on the other hand, it might jeopardize their reliability. Therefore, it is necessary to investigate if those optimized

models are reliable enough to tolerate failures that propagate throughout the system. In other words, it starts to be crucial to evaluate their behaviour in a faulty scenario to determine if they can still be safely deployed on safety and mission-critical systems.

Two principal contributions are described in this section (3.1). First, the reliability assessment of two convolutional neural networks (CNNs) is done by targeting weights represented as 32-bit floating-point numbers. Next, the reliability of the same CNNs is addressed when a reduced bit-width data representation (i.e., fixed-point) is exploited to reduce the memory footprint and power consumption of the CNN application. In both cases, a software-based methodology is proposed to perform fault injection campaigns on weights and biases and evaluate their robustness against hardware faults.

3.1.1 Full Precision ANN Weights

3.1.1.1 Proposed Approach

The first reliability assessment study was performed on neural networks whose weights are represented with real numbers in a 32-bit floating point representation (Figure 3.1). In this work [83], a fault injection methodology at the software level is proposed to investigate the most sensible layers for which a safety mechanism may be purposely devised. Fault injections are carried out at the software level, with the aim of being totally independent of any potential architecture running the targeted neural networks. Specifically, permanent faults (stuck-at-0 and stuck-at-1) are randomly placed in weights according to the following proposed methodology.

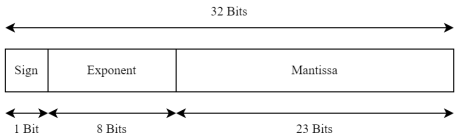


Figure 3.1 Single Precision IEEE 754 Floating-Point Standard

The FI procedure is described in algorithm 3.1: once the training of the neural network is completed, a golden run is performed to collect the golden results (i.e., golden_prediction), line 1 in 3.1. Then, the actual fault injection is done. The initial step requires generating the list of faults, which consists in a list of faulty locations (FLo) as described:

```
1 run_CNN (CNN, standard, golden_prediction_std);
2 run_CNN (CNN, custom, golden_prediction_cst);
3 for (i=0; i < FLo.size(); i++) {
4   inject_FLo (FLo[i], CNN);
5   run_CNN (CNN, custom, faulty_prediction);
6   compare (faulty_prediction, golden_prediction_cst);
7   compare (faulty_prediction, golden_prediction_std);
8   release_FLo (FLo[i], CNN);
9 }
```

Listing 3.1 Fault Injection Procedure (pseudo-code)

$$FLo = < Layer, Connection, Bit, Polarity > \tag{3.1}$$

Layer corresponds to the CNN's layer, *Connection* is the edge connecting one node of the *Layer* (i.e., the weight), and *Bit* is one of the bits of the weight associated to the *Connection*. Finally, *Polarity* can be '0' or '1' depending on the stuck-at fault. In other words, the injections are performed by randomly selecting the faulty bit among all bits of the connection weights. For any fault in the fault list (line 2 in 3.1), a prediction run is carried out, and the results are collected (i.e., faulty_prediction). In the end (line 5 in 3.1), the faulty_predictions are compared with the expected ones, and the results logged for a later analysis.

More in details, the function *compare* of the algorithm (3.1) classifies the prediction of the faulty CNN with respect to the golden one. Depending on the effect, faults are classified as follows:

- **Masked:** No difference is observed from the faulty CNN and the golden one.
- **Observed:** A difference is observed from the faulty CNN and the golden one. Depending on how much the results diverge, we further classify these as:
  - **Safe:** the confidence score of the top ranked element varies by less than +/-5% w.r.t. the golden one;
  - **Unsafe:** the confidence score of the top ranked element varies by more than +/-5% w.r.t. the golden one, or the top ranked element predicted by the faulty CNN is different from that predicted by the golden one. As already discussed in [43], this is the most critical observed fault;

Table 3.1 LeNet [4]

Layer	Type	Weights	Bit Width	#Faults	#Injections
0	Conv	2,400	32	153,600	9,039
1	Conv	51,200	32	3,276,800	9,576
2	FC	3,211,264	32	205,520,896	9,604
3	FC	10,240	32	655,360	9,465

Additionally, this work identifies the "Safe Faults Application Dependent" (SFAD) accordingly to the ISO 26262 standard. SFAD faults can not produce any failure in the operational mode and therefore can be removed from the fault list [84]. Their identification is therefore crucial to focus the test efforts towards faults leading to application failures, only. It can be simply computed as the union between Masked and Safe-Observed fault, as shown in (3.2).

$$SFAD = Masked \cup Safe\_Observed\_Fault \tag{3.2}$$

The total number of injected faults have been computed by using the statistical FI approach presented in [85]. In details, we resorted to the following formula:

$$fault\_injections = \frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 \cdot 0.25}} \tag{3.3}$$

where N is the total number of faulty locations, e is the desired error margin (1%), and t depends on the desired confidence level (t=2.58 corresponds to 99% confidence level [85]). Equation 3.3 has a horizontal asymptotic value ($N \to \infty$), thus limiting the number of fault injections necessary to achieve an evaluation with an error margin of 1% and a confidence level of 99%. Running exhaustive fault injections would be unfeasible due to the complexity and the number of faulty locations which can dramatically explode.

In this work, two convolutional neural networks are used as case study: LeNet and Tiny YOLO. The former is a classifier, probably the most popular convolutional neural network developed for the image recognition of handwritten digits [4]. The latter is a CNN developed for detecting objects in real time [5], up to 45 frames per second.

Table 3.2 Tiny YOLO [5]

Layer	Type	Weights	Bit Width	#Faults	#Injections
0	Conv	432	32	27,648	7,128
1	Conv	4,608	32	294,912	9,301
2	Conv	73,728	32	4,718,592	9,584
3	Conv	294,912	32	18,874,368	9,599
4	Conv	1,179,648	32	75,497,472	9,603
5	Conv	4,718,592	32	301,989,888	9,604
6	Conv	262,144	32	16,777,216	9,599
7	Conv	1,179,648	32	75,497,472	9,603
8	Conv	130,560	32	8,355,840	9,593
9	Conv	32,768	32	2,097,152	9,560
10	FC	884,736	32	56,623,104	9,602
11	FC	65,280	32	4,177,920	9,582

Details about the topologies of the CNNs as well as fault injection figures are given in Tables 3.1 and 3.2. The first and the second columns report the number and the type of the layers, i.e., Convolution (*Conv*) or Fully Connected (*FC*), respectively. The 3rd column indicates the number of connecting weights. Next, column 5th indicates the total number of possible faulty locations: it is given as the multiplication between the number of connections (column "*weights*") and the bit width times two (stuck-at-0 and stuck-at-1). As emerging, the overall number of faulty locations is very high, reflected in a non-manageable fault injection campaign execution time. To reduce the FI execution time, we randomly select a subset of faults by following [85]. To obtain statistically significant results with an error margin of 1% and a confidence level of 95%, an average of 9k FIs have been considered. The exact numbers are given in the last column (#*Injections*) of Tables 3.1 and 3.2.

3.1.1.2 Experimental Results

The fault injection framework was build on the *darknet* open source DNN framework [34] implemented in C language. The first neural network under assessment was LeNet. For the analysis, pre-trained weights available from [34] have been used. For the injection campaign, we randomly selected 37 validation images from the MNIST database.

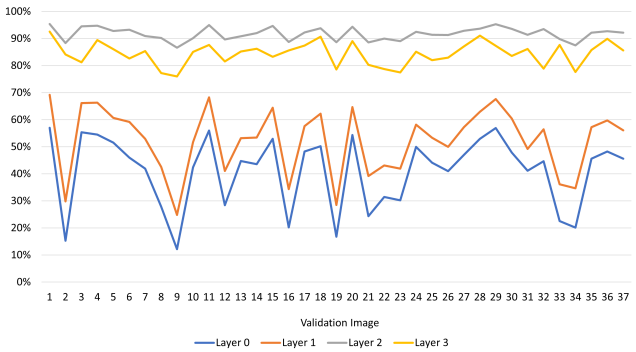


Figure 3.2 LeNet Masked Faults

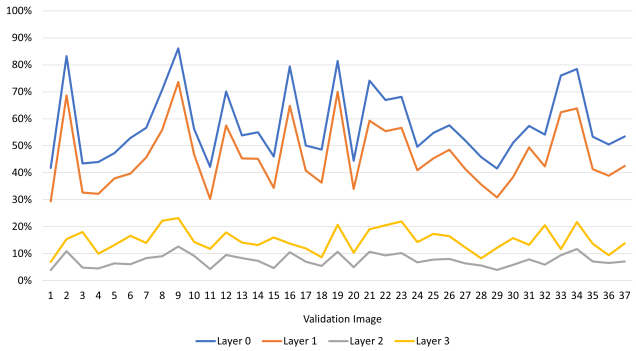


Figure 3.3 LeNet Safe Observed Faults

Figures 3.2, 3.3 and 3.4 illustrate the obtained results in terms of percentage of Masked, Safe Observed and Unsafe Observed. For each chart, we differentiate the injections per layer (from 0 to 3). As shown, the percentage of Masked faults is higher for *Layer*2 and *Layer*3 (i.e., the fully connected ones), while the first two layers show a lower percentage of Masked faults. A similar analysis can be done for Unsafe Observed faults. Fully connected layers show the lowest percentage of Unsafe Observed faults. This means that the less *critical* layers are the fully connected ones (for the LeNet topology).

Concerning the Unsafe Observed faults (Figure 3.4), we would like to stress the fact that their percentage is very low, varying from 0.4% up to 1.8%. Moreover, when considering *Layer*0, the two drops of Unsafe Observed faults percentage (corresponding to workload 17 and 23) only represent a difference of 1.4% meaning that we can consider this variation as negligible. This means that we cannot notice a particular dependency on the input workload. This result is quite interesting because it is showing a different trend with respect to the effect of soft errors. Indeed,

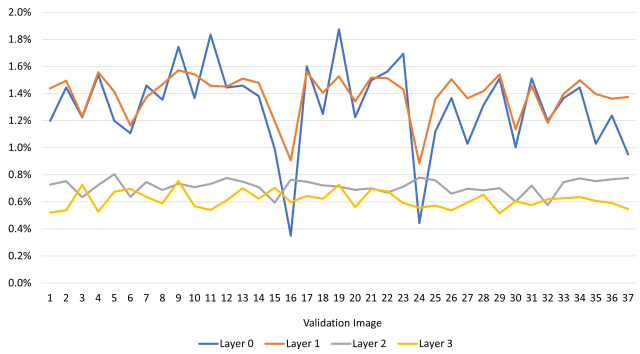
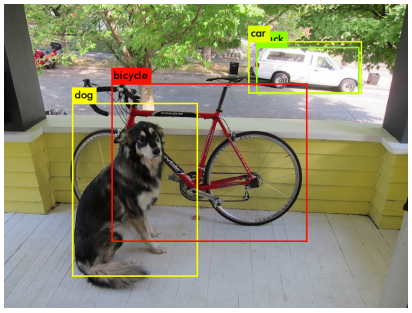


Figure 3.4 LeNet Unsafe Observed Faults

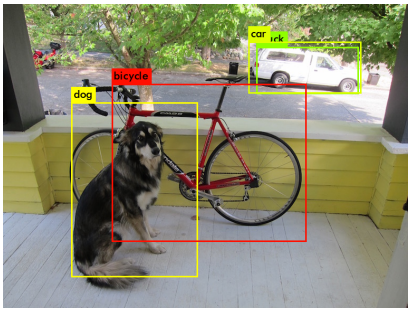
convolutional layers (i.e., the first two layers for LeNet) are supposed to be the more resilient to the presence of a fault, according to results shown in [43]. This is due to the fact that their role is to extract the features from the source image, while the full connected layers are supposed to be the less resilient because they classify the features extracted by the first two levels. On the other hand, these results seem to confirm the conclusion of [52] in which the authors claim this trend. However, in [43] the result presented another trend: a slightly higher resilience was found for the convolutional layer. This can be explained by two factors: first the fault model, here we experimented permanent faults while in [43] the fault model was transient, and the network topology.

The last experiment carried out on LeNet is the analysis of the most **critical bits** of the weights. As previously mentioned, weights are represented as single-precision binary floating-point format, as described by the IEEE 754 standard (Figure 3.1). It turned out that *all* the Unsafe Observed faults have been due to faults affecting the 8 bits used for storing the exponent (i.e., from bit 30 down to bit 23). The sign and the mantissa bits do not have significant impacts (i.e., they led either to Masked or Safe Observed faults).

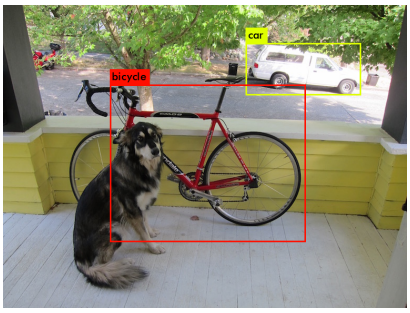
To perform FI campaigns with YOLO, the pre-trained weights available from [34] have been used (*yolov3-tiny.weights*). The workloads have been downloaded from the same repository and consist of seven different images. In this second case study, the same fault classification was used, with only one exception. LeNet classifies the input image as one precise digit. Therefore, only the highest score is considered. Using YOLO, we may have more than one object that can be detected in



(a) Golden Prediction



(b) Safe Observed Faults



(c) Unsafe Observed Faults

Figure 3.5 Example of YOLO Predictions.

an image: for example, we may have several cars in a single picture. In line with this, the definition of Unsafe Observed faults was updated as follows:

1. The number of detected objects is different from the golden and faulty prediction;
2. The number of detected objects is equal, but the label associated with them are different (i.e., wrong detection);
3. The *location* of the top ranked element varies by more than +/-5% with respect to the golden one.

The other definitions (Masked Faults and Safe Observed Faults) stay the same. For the sake of clarity, let us resort to an example, depicted in Figure 3.5.

In the case of Safe Observed faults (Figure 3.5b), the faulty YOLO is able to correctly detect the four objects, but their *locations* are slightly different (less than 5%) with respect to the golden ones. The term *location* means the rectangle identifying the object in the picture. Conversely, Figure 3.5c presents Unsafe Observed faults.

In this picture, YOLO only recognizes two objects (the bike and the car) and, overall, we can say that the prediction is wrong.

Figures 3.6 and 3.7 report the obtained results in terms of percentage of Masked and Unsafe Observed faults. Contrarily to the results obtained with Lenet, we do not have a significant percentage of Safe Observed faults (i.e., the obtained percentage was less than 0.05% on average). As it can be observed, we have a different population of Masked faults compared to the LeNet one. First, the distribution of Masked faults do not vary so much among the layers, except for the first one. As a consequence, we can not claim that fully connected layers are the least critical, since also most of the convolutional layers show a high percentage of Masked faults. On the other hand, it is also interesting to note that depending on the workload (i.e., the input image as plotted on the x-axis), the distribution of Masked faults significantly changes, differently from Lenet.

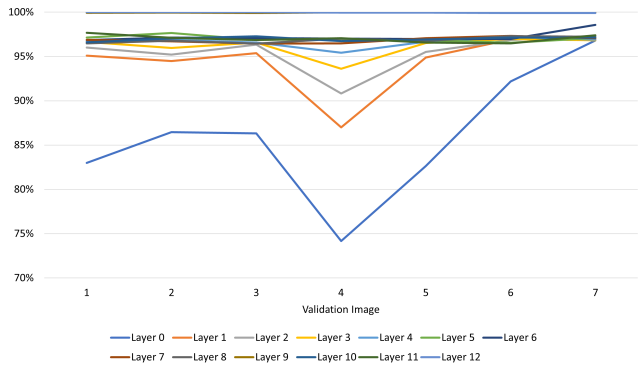


Figure 3.6 Tiny YOLO Masked Faults

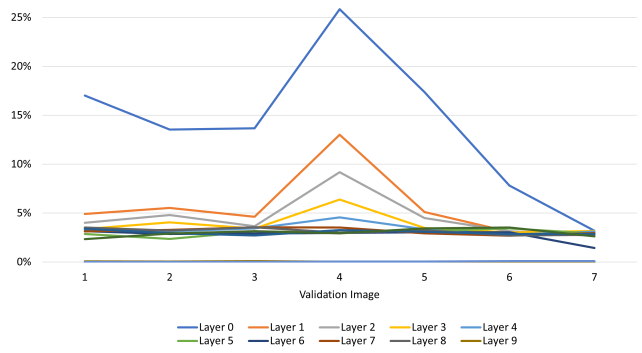


Figure 3.7 Tiny YOLO Unsafe Observed Faults

A final comment is related to the image number 4 for which we can notice a peak of Unsafe Observed faults for all layers, but especially for the *Layer*0. As for LeNet,

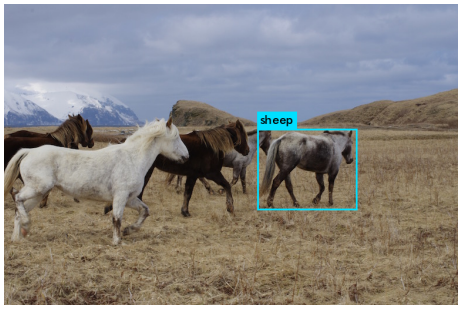


Figure 3.8 YOLO Workload #4.

*Layer*0 is the most sensible layer to the input workload. However, in this case, the variation can not be neglected since an increase of 10% of Unsafe Observed faults is observed. The prediction of the input workload number 4 is illustrated in Figure 3.8. As evident, instead of detecting the horses, the faulty YOLO detects one sheep. For this specific image, probably for its graphic complexity, faults in *Layer*0 lead to a great quantity of Unsafe Observed faults. This means that, depending on the network topology, the input workload can play a significant role.

This outcome proves that it is not possible to generalize. Each CNN has to be analysed in order to identify the most critical elements (e.g., layers). Despite the difference between the achieved results with the two neural networks, the analysis of the most critical bits for the YOLO weights confirms the theory: all the Unsafe Observed faults are due to stuck-at faults affecting the 8 bits used for storing the exponent (i.e., from bit 30 down to bit 23). The sign and the mantissa bits do not have significant impacts (i.e., they led either to Masked or Safe Observed faults).

3.1.2 Reduced Precision ANN Weights

In Section 3.1.1, reliability studies are presented on neural networks exploiting a 32-bit floating-point data type to represent their parameters (weights and biases). In this section, the reliability is investigated when reduced bit-width data types are used to represent the parameters of neural networks, and therefore when compressed DNN models are exploited to reduce the memory footprint of the application as well as its power consumption.

3.1.2.1 Proposed Approach

The intent of this work [2] is to characterize the impact of permanent faults affecting a CNN, when reduced bit-width data types are used for representing its parameters. Therefore, different implementations of the same CNN architecture have been analysed through software FI campaigns. Two case studies are presented: the former targets LeNet-5 [4], a well known classifier for handwritten digit recognition, the latter focuses on YOLO [5], a DNN used for detecting objects in real time.

In the neural networks field, a common approximation approach is to reduce the precision and data type of weights and activation's values. In this research work, we used custom floating-point and fixed-point representations with different precision (i.e., bit-width) at the inference time. As detailed in Section 3.1.1, the *darknet* open source framework [34] was used. Indeed, it supplies a very simple environment where several configurations of DNNs, including CNNs, can be executed either to perform training or inference jobs. In our work, we modified the *darknet* framework to approximate the neural network under assessment and inject stuck-at faults at the inference time. The *darknet* framework leverages on 32-bit floating-point data types, only. To allow data type conversions, the *darknet* source code was modified. All the conversions between the standard 32-bit floating-point and **custom data type** have been carried out by integrating two open source libraries: the *libfixmath* library [86] for managing fixed-point numbers and the *FloatX* library.

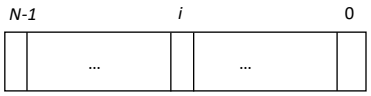


Figure 3.9 Custom Data Type.

Figure 3.9 depicts our custom data type. It is defined as follows:

- N: It determines the data bit-width;
- i: It determines the dynamics and the precision of the data type depending on the data representation:
  - Floating Point (FP): i is the mantissa width, $N-1-i$ is the exponent width;
  - Fixed Point (FxP): i is the fractional width, $N-1-i$ is the integer width.

Since the end-goal is to characterize fault effect propagation through the network (speeding up computations and compacting the model size are out of the scope of this work), we performed on-line conversions while maintaining all internal operations in floating-point (Figure 3.10). The benefits coming from this approach are two-fold: first, it is not necessary to change the framework structure every time new experiments with a different data type have to be performed; second, it allows changing the representation without retraining the DNN model for each data type, exploiting the same set of trained parameters. In this way, the CNN reliability assessment is quicker; it is possible to switch between experiments with different numerical formats in a reasonable amount of time.

To evaluate the effectiveness of the method, a preliminary experiment was performed to confirm that the online conversion does not introduce important accuracy differences while providing execution time benefits. To this end, the data representation of the darknet framework was converted from 32-bit floating point to 32-bit fixed point. In this way, all the operations were performed in the fixed-point arithmetic domain by using the *libfixmath* library [86]. The inference of 70,000 images from the MNIST database was performed with the LeNet-5 in both versions, i.e., the original floating-point and the modified fixed-point. Specifically, the neural network was not retrained. The experimental results showed that the average accuracy error of the fixed-point version with respect to the floating-point one was of -0.01%, i.e., a slight accuracy decrease. Moreover, while the execution time of the fixed-point version experiment was 8,566 seconds, i.e., ≈ 0.122 seconds per image, the execution time of the original floating-point version was 3,280 seconds, i.e., ≈ 0.047 seconds per image. Therefore, the use of the *darknet* 32-bit floating-point framework with on-line conversion allows us to run 2.6x faster experiments with respect to converting the darknet data type, while not incurring in significant inference accuracy differences.

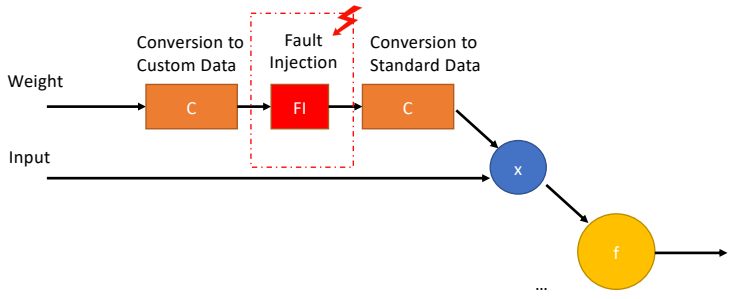


Figure 3.10 On-line Weights Conversions.

For the sake of completeness, we describe how the on-line conversion of a 32-bit floating-point weight is applied before reaching a single neuron (Figure 3.10). The applied scheme works in the following way:

1. The weight is converted from the standard 32-bit floating-point representation to a given custom data type one.
2. The custom data type weight is corrupted according to a chosen fault list and fault location, i.e., the fault is injected.
3. The custom data type weight is converted back to the standard 32-bit floating-point representation in order to preserve the native implementation of the framework. In such a way, the gained value reflects the same fixed-point corrupted value, while still remaining a floating-point data.
4. The weight is multiplied by the input value.
5. The neuron performs the arithmetic computations.

Although all the operations are executed between 32-bit floating-point variables, it should be outlined that the loss of precision caused by the first conversion is preserved. Indeed, when moving from the standard 32-bit floating-point representation to a low-precision one (e.g., 16-bit fixed-point), we are witnessing a truncation error effect. Then, converting back from a narrow range of value to a wider one, the truncation error still remains.

Figure 3.11 illustrates the FI setup and configuration. First, the neural network under assessment was trained with 32-bit floating-point data types: it is referred to as **Standard**. Then, this model is approximated by using a custom data type representation: the obtained DNN model is referred to as **Custom**. The outputs of the inferences are stored (i.e., the *GoldenStandard* and the *GoldenCustom*), and compared to determine the **Accuracy Loss** due to the approximation. The FI campaign is carried out on the Custom CNN and the faulty inference outputs are stored in the *Faulty Custom* log. The latter is then compared with the *Golden Custom* and the *Golden Standard* to assess the resilience. The Custom CNN (i.e., approximated model) is intended to replace the Standard CNN in edge/resource-limited devices. In this work, two different experiments have been conducted: first, the reliability of the Custom CNN is assessed with respect to the Standard version.

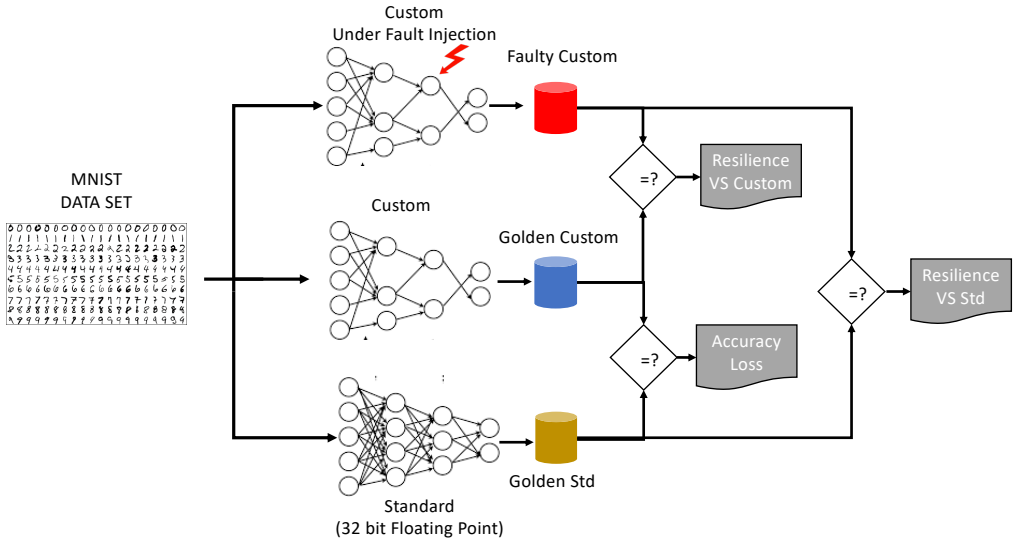


Figure 3.11 Fault Injection Scenario [2].

Next, the custom-data-type CNN was trained, and the Faulty Custom was directly compared with the Custom CNN itself.

The FI procedure follows exactly the same methodology described in 3.1, with a minor change regarding the fault injection procedure. Faults are injected regardless of their polarity (stuck-at-0 or stuck-at-1). Once the fault location is fixed, the target bit is inverted (if '0' it becomes a '1' and vice-versa) as a bit-flip. In this way, we do not distinguish between the singular effect of the two fault models while obtaining great flexibility for the considerable amount of performed simulations. Additionally, a slight difference is introduced in this work to fine tune the faulty predictions. We consider two types of CNNs: (i) a classifier and (ii) an object detector. Concerning the classifier, outputs of the faulty CNN are labelled as follows:

- **Masked:** No difference is observed between the faulty CNN and the golden one.
- **Observed:** A difference is observed between the faulty CNN and the golden one. Depending on how much the results diverge, we further classify these as:
  - **Good:** The confidence score of the top-ranked is higher with respect to the golden CNN. In other words, the faulty CNN provides a better inference than the golden one;

  - **Accept:** The confidence score of the top-ranked element is reduced by less than 5% with respect to the golden CNN;
  - **Warning:** The confidence score of the top-ranked element is reduced by more than 5% with respect to the golden CNN;
  - **Critical:** The top-1 prediction is different. In other words, the faulty CNN makes a wrong inference.

From a safety assessment perspective, we consider three classes of faults (*Critical*, *Warning*, and *Accept*) as events **reducing the CNN safety**. Indeed, whenever one of those fault classes occurs, either the top-1 prediction is different (*Critical*) or the top-1 prediction confidence level decreases (*Warning*, *Accept*). On the other hand, the two fault classes *Masked* and *Good* either **leave the safety of the CNN unaltered** (Masked) **or even improve it** (Good).

On the other hand, the effects of the injected faults on the object detector CNN are classified differently. The CNN output is an image having bounding boxes indicating the detection of the objects. To assess whether any two bounding boxes overlap, we use the intersection over union (IoU) metric as defined in [59]. IoU is the ratio of the intersection area over the union area of two bounding boxes. The closer IoU is to 1, the higher overlap the two bounding boxes have. Thanks to the IoU metric, we can redefine the faulty outcome as follows:

- **Masked:** No difference is observed between the faulty CNN and the golden one.
- **Observed:** A difference is observed between the faulty CNN and the golden one. By using the IoU calculated between the boxes of the golden CNN and those of the faulty one, we further classify the observed outcomes as follows:
  - **Accept:** The IoU is lower than 1 and higher than 0.95;
  - **Warning:** The IoU is lower than 0.95 and higher than 0.9;
  - **Critical:** The number of bounding boxes is different, or the label associated with the boxes does not match the good ones. In other words, the faulty CNN identified the wrong objects. Moreover, if the IoU is lower than 0.9, the fault is classified as critical, meaning that the faulty CNN correctly identifies the objects, but it is not able to locate them precisely enough.

The *Good* outcome is not considered, since it does not make sense for object detection tasks. It is worth noting that the fault classification used for the object detector is more stringent than the one used for the classifier. Indeed, we consider the object detection task more critical than the classification one. From a safety assessment perspective, we consider faults falling in the *Critical*, *Warning*, and *Accept* classes as events reducing the CNN safety. Conversely, *Masked* faults leave the safety of the CNN unaltered.

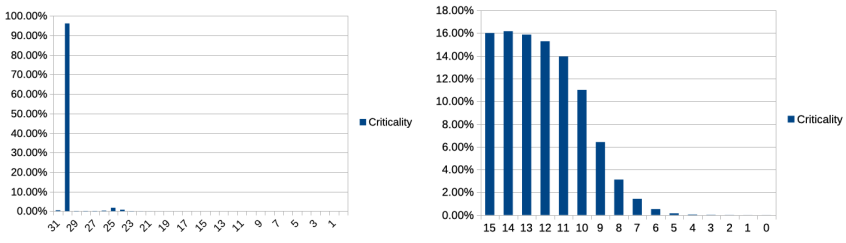
3.1.2.2 Experimental Results

LeNet-5

The first case study targets LeNet-5, a well-known classifier for handwritten digit recognition tasks introduced by Y. Lecun et al. in 1998. For the analysis, pre-trained weights available in a 32-bit floating-point representation from [34] have been used. For the injection campaign, a workload of 2,023 images was randomly selected from the MNIST test/validation dataset.

In our preliminary work [87], different fixed-point data types have been analysed. One of those was configured with $N = 16$ and $i = 8$, meaning that 8 bits were devoted to represent the fractional part and 8 bits the integer one. Figures 3.12a and 3.12b illustrate the criticality of each bit of the data type. As emerging, in the 32-bit floating-point representation, only one bit (i.e., the bit 30^{th}) is the main responsible for the Critical Observed faulty behaviours: up to 95% of critical observed faulty behaviours is due to a fault at bit 30^{th}. On the other hand, in the 16-bit fixed-point representation (with 8 bits for integers and 8 bits for the fractional part), the number of bits responsible for the Critical Observed faulty behaviours is six. This means that the custom CNN has a lower memory footprint of 50%, but it also shows a lower resilience with respect to the standard CNN, since a higher number of faulty bits may seriously affect the results of the inferences.

To carefully select the custom data representations, we analysed the LeNet-5's distribution of weights, which is shown in Figure 3.13, and evidences that all values are in the range -0.6 to 0.6 with most of them around zero. Based on this, we deduced that the data type does not need higher dynamics while a high precision is preferred. Hence, we selected the custom data types reported in Table 3.3.



(a) Standard 32-bit Floating Point (b) Custom 16-bit Fixed Point

Figure 3.12 Critical Bits

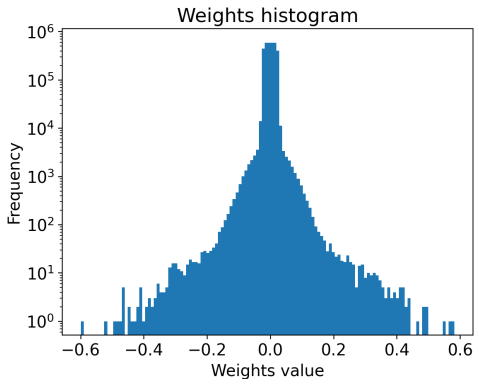


Figure 3.13 LeNet-5 pre-trained weights distribution.

Two data types are used, the fixed and floating point (FxP and FP, respectively) with different bit width. Moreover, we computed the accuracy loss of the CNN resulting from the adoption of custom data type weights. As highlighted in Table 3.3, five different scenarios have been studied. The 2nd column of the Table reports the data type used in each FI campaign, while the 3rd column reports the bit-width of the weights. The 4th column shows the number of bits allocated to encode the different parts of the number, i.e., sign, exponent, and fractional parts (to represent floating-point numbers), and the integer and fractional part (for fixed-point data). To compute the accuracy of the CNN in the different scenarios, the inference of the images belonging to the validation set of the MNIST database (10,000 images) has been run on LeNet-5, clearly without injecting any faults, i.e., in a golden scenario. The results show that only when the bit width is reduced to 8, the neural network exhibits some accuracy loss. In detail, for the network with weights encoded by using 8-bit floating-point variables (scenario FP8), the accuracy loss is 0.02%, while it is 0.04% when the weights were encoded by using 8-bit fixed-point variables (scenario FxP8).

Table 3.3 LeNet-5 Data Type Accuracy Loss [%]

Scenario	Data type	Bit-width	Bit encoding	[%] Accuracy Loss
FP32	floating-point	32	1 sign, 8 exponent, 23 fractional	Ref.
FP16	floating-point	16	1 sign, 5 exponent, 10 fractional	0%
FP8	floating-point	8	1 sign, 4 exponent, 3 fractional	0.02%
FxP32	fixed-point	32	1 integer, 31 fractional	0%
FxP16	fixed-point	16	1 integer, 15 fractional	0%
FxP8	fixed-point	8	1 integer, 7 fractional	0.04%

Table 3.4 LeNet-5 Fault List for Fault Injection Campaigns

	Layer	L0	L2	L4	L6
	Detail	Convolutional	Convolutional	Fully Connected	Fully Connected
	Connections	2,400	51,200	3,211,264	10,240
Scenarios FP32, FxP32	Bit-width	32	32	32	32
	#Faults	76,800	1,638,400	102,760,448	327,680
	#Injections	13,678	16,474	16,638	15,837
Scenarios FP16, FxP16	Bit-width	16	16	16	16
	#Faults	38,400	819,200	51,380,224	163,840
	#Injections	11,610	16,310	16,636	15,107
Scenarios FP8, FxP8	Bit-width	8	8	8	8
	#Faults	19,200	409,600	25,690,112	8,1920
	#Injections	8,915	15,991	16,630	13,831

Similarly to the reliability study discussed in Section 3.1, a statistically meaningful subset of FIs was performed, according to the mathematical formula proposed in [85], equation 3.3. Table 3.4 provides details about the FI setup, the configuration, and the fault lists of each LeNet-5's layer. The first two rows (labelled *Layer* and *Detail*) present the target layers; the third one (*Connections*) specifies the quantity of their connection weights. The number of possible faults is computed as the multiplication between the connections number (*Connections*) and the weight size (*Bit-width*). As the rows *#Faults* point out, the overall number of possible faults is very high, and this reflects in a non-manageable FI campaign execution time. Therefore, to reduce the execution time, a subset of faults is considered. To obtain statistically significant results with an error margin of 1% and a confidence level of 99%, an average of 15.6k FIs have to be considered for the 32-bit scenarios (FP32 and FxP32), 15k for the 16-bit scenarios (FP16 and FxP16), and 13.8k for the 8-bit scenarios (FP8 and FxP8). The exact numbers are given in the rows of Table 3.4 labelled *#Injections*, and they have been computed by using the approach presented

in [85].

We conducted two sets of experiments (see Figure 3.11). In the first one, we evaluated the reliability by using as reference the **Standard** 32-bit floating-point CNN. This is useful to approximate the CNN (i.e., to change its data type and/or bit-width) after that it has been trained. In the second one, we assess the CNN reliability by using as reference the **Custom** fault-free CNN. This is useful to directly train the custom data-type/bit-width CNN. The safety of the different CNN versions is evaluated. Therefore, faults in the classes *Critical*, *Warning*, and *Accept* are considered as events reducing the CNN safety. The sum of these contributions is represented by the symbol '$<$' in Tables 3.5 and 3.6. Conversely, we consider the faults in the classes *Masked* and *Good* as events, either leaving the safety of the CNN unaltered or even improving it. The sum of these contributions is represented by the symbol '$\geq$' in Tables 3.5 and 3.6. The comprehensive results of the first set of experiments (i.e., having the Standard 32-bit floating-point CNN as a reference) are shown in Table 3.5 where each row corresponds to one of the CNN variants (FP32, FP16, FP8, FxP32, FxP16, FxP8 defined in Table 3.3). Each column corresponds to a faulty behaviour class, as described above. First, we can note a different resilience to faults depending on the data type: floating *versus* fixed. More in detail, **the safety decreasing effect is lower for the fixed-point than for the floating-points**, for a given bit-width. As an example, we may resort to scenarios FP32 and FxP32 (32-bit CNNs): the *safety increasing (decreasing)* effect varies from 69% (31%) of the floating-point version (scenario FP32) to 74% (26%) of the fixed-point version (scenario FxP8). This corresponds to a difference of 5%. The average difference between floating- and fixed-point versions with respect to safety increasing/decreasing effect over the three variants (32, 16, and 8 bits) is 8.96% over all the layers. This can be seen by comparing the scenarios FP32 with FxP32, FP16 with FxP16, and FP8 with FxP8, in terms of the average safety increase/decrease effect variation (columns 8 and 9). Is it worth highlighting that, in general terms, the safety decreasing effect is critical only in a few cases. The percentage of critical faults is always lower than 3.42% for all the variants. In particular, fixed-point variants have a very small percentage of critical faults, always lower than 0.46%. Moreover, the contribution of *Good* faults to the safety increasing effect turns out to be significant, especially for 16- and 8-bit versions. As an example, in the scenario

FxP8 for the layer L0, we observed a safety increasing effect in 52.21% of the cases, with a 52.18% of *Good* faults.

Furthermore, the bit-width plays an important role for the reliability: **the lower the bit-width, the lower the resilience**. Therefore, a designer who wanted to use a more efficient version of the CNN (reduced memory footprint) has to be aware that it would be also less resilient with respect to the original CNN (FP32). However, it is worth also remarking that using fixed-point data representation, instead of the floating-point counterpart, provides the better results in terms of trade-off between resilience and efficiency. This is reported in the last two columns of Table 3.5. For instance, we may compare scenarios FP8 and FxP8 (8-bit CNNs) for layer L0: we observed a safety loss with respect to FP32 of 37% in the floating-point version (scenario FP8) and only of 16% in the fixed-point version (scenario FxP8). Therefore, choosing the CNN in the scenario FxP8, namely 8-bit fixed-point (1 bit for integer and 7 bits for fractional), allows the designer to compact the memory footprint by a 4x factor while reducing the safety only by 16%. Moreover, by looking more closely, the occurrence of critical faults in scenario FxP8 even decreases from 1.32% of FP32 to 0.45%, while in scenario B it increases to 3.41%. Additionally, for scenario FxP32 (32-bit fixed-point CNN), it has been observed that the CNN achieves improved safety with respect to the FP32 scenario, for the same memory footprint for layers L0 and L2 (+5.34% and +1.43%, respectively). Thus, simply changing the CNN data type to a fixed-point representation may improve its resilience for some layers.

Table 3.6 reports the complete results of the second set of experiments (i.e., having the Custom CNN as a reference). While in the first set we compared the FI results of each scenario to the ones obtained with the fault-free 32-bit floating-point CNN (FP32), in this set of experiments *we compare the results of each scenario to the results obtained with the corresponding fault-free custom CNN*. This scenario corresponds to directly training the custom-data-type CNN, so the reliability has to be assessed with respect to the Custom CNN itself. For details, see Figure 3.11. Note that the row related to the FP32 version is the same one in both experiment sets.

In general, the trends highlighted in the first set of experiments are observed also in this scenario:

1. The safety is impacted by the bit-width reduction;

Table 3.5 LeNet-5 Fault Injection outcomes with respect to the **Golden Standard** version.

Layer	Data	Observed				Masked	Safety		Gain w.r.t. FP32	
		Critical	Warning	Accept	Good		$<$	$\geq$	Safety*	Memory
L0	FP32	1.32%	0.06%	29.96%	28.98%	39.68%	31.34%	68.66%	-	-
	FP16	2.61%	0.12%	53.28%	41.27%	2.71%	56.01%	43.98%	-24.67%	2X
	FP8	3.41%	0.91%	64.13%	31.53%	0.02%	68.45%	31.55%	-37.11%	4X
	FxP32	0.03%	0.04%	25.93%	25.54%	48.47%	26.00%	74.01%	+5.34%	0
	FxP16	0.05%	0.08%	49.98%	46.94%	2.96%	50.11%	49.90%	-18.77%	2X
	FxP8	0.45%	0.60%	46.74%	52.18%	0.03%	47.79%	52.21%	-16.45%	4X
L2	FP32	1.37%	0.02%	23.88%	23.39%	51.34%	25.27%	74.73%	-	-
	FP16	2.59%	0.08%	55.00%	38.57%	3.76%	57.67%	42.33%	-32.40%	2X
	FP8	1.10%	0.91%	65.86%	32.10%	0.03%	67.87%	32.13%	-42.60%	4X
	FxP32	0.01%	0.01%	23.82%	23.62%	52.54%	23.84%	76.16%	+1.43%	0
	FxP16	0.03%	0.02%	49.87%	46.61%	3.47%	49.92%	50.08%	-24.65%	2X
	FxP8	0.42%	0.45%	46.57%	52.53%	0.03%	47.44%	52.56%	-22.17%	4X
L4	FP32	0.71%	0.00%	3.85%	3.86%	91.57%	4.56%	95.44%	-	-
	FP16	0.84%	0.13%	57.79%	36.12%	5.13%	58.75%	41.25%	-54.19%	2X
	FP8	0.49%	0.06%	66.69%	32.72%	0.04%	67.24%	32.76%	-62.67%	4X
	FxP32	0.00%	0.00%	10.99%	11.49%	77.52%	10.99%	89.01%	-6.43%	0
	FxP16	0.00%	0.00%	49.40%	45.59%	5.01%	49.40%	50.60%	-44.84%	2X
	FxP8	0.40%	0.36%	46.38%	52.82%	0.04%	47.14%	52.86%	-42.58%	4X
L6	FP32	0.61%	0.01%	7.69%	8.14%	83.54%	8.32%	91.68%	-	-
	FP16	1.19%	0.03%	56.34%	37.65%	4.78%	57.57%	42.43%	-49.25%	2X
	FP8	1.76%	0.40%	65.07%	32.74%	0.04%	67.22%	32.78%	-58.91%	4X
	FxP32	0.01%	0.04%	13.50%	14.11%	72.33%	13.55%	86.45%	-5.24%	0
	FxP16	0.03%	0.08%	49.15%	46.11%	4.63%	49.26%	50.74%	-40.94%	2X
	FxP8	0.44%	0.53%	46.28%	52.72%	0.04%	47.25%	52.75%	-38.93%	4X

* Safety increasing effect difference between a given scenario and FP32

2. For fixed-point CNN versions, we observed a more graceful safety decrease compared to floating-point ones.

Besides that, an interesting effect is the following: for floating-point custom variants (FP16 and FP8) the difference between the two sets of experiments (Tables 3.5 and 3.6) in terms of Safety gain with respect to FP32 is higher than for fixed-point ones (FxP32, FxP16, and FxP8). For example, in layer L0, for variant FP8 (8-bit floating-point) the Safety gain with respect to FP32 is -37.1% for the first set of experiments (Table 3.5) and -19.9% for the second set of experiments (Table 3.6), that is a 17.2% difference. Conversely, for variant FxP8 (8-bit fixed-point) the Safety gain with respect to FP32 is -16.4% for the first set of experiments (Table 3.5) and -17.9% for the second set of experiments (Table 3.6), which is a difference of 1.5%. On average, the difference between the two sets of experiments for floating-point

Table 3.6 LeNet-5 Fault Injection outcomes with respect to the **Golden Custom** version.

Layer	Data	Observed				Masked	Safety		Gain w.r.t. FP32	
		Critical	Warning	Accept	Good		$<$	$\geq$	Safety*	Memory
L0	FP32	1.32%	0.06%	29.96%	28.98%	39.68%	31.34%	68.66%		-
	FP16	2.61%	0.12%	42.10%	40.86%	14.30%	44.84%	55.16%	-13.50%	2X
	FP8	3.30%	0.88%	47.08%	44.67%	4.07%	51.26%	48.74%	-19.91%	4X
	FxP32	0.03%	0.04%	24.45%	23.81%	51.67%	24.51%	75.49%	+6.83%	0
	FxP16	0.05%	0.08%	41.96%	40.81%	17.10%	42.09%	57.91%	-10.75%	2X
	FxP8	0.11%	0.15%	49.03%	46.94%	3.76%	49.29%	50.71%	-17.95%	4X
L2	FP32	1.37%	0.02%	23.88%	23.39%	51.34%	25.27%	74.73%		-
	FP16	2.59%	0.08%	35.66%	34.59%	27.07%	38.34%	61.66%	-13.07%	2X
	FP8	0.96%	0.89%	46.21%	43.17%	8.77%	48.06%	51.94%	-22.79%	4X
	FxP32	0.01%	0.01%	21.41%	20.93%	57.64%	21.43%	78.57%	+3.84%	0
	FxP16	0.03%	0.02%	38.70%	37.74%	23.52%	38.75%	61.25%	-13.47%	2X
	FxP8	0.06%	0.05%	47.94%	45.79%	6.17%	48.05%	51.95%	-22.77%	4X
L4	FP32	0.71%	0.00%	3.85%	3.86%	91.57%	4.56%	95.44%		-
	FP16	0.84%	0.13%	6.21%	6.23%	86.59%	7.17%	92.83%	-2.61%	2X
	FP8	0.32%	0.06%	10.45%	10.11%	79.06%	10.83%	89.17%	-6.26%	4X
	FxP32	0.00%	0.00%	4.06%	4.02%	91.92%	4.06%	95.94%	+0.50%	0
	FxP16	0.00%	0.00%	7.83%	7.75%	84.42%	7.83%	92.17%	-3.27%	2X
	FxP8	0.01%	0.00%	10.67%	10.33%	78.99%	10.68%	89.32%	-6.12%	4X
L6	FP32	0.61%	0.01%	7.69%	8.14%	83.54%	8.32%	91.68%		-
	FP16	1.19%	0.03%	11.47%	12.63%	74.67%	12.70%	87.30%	-4.39%	2X
	FP8	1.59%	0.39%	15.87%	17.10%	65.05%	17.85%	82.15%	-9.53%	4X
	FxP32	0.01%	0.04%	7.15%	7.27%	85.52%	7.21%	92.79%	+1.11%	0
	FxP16	0.03%	0.08%	13.48%	13.72%	72.69%	13.59%	86.41%	-5.27%	2X
	FxP8	0.05%	0.16%	17.25%	18.27%	64.27%	17.47%	82.53%	-9.15%	4X

* Safety increasing effect difference between a given scenario and FP32

variants (FP16 and FP8) over all the layers is 33.72%, while for fixed-point custom variants (FxP32, FxP16, and FxP8) is 14.81%. Practically, this means that a designer who chose to approximate the original Standard FP32 CNN version by using a custom floating-point variant without retraining it, would be exposed to higher safety degradation than by using the fixed-point alternative with the same bit-width. When a training is performed directly on the custom-data-type CNN, the safety degradation difference between the floating-point variants and the fixed-point ones is smaller. However, fixed-point ones still guarantee less critical fault occurrence, i.e., less than 0.12%. Finally, based on these outcomes, we believe that the CNN in FxP8 scenarios provides the best results in terms of memory footprint reduction, i.e., 4X, with a significant resilience, i.e. less than 0.45% critical faults.

Table 3.7 YOLO Data Type Accuracy Loss [%]

Scenario	Data type	Bit-width	Bit encoding	[%] Accuracy Loss
FP32	floating-point	32	1 sign, 8 exponent, 23 fractional	Ref.
FP16	floating-point	16	1 sign, 5 exponent, 10 fractional	42% masked, 58% acceptable
FP8	floating-point	8	1 sign, 4 exponent, 3 fractional	28% warning, 72% critical
FxP32	fixed-point	32	3 integer, 29 fractional	100% masked
FxP16	fixed-point	16	3 integer, 13 fractional	42% masked, 58% acceptable
FxP8	fixed-point	8	3 integer, 5 fractional	100% critical

Masked: IoU=1; acceptable: $0.95<\text{IoU}<1$; warning: $0.9\leq\text{IoU}\leq0.95$; critical: $\text{IoU}<0.9$ or different objects recognized

YOLO

To carefully select the custom data representation, we first analysed the weight distribution of YOLO. It is shown in Figure 3.14.

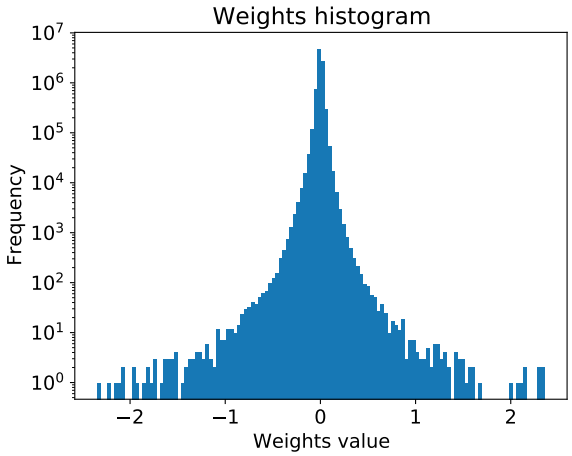


Figure 3.14 YOLO pre-trained weights distribution

As evidenced, all values are in the range -2.35 to 2.36 with most of them around zero. Thus, also for this CNN, the data type does not need higher dynamics while a high precision is preferred. Therefore, we selected the custom data type reported in Table 3.7.

Moreover, we computed the accuracy loss of the CNN resulting from the adoption of custom data types. As highlighted in Table 3.7, five different scenarios have been analysed. The second column of the table reports the data type used in each scenario, while the third column reports the bit-width of the weights. The fourth column shows the amount of bits allocated to encode the different parts of the number, i.e., sign, exponent, and fractional parts in the case of floating-point representations, and

Table 3.8 YOLO Fault List for Fault Injection Campaigns

		FP32 and FxP32 Bit-width=32		FP16 and FxP16 Bit-width=16		FP8 and FxP8 Bit-width=8	
Layer	**Connections**	**#Faults**	**#Injections**	**#Faults**	**#Injections**	**#Faults**	**#Injections**
L0	432	13,824	7,551	6,912	4,884	3,456	2,862
L2	4,608	147,456	14,954	73,728	13,577	36,864	11,466
L4	18,432	589,824	16,184	294,912	15,752	147,456	14,954
L6	73,728	2,359,296	16,524	1,179,648	16,410	589,824	16,184
L8	294,912	9,437,184	16,612	4,718,592	16,583	2,359,296	16,524
L10	1,179,648	37,748,736	16,634	18,874,368	16,626	9,437,184	16,612
L12	4,718,592	150,994,944	16,639	75,497,472	16,637	37,748,736	16,634
L13	262,144	8,388,608	16,608	4,194,304	16,575	2,097,152	16,510
L14	1,179,648	37,748,736	16,634	18,874,368	16,626	9,437,184	16,612
L15	130,560	4,177,920	16,575	2,088,960	16,509	1,044,480	16,380
L18	32,768	1,048,576	16,381	524,288	16,129	262,144	15,648
L21	884,736	28,311,552	16,631	14,155,776	16,621	7,077,888	16,602
L22	65,280	2,088,960	16,509	1,044,480	16,380	522,240	16,127

integer and fractional part in the case of fixed-point ones. To compute the accuracy loss of the network in the different scenarios, the inference for 7 images has been run on YOLO, clearly without injecting any faults, i.e., in a golden scenario. The results are reported according to the classification described for the object detector. The results show that for the FxP32 scenario there is no degradation, for the 16-bit data types (both FP16 and FxP16) in 42% of the cases (3 images out of 7) there is no degradation and for 58% of the cases (4 images out of 7) there is an acceptable degradation (i.e., all objects are correctly recognized, and the IoU metric is between 0.95 and 1). Finally, when using the 8-bit floating-point data type (FP8), the CNN is able to deliver usable results (i.e., classified as *warning*, $0.9 \leq \text{IoU} \leq 0.95$) for 28% of the inputs (2 images out of 7) and cannot produce correct outputs (*critical*) for the rest (5 images out of 7). On the contrary, with the 8-bit fixed-point data type (FxP8), the CNN is unable to provide the correct results at all, i.e., for all input images the output was classified as *critical*.

As for LeNet-5, also for YOLO we consider only the layers performing arithmetic computations involving trainable weights, i.e., the thirteen convolutional layers. Table 3.8 provides details about the configuration as well as the fault list of each layer. The first column (labelled *Layer*) reports the target layers; the second one (*Connections*) specifies the number of connection weights. The number of possible faults is computed as the multiplication between the connections number (*Connections*) and the weight size (*Bit-width*).

As the columns *#Faults* point out, the overall number of possible faults is very high, and this reflects in a non-manageable FI campaign execution time. Similarly to the experiments performed with LeNet-5, a subset of faults was selected. For each layer, we injected the number of faults reported in the columns *#Injections*. The number of faults was obtained by using the approach presented in [85], with an error margin of 1% and a confidence level of 99%. On average, 15.7k faults are injected in each layer for 32-bit scenarios (FP32 and FxP32), 15.3k for 16-bit scenarios (FP16 and FxP16), and 14.8k for 8-bit scenarios (FP8 and FxP8). The injections are performed by randomly selecting the faulty bit among all bits of the connection weights.

In line with the LeNet-5 FI campaign, two sets of experiments are carried out. First, we evaluated the reliability by using as a reference the **Standard** 32-bit floating-point CNN, and then by using the **Custom** fault-free one. In the following, we define faults belonging to the classes *Critical*, *Warning*, and *Accept* as events reducing the CNN safety. The sum of these contributions is represented by the symbol '$<$' in the Tables. On the other hand, we consider the masked faults as events leaving the safety of the CNN unaltered. Therefore, their contribution is represented by the symbol '$=$' in Tables. Table 3.9 reports the results of the first set of FI campaigns.

While for LeNet-5 we figured out that the fixed-point versions of the CNN had a higher average safety level (8.96%) with respect to the floating-point counterparts, for YOLO the difference is not as significant. Indeed, for YOLO, the fixed-point versions of the CNN have a slightly lower average safety level, i.e., -0.44%, with respect to the floating-point versions. This is calculated for the 32- and 16-bit versions, as the 8-bit fixed-point one always yields critical errors (see Table 3.7). If we include also the 8-bit versions, the fixed-point versions have a lower average safety level of -3.42% compared to the floating-point ones. This is probably due to the different distribution of the pretrained weight values for the YOLO network compared to LeNet-5 (compare Figure 3.14 with Figure 3.13). Indeed, for YOLO, the need of more bits for the integer part of the weights reduced the representation precision. Furthermore, it is worth highlighting that, in general terms, YOLO turns out to be much less resilient than LeNet-5. Indeed, while for LeNet-5 the percentage of critical faults is always lower than 3.42%, the YOLO network reaches up to 48.46% of critical faults (layer L0, FP8 scenario) and, as already mentioned, the FxP8 version produces critical faults even in a fault-free scenario. This is probably due to the much more stringent definition that we used for critical faults. Indeed,

for LeNet-5 we classified a fault as critical only when the top-1 prediction was wrong; conversely, for YOLO a fault is classified as critical also when an object is correctly classified, but it is not located perfectly (IoU<0.9). Finally, as in the LeNet-5 case, also for YOLO reducing the bit-width implies reducing the CNN resilience. In Table 3.10, we report the results of the second set of FI campaigns, where we compare the results of each scenario with the results obtained with the corresponding fault-free custom CNN. First, it is immediately clear that, also for this FI campaign, the CNN safety is impacted by the bit-width reduction. Second, also in this case, YOLO exhibits high critical fault occurrence, i.e., up to 42.08% in the layer L0 in FP8 scenario. Finally, for YOLO we notice the same phenomenon that we observed for LeNet-5: for floating-point custom variants (FP16 and FP8), the difference between the two sets of experiments (Tables 3.9 and 3.10) in terms of Safety gain with respect to FP32 is higher than for fixed-point ones (FxP32, FxP16, and FxP8). Indeed, on average, the difference for floating-point variants over all the layers is 59.38%, while for fixed-point custom variants is 13.92%. As already mentioned, this means that approximating the FP32 CNN version by using a custom floating-point variant without retraining exposes to higher safety degradation than by using the same bit-width fixed-point alternative.

Table 3.9 YOLO Fault Injection outcomes with respect to the **Golden Standard** version.

Level	Data	Observed			Masked	Safety		Gain w.r.t. FP32	
		Critical	Warning	Accept		<	=	Safety*	Memory
L0	FP32	20.17%	0.66%	16.25%	62.92%	37.08%	62.92%	-	-
	FP16	30.70%	1.04%	41.32%	26.95%	73.05%	26.95%	-35.97%	2x
	FP8	48.46%	49.88%	1.66%	0.00%	100.00%	0.00%	-62.92%	4x
	FxP32	13.35%	0.54%	12.43%	73.68%	26.32%	73.68%	10.76%	0
	FxP16	27.23%	1.11%	44.52%	27.14%	72.86%	27.14%	-35.78%	2x
L2	FP32	6.81%	0.10%	14.46%	78.63%	21.37%	78.63%	-	-
	FP16	12.07%	0.17%	49.85%	37.91%	62.09%	37.91%	-40.72%	2x
	FP8	16.67%	78.65%	4.68%	0.00%	100.00%	0.00%	-78.63%	4x
	FxP32	7.40%	0.43%	12.22%	79.95%	20.05%	79.95%	1.33%	0
	FxP16	14.38%	0.84%	51.95%	32.83%	67.17%	32.83%	-45.80%	2x
L4	FP32	5.05%	0.05%	11.24%	83.66%	16.34%	83.66%	-	-
	FP16	9.88%	0.10%	49.32%	40.70%	59.30%	40.70%	-42.97%	2x
	FP8	13.30%	82.95%	3.75%	0.00%	100.00%	0.00%	-83.66%	4x
	FxP32	6.32%	0.35%	11.86%	81.47%	18.53%	81.47%	-2.19%	0

	FxP16	12.45%	0.70%	52.81%	34.04%	65.96%	34.04%	-49.62%	2x
	FP32	4.02%	0.01%	7.59%	88.37%	11.63%	88.37%	-	-
	FP16	8.29%	0.04%	49.55%	42.13%	57.87%	42.13%	-46.25%	2x
L6	FP8	9.48%	88.49%	2.02%	0.00%	100.00%	0.00%	-88.37%	4x
	FxP32	4.49%	0.26%	11.04%	84.21%	15.79%	84.21%	-4.17%	0
	FxP16	8.82%	0.52%	53.75%	36.91%	63.09%	36.91%	-51.47%	2x
	FP32	3.41%	0.01%	3.94%	92.65%	7.35%	92.65%	-	-
	FP16	7.15%	0.02%	50.53%	42.30%	57.70%	42.30%	-50.35%	2x
L8	FP8	7.06%	91.51%	1.44%	0.00%	100.00%	0.00%	-92.65%	4x
	FxP32	2.84%	0.19%	9.22%	87.75%	12.25%	87.75%	-4.90%	0
	FxP16	5.49%	0.38%	54.32%	39.81%	60.19%	39.81%	-52.84%	2x
	FP32	3.26%	0.00%	1.70%	95.04%	4.96%	95.04%	-	-
	FP16	6.42%	0.02%	52.08%	41.49%	58.51%	41.49%	-53.55%	2x
L10	FP8	4.93%	94.16%	0.91%	0.00%	100.00%	0.00%	-95.04%	4x
	FxP32	1.67%	0.11%	6.85%	91.37%	8.63%	91.37%	-3.67%	0
	FxP16	3.14%	0.21%	54.33%	42.32%	57.68%	42.32%	-52.72%	2x
	FP32	3.17%	0.00%	0.84%	95.98%	4.02%	95.98%	-	-
	FP16	6.18%	0.07%	52.75%	41.00%	59.00%	41.00%	-54.98%	2x
L12	FP8	3.69%	95.54%	0.77%	0.00%	100.00%	0.00%	-95.98%	4x
	FxP32	0.76%	0.06%	5.34%	93.85%	6.15%	93.85%	-2.13%	0
	FxP16	1.39%	0.08%	54.87%	43.66%	56.34%	43.66%	-52.32%	2x
	FP32	3.27%	0.01%	3.37%	93.36%	6.64%	93.36%	-	-
	FP16	6.76%	0.04%	51.46%	41.74%	58.26%	41.74%	-51.62%	2x
L13	FP8	6.09%	92.80%	1.10%	0.00%	100.00%	0.00%	-93.36%	4x
	FxP32	1.86%	0.18%	9.08%	88.87%	11.13%	88.87%	-4.49%	0
	FxP16	3.61%	0.37%	55.13%	40.88%	59.12%	40.88%	-52.47%	2x
	FP32	3.13%	0.01%	1.25%	95.61%	4.39%	95.61%	-	-
	FP16	5.83%	0.07%	52.96%	41.13%	58.87%	41.13%	-54.47%	2x
L14	FP8	3.92%	96.08%	0.00%	0.00%	100.00%	0.00%	-95.61%	4x
	FxP32	1.03%	0.13%	5.96%	92.89%	7.11%	92.89%	-2.72%	0
	FxP16	2.05%	0.30%	54.55%	43.10%	56.90%	43.10%	-52.50%	2x
	FP32	0.82%	0.01%	0.19%	98.97%	1.03%	98.97%	-	-
	FP16	1.75%	0.04%	65.39%	32.82%	67.18%	32.82%	-66.15%	2x
L15	FP8	1.05%	98.95%	0.00%	0.00%	100.00%	0.00%	-98.97%	4x
	FxP32	0.34%	0.06%	0.36%	99.24%	0.76%	99.24%	0.26%	0
	FxP16	0.57%	0.11%	56.69%	42.63%	57.37%	42.63%	-56.35%	2x
	FP32	3.26%	0.02%	0.19%	96.53%	3.47%	96.53%	-	-
	FP16	6.10%	0.04%	53.82%	40.04%	59.96%	40.04%	-56.49%	2x
L18	FP8	5.55%	92.84%	1.61%	0.00%	100.00%	0.00%	-96.53%	4x
	FxP32	0.98%	0.14%	0.89%	98.00%	2.00%	98.00%	1.48%	0
	FxP16	2.04%	0.26%	57.61%	40.10%	59.90%	40.10%	-56.43%	2x
	FP32	3.22%	0.00%	0.06%	96.72%	3.28%	96.72%	-	-
L21									

	FP16	5.61%	0.02%	53.96%	40.41%	59.59%	40.41%	-56.31%	2x
	FP8	3.83%	94.51%	1.67%	0.00%	100.00%	0.00%	-96.72%	4x
	FxP32	0.37%	0.06%	0.56%	99.02%	0.98%	99.02%	2.29%	0
	FxP16	0.63%	0.17%	57.73%	41.47%	58.53%	41.47%	-55.25%	2x
L22	FP32	0.44%	0.00%	0.03%	99.52%	0.48%	99.52%	-	-
	FP16	0.89%	0.00%	56.66%	42.45%	57.55%	42.45%	-57.07%	2x
	FP8	1.14%	98.83%	0.03%	0.00%	100.00%	0.00%	-99.52%	4x
	FxP32	0.14%	0.02%	0.04%	99.79%	0.21%	99.79%	0.27%	0
	FxP16	0.29%	0.02%	57.05%	42.63%	57.37%	42.63%	-56.89%	2x

* Safety increasing effect difference between a given scenario and FP32

Table 3.10 YOLO Fault Injection outcomes with respect to the **Golden Custom** version.

Level	Data	Observed			Masked	Safety		Gain w.r.t. FP32	
		Critical	Warning	Accept		$<$	$=$	Safety*	Memory
L0	FP32	20.17%	0.66%	16.25%	62.92%	37.08%	62.92%	-	-
	FP16	30.70%	1.03%	28.10%	40.16%	59.84%	40.16%	-22.76%	2x
	FP8	42.08%	5.63%	37.02%	15.27%	84.73%	15.27%	-47.64%	4x
	FxP32	13.35%	0.54%	12.43%	73.68%	26.32%	73.68%	10.76%	0
	FxP16	27.23%	1.13%	23.50%	48.14%	51.86%	48.14%	-14.78%	2x
L2	FP32	6.81%	0.10%	14.46%	78.63%	21.37%	78.63%	-	-
	FP16	12.07%	0.18%	23.04%	64.71%	35.29%	64.71%	-13.92%	2x
	FP8	14.19%	2.64%	38.61%	44.55%	55.45%	44.55%	-34.07%	4x
	FxP32	7.40%	0.43%	12.22%	79.95%	20.05%	79.95%	1.33%	0
	FxP16	14.39%	0.84%	23.60%	61.16%	38.84%	61.16%	-17.46%	2x
L4	FP32	5.05%	0.05%	11.24%	83.66%	16.34%	83.66%	-	-
	FP16	9.88%	0.10%	18.29%	71.73%	28.27%	71.73%	-11.94%	2x
	FP8	11.84%	1.70%	30.70%	55.76%	44.24%	55.76%	-27.90%	4x
	FxP32	6.32%	0.35%	11.86%	81.47%	18.53%	81.47%	-2.19%	0
	FxP16	12.45%	0.70%	22.51%	64.34%	35.66%	64.34%	-19.33%	2x
L6	FP32	4.02%	0.01%	7.59%	88.37%	11.63%	88.37%	-	-
	FP16	8.29%	0.04%	13.34%	78.33%	21.67%	78.33%	-10.04%	2x
	FP8	8.37%	1.25%	22.66%	67.72%	32.28%	67.72%	-20.65%	4x
	FxP32	4.49%	0.26%	11.04%	84.21%	15.79%	84.21%	-4.17%	0
	FxP16	8.82%	0.53%	20.99%	69.66%	30.34%	69.66%	-18.72%	2x
L8	FP32	3.41%	0.01%	3.94%	92.65%	7.35%	92.65%	-	-
	FP16	7.15%	0.02%	7.80%	85.03%	14.97%	85.03%	-7.61%	2x
	FP8	6.22%	1.04%	16.41%	76.33%	23.67%	76.33%	-16.31%	4x
	FxP32	2.84%	0.19%	9.22%	87.75%	12.25%	87.75%	-4.90%	0
	FxP16	5.49%	0.38%	17.48%	76.65%	23.35%	76.65%	-16.00%	2x
L10	FP32	3.26%	0.00%	1.70%	95.04%	4.96%	95.04%	-	-
	FP16	6.42%	0.02%	3.89%	89.67%	10.33%	89.67%	-5.36%	2x
	FP8	4.42%	0.52%	10.04%	85.01%	14.99%	85.01%	-10.03%	4x
	FxP32	1.67%	0.11%	6.85%	91.37%	8.63%	91.37%	-3.67%	0
	FxP16	3.14%	0.21%	12.41%	84.23%	15.77%	84.23%	-10.81%	2x
L12	FP32	3.17%	0.00%	0.84%	95.98%	4.02%	95.98%	-	-
	FP16	6.18%	0.07%	2.20%	91.55%	8.45%	91.55%	-4.44%	2x
	FP8	3.44%	0.23%	7.94%	88.39%	11.61%	88.39%	-7.59%	4x
	FxP32	0.76%	0.06%	5.34%	93.85%	6.15%	93.85%	-2.13%	0
	FxP16	1.39%	0.08%	9.90%	88.63%	11.37%	88.63%	-7.35%	2x
L13	FP32	3.27%	0.01%	3.37%	93.36%	6.64%	93.36%	-	-
	FP16	6.76%	0.04%	6.42%	86.78%	13.22%	86.78%	-6.58%	2x

	FP8	5.20%	1.18%	14.59%	79.03%	20.97%	79.03%	-14.33%	4x
	FxP32	1.86%	0.18%	9.08%	88.87%	11.13%	88.87%	-4.49%	0
	FxP16	3.61%	0.37%	16.92%	79.09%	20.91%	79.09%	-14.27%	2x
	FP32	3.13%	0.01%	1.25%	95.61%	4.39%	95.61%	-	-
	FP16	5.83%	0.07%	2.66%	91.43%	8.57%	91.43%	-4.18%	2x
L14	FP8	3.60%	0.25%	4.60%	91.54%	8.46%	91.54%	-4.06%	4x
	FxP32	1.03%	0.13%	5.96%	92.89%	7.11%	92.89%	-2.72%	0
	FxP16	2.05%	0.30%	11.29%	86.35%	13.65%	86.35%	-9.25%	2x
	FP32	0.82%	0.01%	0.19%	98.97%	1.03%	98.97%	-	-
	FP16	1.75%	0.04%	0.34%	97.87%	2.13%	97.87%	-1.10%	2x
L15	FP8	0.92%	0.17%	0.46%	98.45%	1.55%	98.45%	-0.53%	4x
	FxP32	0.34%	0.06%	0.36%	99.24%	0.76%	99.24%	0.26%	0
	FxP16	0.57%	0.12%	0.57%	98.74%	1.26%	98.74%	-0.23%	2x
	FP32	3.26%	0.02%	0.19%	96.53%	3.47%	96.53%	-	-
	FP16	6.10%	0.04%	0.38%	93.48%	6.52%	93.48%	-3.05%	2x
L18	FP8	5.22%	0.86%	10.56%	83.35%	16.65%	83.35%	-13.18%	4x
	FxP32	0.98%	0.14%	0.89%	98.00%	2.00%	98.00%	1.48%	0
	FxP16	2.04%	0.26%	2.01%	95.70%	4.30%	95.70%	-0.83%	2x
	FP32	3.22%	0.00%	0.06%	96.72%	3.28%	96.72%	-	-
	FP16	5.61%	0.02%	0.14%	94.23%	5.77%	94.23%	-2.50%	2x
L21	FP8	3.62%	0.48%	9.10%	86.80%	13.20%	86.80%	-9.92%	4x
	FxP32	0.37%	0.06%	0.56%	99.02%	0.98%	99.02%	2.29%	0
	FxP16	0.63%	0.17%	1.19%	98.00%	2.00%	98.00%	1.28%	2x
	FP32	0.44%	0.00%	0.03%	99.52%	0.48%	99.52%	-	-
	FP16	0.89%	0.00%	0.04%	99.07%	0.93%	99.07%	-0.45%	2x
L22	FP8	1.13%	0.06%	0.21%	98.61%	1.39%	98.61%	-0.91%	4x
	FxP32	0.14%	0.02%	0.04%	99.79%	0.21%	99.79%	0.27%	0
	FxP16	0.29%	0.02%	0.10%	99.59%	0.41%	99.59%	0.07%	2x

* Safety increasing effect difference between a given scenario and FP32

3.2 Dynamic Parameters in Artificial Neural Networks

Artificial neurons in artificial neural networks can be considered as dynamic parameters because their values depend not only on connecting weights and biases, but also on input stimuli. A graphic illustration of an artificial neuron is given in 3.15.

It is worth underlying that individual network parts differ in their error resilience [67]. Particularly, neurons exhibit different fault tolerance and resilience levels.

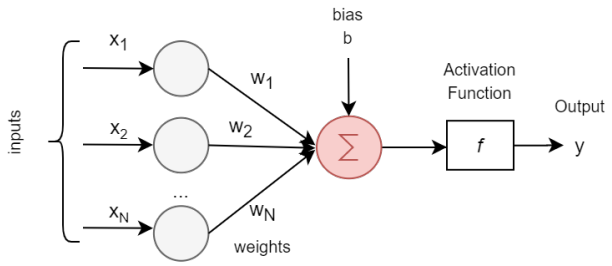


Figure 3.15 Basic Scheme for a neuronal computation.

Some of them strongly contribute to the output classification of the neural network, and their failures have a greater influence on the degradation of the final predictions. If a neuron contributes more to the final prediction, it is considered as critical or important; otherwise, it is considered as resilient or redundant. An error in critical neurons may significantly compromise the accuracy of the final neural network prediction. From another perspective, determining the most critical neurons of a neural network means identifying all those neurons carrying more information than others.

3.2.1 Proposed Approach

The investigation of ANNs as mathematical models induced the following observation: the output of a neuron is nothing but the result of a summation. Based on the above insight, we demonstrate that critical neurons are those producing at their output the highest absolute values during the inferences. Moreover, our theoretical-based criticality analysis is founded on a further key observation. According to behavioural theories in Neuroscience [88], **brain memories occur when *specific groups* of neurons are reactivated**. Based on precise stimuli, neurons become active in a particular pattern of neuronal activity. It means that if our brain thinks of a sky or a meadow, different ensembles of neurons become active. By transferring this concept to the world of artificial neural networks, in a multi-output neural network, the contribution of a single neuron can be seen in two ways. One is meant for guaranteeing the correct prediction of the single output class, the other is meant for guaranteeing the correct predictions of the entire multi-output neural network. In other words, imagine you have a 2-output neural network classifying apples and pears pictures, there will be neurons that are more significant for the class *apple* and others for the class *pear*; at

the same time, all the neurons guarantee the overall correct predictions. To this end, in [89] we propose a methodology to assign resilience scores to individual neurons. It builds on three steps:

1. **Class-oriented Analysis (CoA):** For each single output class, the most important neurons are extracted with Algorithm 1 and sorted in descending order based on their criticality. This sorting is saved on a final list, named *score map* which is created for each output class.

2. **Network-oriented Analysis (NoA):** The process is repeated for the entire neural network (without distinguishing between output classes) and a single score map is obtained.

3. **Final Network-oriented Score-Map:** The network-oriented score map is updated based on the outcomes of the class-oriented analysis.

These three phases are carefully described in the following.

In the first class-oriented analysis, we consider the importance of a neuron related to each single output class. In particular, it is worth specifying that we refer to the neuron as: each pixel in the output feature maps of a convolutional layer, each node in the pooling (min, max, average) or fully connected layers. Typically, batch normalization and activation functions (e.g., rectified linear unit, sigmoid, Gaussian) are not considered as independent layers, and thus, they do not come with additional neurons. In Algorithm 1, scores are assigned to neurons considering both static and dynamic parameters of the ANN: by catching the neuron's output (y), both the weights (static parameters) and the inputs (dynamic parameters) are taken. At the beginning, an initial score equal to zero is assigned to each neuron (line 6). Thereby, for each output class of the neural network (line 7), a new score map is created (line 8). For each instance in the training dataset related to the specific output class, a forward propagation cycle is performed (line 10). In the meantime, a score is given to each neuron (lines 13-15), by averaging the absolute output values produced during all the inferences (line 20). The score gets updated at every inference iteration. At the end of the process, each class keeps its own score map, where every neuron holds a score value (line 22). The highest absolute scores are relative to the most critical neurons for that given class. In more detail, the score map is represented as a list sorting the neurons from the highest to the lowest value. It is worth noting

Algorithm 1: Assignment of resilience scores to individual neurons.

```
1  N ← Total neurons;
2  C ← Output classes;
3  I_{i, i∈[0,C]} ← Inputs for a specific class;
4  score_{k, k∈[0,N]} ← Score assigned to a neuron;
5  y_{k, k∈[0,N]} ← Output value of a neuron;
6  score_{k, k∈[0,N]} ← 0;
7  for each output class of the network c∈[0,C] do
8      new()
9      for each instance in the training dataset i, i∈[0,I_c] do
10         inference()
11         for each neuron k, k∈[0,N] do
12             if i == 0 then
13                 score_k ← | y_k |
14             else
15                 score_k ← score_k + | y_k |
16             end
17         end
18     end
19     for each neuron k, k∈[0,N] do
20         score_k ← score_k/I_C
21     end
22     save()
23 end
```

that the output is sampled for every neuron after the eventual batch normalization or activation function. Starting from the classes' score maps, it is possible to extract a subset (t) of critical neurons in the form *className_critical_t*. The subset parameter (t) determines the amount of neurons that will be considered as critical, and it is not a fixed value. Defining that number means tuning the reliability of a neural compuring system: in other words, the larger the size, the larger the set of neurons that will be considered critical.

Next, in the network-oriented analysis (NoA) phase, we build a final score map where neurons are sorted based on the magnitude of their average contribution over the training set, without differentiating between the output classes. It is worth pointing out that both the score maps resulting from the NoA and the one from the CoA contain all the neurons of the neural network: only their values change, and consequently the ordering. To this end, Algorithm 1 is run again: line 7 is removed

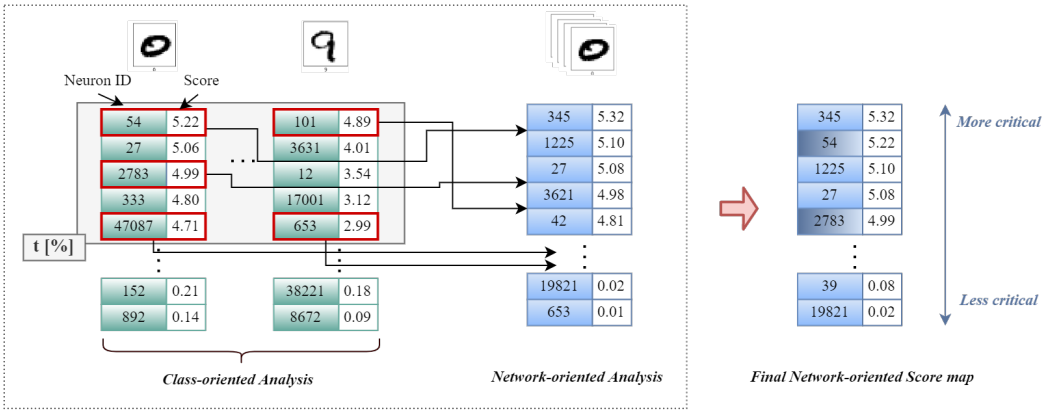


Figure 3.16 The critical neuron identification process: a practical example with the MNIST dataset.

and line 9 is modified so that the inputs are picked up from the entire training dataset. Then, since it might happen that neurons that were found to be critical for individual classes take on a low value in the NoA, the outcome of this score map is updated considering the score maps resulting from the CoA. All neurons assuming a higher value in the class-oriented score map (given a subset parameter t) are overwritten. The set of critical neurons to take into account for the final network-oriented score map is computed by doing the union without repetitions of all the classes' score maps (C), as follows:

$$critical\_t \leftarrow \bigcup_{i=1}^{C} c_{\mathrm{i}}\_critical\_t \tag{3.4}$$

As an example, the whole process is illustrated in Figure 3.16 for a generic neural network trained on MNIST. First, a percentage of critical neurons is selected (t) from the class-oriented score maps. Among these neurons, all those having a lower value in the network-oriented score map are overwritten with the highest value in the classes, i.e., the red squares. Whereas, neurons such as Neuron *27* in Class *0* having a value lower in the t% of the CoA are not updated in the final score map. Interestingly, Neuron *653* in Class *9* assumes the lowest value in the network-oriented score map and, being part of the t%, is updated. In the end, as depicted on the right side of the Figure 3.16, the final updated network-oriented score map is produced, where the per-class criticality is considered with a t factor. The larger t is, the more neurons

are considered critical and therefore strengthened in the final network-oriented score map.

In the literature, similar approaches have been developed to identify the network's critical neurons. As stated before, they are based only on a network-oriented analysis [67, 68, 3]. In this paper, we demonstrate that our methodology gets the analysis stronger in terms of reliability. Although this approach is applied to a ANN performing image classification, it can also be extended to other tasks.

3.2.2 Experimental Results

To experimentally prove the effectiveness of the proposed methodology, we used three different CNNs trained on three representative and popular datasets: MNIST [90], SVHN [91], and CIFAR-10 [92]. The MNIST dataset is used to recognize handwritten digits and consists of a training set of 60,000 28x28 gray-scale images, and a test set of 10,000 examples. The Street View House Numbers (SVHN) is a real-world image dataset obtained from house numbers in Google Street View images. It contains more than 600,000 digit images: 73,257 digits are used for training, 26,032 digits for testing, and additional ones as extra training data. SVHN comes in two formats: the original and 32x32 cropped. We used the latter. CIFAR-10 dataset is an object recognition dataset made of 60,000 32x32 colour images: 50,000 training images and 10,000 test images [92]. We implemented three CNNs using PyTorch [93] on a Linux server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM. PyTorch is a fast and flexible framework widely used by both industry and academia for deep learning and machine learning based applications. The first neural network is a custom version of LeNet-5 and is composed of 7 layers (i.e., 3 convolutional, each one followed by max pooling and the last fully connected) with an input size of 28x28x1. After each convolutional layer, the Rectified Linear (ReLU) activation function is used. It was trained and tested on MNIST dataset, reaching a 99.31% of accuracy over the MNIST test set. Next, we implemented a second neural network following the ConvNet [94] model. It was trained and tested over SVHN dataset, classifying correctly the 92.01% of test images. It consists of 2 convolutional layers, each one followed by LP-pooling and normalization layers (also known as 2-stages or multistage features). They are fed to a 2-layer classifier (fully connected layers). The last CNN was built with the All-CNN configuration [95], an architecture that consists solely of convolutional layers ([95] demonstrates

Table 3.11 ANN Benchmarks

CNN Model	Dataset	Application	Accuracy	Total Neurons
Custom LeNet-5	MNIST	Image Classification	99.31	48,650
ConvNet	SVHN	Object Recognition	92.01	185,374
All-CNN	CIFAR-10	Object Recognition	90.57	361,046

that max-pooling can simply be replaced by a convolutional layer with increased stride without loss in accuracy). The architecture is made of 9 convolutional layers. We exploited the CIFAR-10 dataset for this last CNN. The final accuracy was equal to 90.57% over the test set. Further details, such as the total amount of neurons, are provided in Table 3.11.

To profile the criticality of the three CNNs, Algorithm 1 was executed to assign resilience scores to individual neurons. As stated before, the MNIST, SVHN, and CIFAR-10 *training* datasets were used to assign the resilience scores. In contrast, their *test* datasets were used for the fault injections experiments (both at the software and the RTL level).

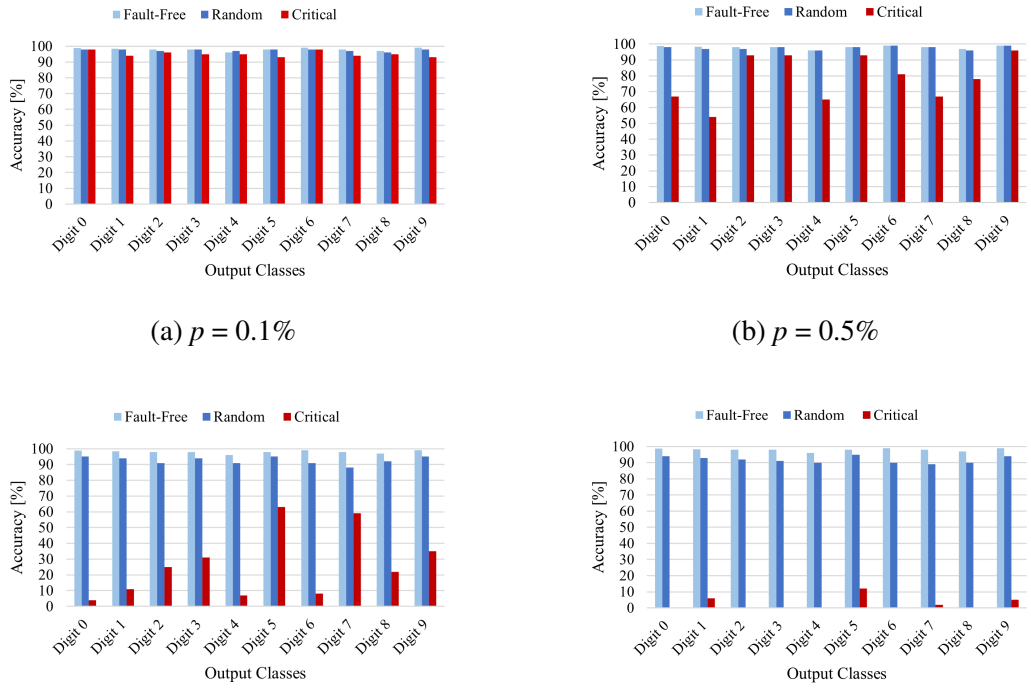
Class-oriented Analysis

Initially, each training dataset was divided into subclasses, i.e., the number of outputs (in our case study, we had 10 outputs for all the three exploited neural networks, but the same reasoning applies to a different number of output classes). Each subclass contained only the images representing the selected output class. Hence, the proposed algorithm was executed to obtain the 10 final score maps for each CNN. Each of them ordered the total neurons from the one activated with the highest average value to the one with the lowest (from the most critical to the least one), for that particular output class. Next, we performed software FI campaigns (i) to shed light on the importance of the class-oriented analysis, and (ii) to show that individual output classes hold different robustness levels with respect to errors. This certainly depends on the training phase and structure of the dataset that is used to train the network (typically, in the training set, the training images are not evenly distributed among the output classes). We exploited the dropout probability fault model (*p-dropout*), in which a fraction of neurons outputs is set to zero, and thus, their contribution is cancelled. The same fixed amount of neuron outputs (p) was set to zero in two scenarios and, after the injection, the resulting accuracy of each CNN was measured by running

the total test set of images (which was different from the training set used to gather the resilience scores). For each output class, in the first scenario (*Random*), neurons were randomly chosen from the class score map. In the second (*Critical*), the same amount of neurons was neatly selected always starting from the top of the class score map, i.e., always starting from the most critical neurons. As for the *Random* scenario, since we rely on a random choice of neurons to kill, the experiments were repeated 1,000 times (every time picking up different p random neurons) and we report in Figures 3.17-3.18-3.19 the average percentage obtained through the experiments. The experiments were conducted for each output class of the targeted CNNs and, particularly, they were replicated for growing p-percentages. Experimental results for the three FI campaigns are reported in Figures 3.17-3.18-3.19. The scenario (*Fault-free*) is the golden accuracy of the class and, as for the *Random* and *Critical*, it was computed by running only the inferences of the images belonging to the given output class. It is evident that random injections do not affect, or only to a negligible extent (when p gets bigger), the behaviour of the neural network. Indeed, in all cases the accuracy fluctuates around the *Fault-free* one, apart from the third case (p=5 %) where it slightly decreases. This confirms the theory under which neural networks are equipped with more neurons than they really need [13]. In fact, up to a certain point, they can get enough of some neurons and still work correctly. By contrast, this trend is not confirmed in the *Critical* scenario: the accuracy of the output classes remarkably drops when killing the p highest neurons.

Concerning MNIST LeNet (Figure 3.17), for p=0.1 % the maximum percentage variation from the *Fault-free* accuracy to scenario *Critical* is equal to 6.13 % and corresponds to the last class (digit 9). Then, when killing p=0.5 % critical neurons, the highest percentage variation drastically increases, reaching the 44.33 % for the second class (digit 1), where the CNN accuracy drops from the *Fault-free* 99.33 % to 54.54 %.

The situation worsens with p=1 % for all the classes except for digits 5 and 7, where the accuracy keeps close to 60%. In the last scenario, when dropping the p=1.5 % of critical neurons from the classes, the correct predictions become 0 or close to it. As illustrated in the graph, it turns out that for the LeNet trained on MNIST dataset, the most robust class corresponds to the digit 5, while the least robusts are digit 1 and digit 4.



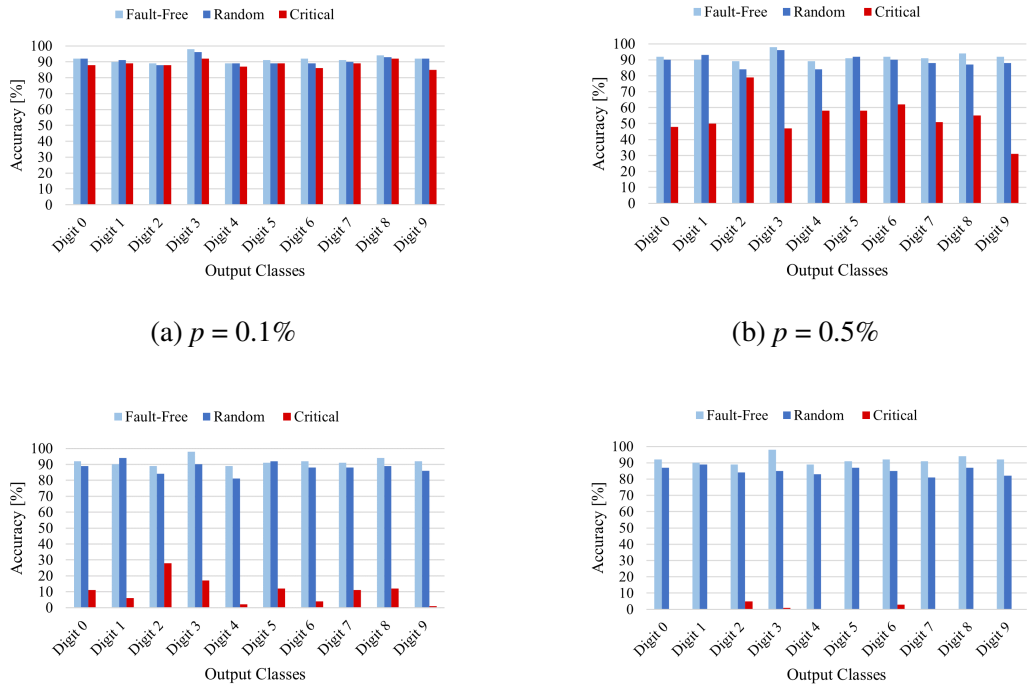
(a) $p = 0.1\%$ (b) $p = 0.5\%$

(c) $p = 1\%$ (d) $p = 1.5\%$

Figure 3.17 MNIST LeNet: software fault injection campaigns on *random* and *critical* neurons. A fixed percentage p of neurons is dropped.

The outcome of the software fault injection for the SVHN network (ConvNet) is shown in Figure 3.18. When crashing *p=0.1 % Critical* neurons the CNN accuracy decreases until reaching a maximum percentage variation of 7.3% (digit 9). With the increase of the dropped critical neurons *p=0.5 %* we observe a considerable drop in accuracy, with a maximum of 61.9% of variation percentage again for digit 9. The correct functionality of the neural network worsens considerably for *p=1 %* until it reaches zero in almost all classes for *p=1.5 %*. Overall, the most robust class turns out to be the third one, i.e., digit 2. In fact, despite the dropped neurons, it is able to keep an accuracy close to 80% with the highest 927 neurons dropped (*p=0.5 %*). On the other hand, the least resilient class is the last one (digit 9). In fact, it is significantly sensitive to removed neurons (starting from *p=0.1 %*).

With respect to LeNet (MNIST) and ConvNet (SVHN), All-CNN (CIFAR-10) demonstrates greater sensitivity. As shown in Figure 3.19, since from *p=0.1 %* we can observe a greater reduction in accuracy (the maximum drop in accuracy is for Class *"Horse"* and corresponds to 16.2% from the *Fault-free* value). Also, for



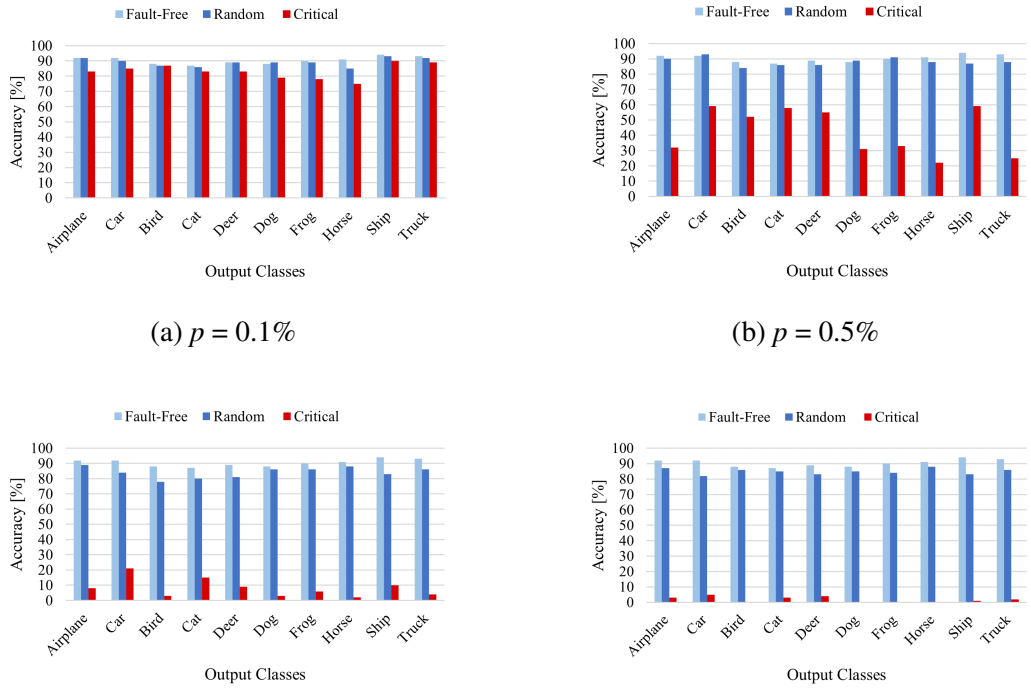
(a) $p = 0.1\%$ (b) $p = 0.5\%$

(c) $p = 1\%$ (d) $p = 1.5\%$

Figure 3.18 SVHN ConvNet: software fault injection campaigns on *random* and *critical* neurons. A fixed percentage p of neurons is dropped.

p=0.5 %, all the classes' accuracy keeps under the 60%, with the maximum variation percentage from the golden accuracy equal to 69.82% for the Class *"Horse"*. When the dropped neurons become *p=1 %* from each class (meaning about 1,854 neurons over the total 185,374), the accuracy of the classes drops below 20 %, except for the Class *"Car"* with the 21.4 %. The experimental results indicate that the most robust class is the class *"Car"* while the least resilient one is the class *"Horse"*.

Overall, data from Figures 3.17-3.18-3.19 suggest similar conclusions and the different per-class resilience is confirmed in the three targeted CNNs. It is clear that the *p*-percentage refers to different neural networks of different sizes: the CIFAR-10 network contains almost 7.5x and 4x times the amount of neurons than MNIST and SVHN networks, respectively. It means that the former starts misbehaving with about 361 neurons crashed (*p=0.1 %*), while the other two (with the same percentage) with about 49 and 185, respectively. Finally, these outcomes experimentally demonstrate the initial assumption stating that there are neurons playing a key role, and therefore, defined critical for the output classes.



(a) $p = 0.1\%$ (b) $p = 0.5\%$

(c) $p = 1\%$ (d) $p = 1.5\%$

Figure 3.19 CIFAR-10 All-CNN: software fault injection campaigns on *random* and *critical* neurons. A fixed percentage p of neurons is dropped.

The reader should note that, to avoid confusion, we used the p parameter to indicate the amount of neurons dropped from the individual classes, and the t parameter to represent the set of critical neurons in the network-oriented score map. They are both percentages working on the score maps, but the first is used in the class-oriented analysis and is used to drop, the second in the network-oriented and serves as a parameter to indicate the reliability level of the system.

Network-oriented Analysis

So far, we have performed software fault injection campaigns to demonstrate that each output class owns a set of neurons that are more important than others for correctly predicting their images. If this is taken into account when ranking the network's neurons based on their criticality, we experimentally demonstrate that the reliability analysis becomes more accurate. In this phase, we computed the neurons' resilience scores without differentiating among the output classes. Hence, the entire

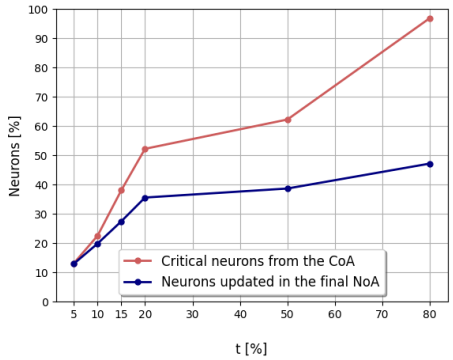
MNIST, SVHN, and CIFAR-10 training datasets were used to collect the neurons' scores. We obtained a network-oriented score map for each CNN (LeNet, ConvNet, All-CNN). For the sake of clarity, these lists did not consider the contribution of the classes yet.

Final Network-oriented Score Map

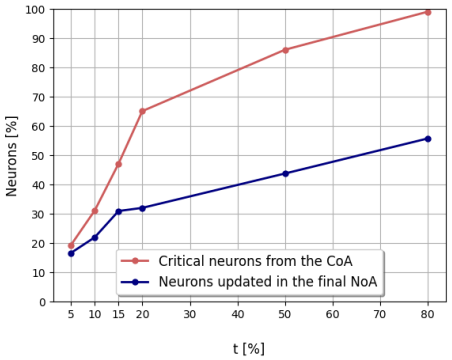
After the CoA and NoA, a final network-oriented score map was obtained based on the analysis of the class-oriented approach and given the t parameter. This t value represents the amount of neurons taken from the classes score maps (always from starting from the top positions). By applying (3.4), i.e., the union (without repetition) operation, we removed duplicate neurons by keeping the highest values assumed among the classes rankings. Therefore, with each t value, we computed the percentage of neurons with the equation (3.4) in the CoA: their value will be compared with that obtained in the initial NoA. Then, for each neuron in the set (3.4), if its value was higher than that in the NoA, its value was updated in the final score map; otherwise, the highest from the NoA was kept.

Next, to study the influence of the CoA on the NoA with a growing t percentage, we performed a further study on the three CNNs. The first experiment is shown in Figure 3.20a and targets LeNet (MNIST). The x-axis represents the increasing t percentage, whereas the y-axis shows the corresponding percentage of neurons over the total. The red line outlines the percentage of critical neurons calculated with (3.4) after the CoA, for the corresponding t value. The blue line illustrates the percentage of neurons that are updated in the final network-oriented score map due to their higher criticality value. As it turns out, the lower the t percentage, the higher the percentage of neurons in the set (3.4) whose value is updated in the final network-oriented score map. For example, when t=*5 %* in LeNet (MNIST), the union without repetition (3.4) includes 6,291 critical neurons (red point), meaning the 12 % of the total 48,650 neurons. A total of 6,212 neurons (blue point) over 6,291 (red point) are overwritten with the values obtained from the CoA (3.4). In other words, the 98.74 % of neurons has a different level of criticality when moving from the class-oriented to the network-oriented methodology. For higher t values, this percentage reduces, reaching 45% for t=*80 %*.

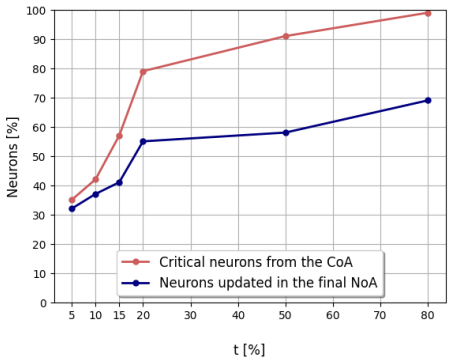
Furthermore, as illustrated in Figures 3.20b and 3.20c, the same analysis is reproduced for ConvNet (SVHN) and All-CNN (CIFAR-10). Similarly to what



(a) LeNet (MNIST)



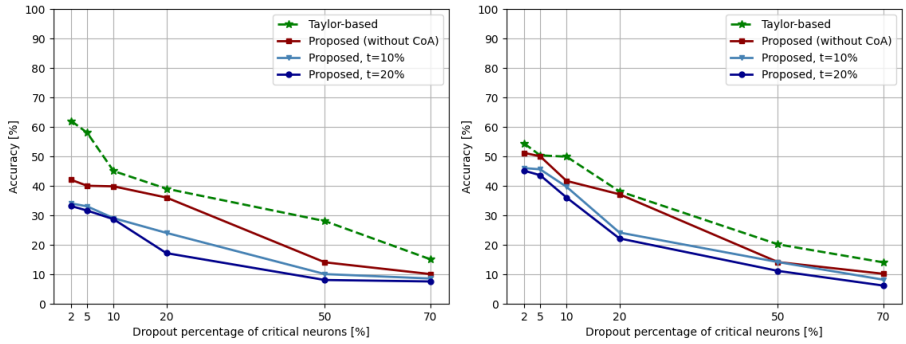
(b) ConvNet (SVHN)



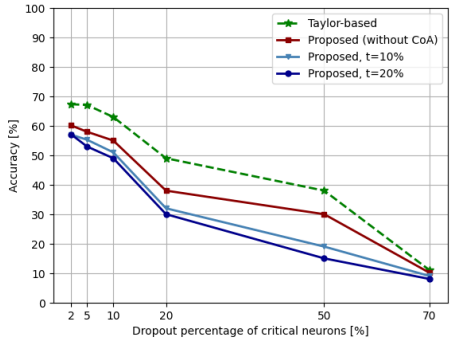
(c) All-CNN (CIFAR-10)

Figure 3.20 Network-oriented Analysis with a growing t percentage of critical neurons from the CoA.

discussed for LeNet, the lower the set of critical neurons in (3.4) (determined by the t percentage and showed as a red line), the higher the percentage of neurons in this set that will be updated in the final network-oriented score map (blue line). Overall, we can say that even with the highest t=*80 %*, the number of neurons having a criticality higher in the CoA it is approximately half of the total neurons and, as experimentally demonstrated, depends also on the size of the neural network. Specifically, when t=*80 %* we updated 45.57%, 55.64%, 69.04% of neurons (respectively for LeNet, ConvNet, and All-CNN) in the final network-oriented score map. A further observation related also to the size of the targeted neural networks is the initial set of critical neurons for t=*5 %*. The smaller the network size, the higher the probability of having replicated neurons. In other words, when t=*5 %* the union without repetition yields the following figures: 12.93%, 19.18%, and 35.93% for LeNet, ConvNet, and All-CNN, respectively.



(a) LeNet (MNIST) (b) ConvNet (SVHN)



(c) All-CNN (CIFAR-10)

Figure 3.21 Showing the robustness of the proposed approach based on the contribution of the CoA and the NoA (blue lines). Its effectiveness is compared against [3] (green line) and our proposed methodology without the contribution of the class-oriented analysis (red line).

Finally, to demonstrate how the proposed profiling methodology behaves with respect to a state-of-the-art network-oriented methodology proposed in [3], we present a further analysis. As stated, the final network-oriented score map contains the neurons of the network ordered based on their criticality, reinforced by a t percentage with the CoA. We carried out software FI campaigns for the three CNNs. Specifically, a fixed percentage of critical neurons was set to 0 in three scenarios: the proposed (CoA + NoA), the proposed without the CoA, the Taylor-based [3]. Then, the accuracy of the neural network over the entire test set was computed. Specifically, we removed a 2 %, 5 %, 10 %, 20 %, 50 %, 70 % of critical neurons from the respective ordered network-oriented lists. For the purpose, two different network-oriented score maps were created following our proposed approach, each one with a growing set of critical neurons (t=10 %, t=20 %). The aim was to demonstrate that

with a growing t we obtained a more robust network-oriented score map. Figure 3.21 shows the results of our FI simulations with the dropout model for the MNIST, SVHN, and CIFAR-10 CNNs. As it turns out, the accuracy that the CNN under assessment achieves is always lower when removing the same percentage of critical neurons from our network-oriented score map. It means that: first, the ordering of the critical neurons greatly affects the reliability of the system; second, our final score map holds (in the highest positions) neurons that are really critical not only to the entire neural network, but also to individual output classes.

3.3 Chapter Summary

In summary, the chapter proposes software-level methodologies to assess the reliability of neural networks by considering the contribution and the vulnerabilities of **static** (i.e., weights) and **dynamic** (i.e., neurons) units. As described, weights in neural networks can be considered as static parameters because their values, after they have been properly trained, are fixed and constant. On the other hand, the values of neurons depend on both weights and input stimuli, and therefore, can be regarded as dynamic entities. Specific techniques have been proposed to evaluate the resilience of neural networks when their static parameters are represented as (i) 32-bit floating point numbers, (ii) fixed-point numbers of different bit width. Trade-offs between reliability and memory footprint in ANN-based applications are discussed in the chapter and, interestingly, results about the most critical bits in floating- and fixed-point representations are presented. When it comes to floating-point weights, results provide evidence that the most critical bits are those used to represent the exponent part (specifically, the most significant bit): faulty bits in that place could cause the value to explode and easily propagate through the neural network. On the other hand, critical bits in fixed-point numbers include all bits of the integer part. This leads to the following observation: although fixed-point weights with a reduced bit-width representation allow compressing the DNN model by reducing its memory footprint, the likelihood of having faults in critical bits increases (more faulty bits lead to critical behaviours). These are interesting achievements because, on the basis of these outcomes, a designer may develop a suitable mitigation technique which mainly targets the more noticeable criticalities.

Concerning the dynamic parameters, i.e., neurons, a methodology is proposed to identify the most critical neurons of a neural network. It is experimentally demonstrated that not all neurons play the same role in the final classification task. It is known that neural networks are furnished with more neurons than needed, but, which neurons can get rid of is the focus of the class-oriented analysis. On the heels of this study, we presented a technique to identify the most critical neurons in a neural network and sort them according to their criticality. Compared to the state-of-the-art works, our scope is not to compress the network by removing unimportant neurons or connections. Rather, our scope is to find *where* the most important neurons are and to profile the application criticality. Most importantly, contrary to pruning approaches, the methodology presented in this work does not require additional learning steps or the adoption of a threshold, which are computationally expensive. The method bases on two levels of analysis: first, the neuron is viewed as an element of each output class (class-oriented analysis); second, the same is interpreted as belonging to the entire neural network (network-oriented analysis). The method can be efficiently applied to neural networks with any layers and any typologies. The experimental results show that our final sorting is more effective than those based only on a network-oriented analysis, since we also consider the criticality of class-specific neurons. To the best of our knowledge, this is the first time that the importance of the neuron as related to the single output class is taken into account.

It is worth to highlight that, in [89], the entire training set was used to assign resilience values to individual neurons. In a recent study, we experimentally prove that even a reduced number of validation set instances are useful to define the most critical neurons. To measure how good rankings obtained with a subset of the validation set are, we measure their intersection with the ordering computed using all the images from the test set. Particularly, in Fig. 3.22, we report the intersection for differently sized subsets of the validation set for ResNet-32, a deep convolutional neural network. It is clear that, just by using 20% of the validation set images, it is possible to obtain a neuron ordering that is similar to the one obtained using the test set, independently of the considered percentage (namely *pNeu*). It follows that the approach proposed in [89] requires fewer images than initially suggested to produce a meaningful neuron ranking.

Future work will extend this analysis to deeper ANNs and different datasets. The reader should note that the adoption of MNIST, SVHN and CIFAR-10 datasets in [89] was consistent with the considered low-power and resource-constrained ASIC

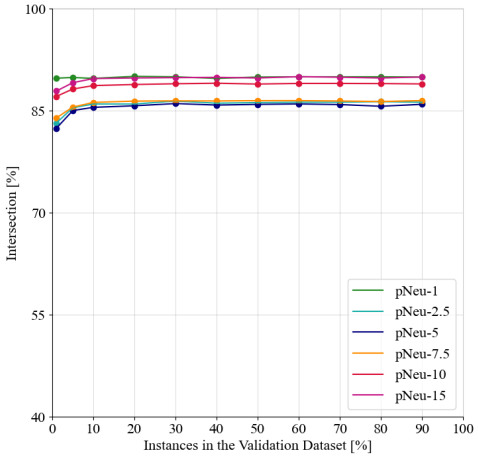


Figure 3.22 Intersection between the critical neuron ranking obtained using the test set and the rankings obtained using subsets of the validation set. The figure shows how, for ResNet-32, the intersection remains stable for different values of *pNeu*.

world. In the future, we will exploit deeper ANNs and more complex datasets moving the target to GPUs and high-performance architectures, or ad-hoc Hardware Neural Network chips. Moreover, in the future, we will address also other fault models such as transient errors.

Chapter 4

Reliability Assessment at the Architectural Level

4.1 Proposed Approach

ANN reliability is commonly assessed by performing FI campaigns. However, due to the excessive time required to run a single inference at Hardware Description Level (HDL), fault injections are typically performed at the software level by targeting only the neural network model. This is crucial to get an overall assessment of neural network behaviour in faulty situations, however, it is also well known that the hardware architecture employed has a significant impact on the overall system's dependability [21, 96]. Indeed, devices are getting more prone to physical errors as the shrinking of semiconductor technologies continues, and the probability that some computing elements fail increases as well. Therefore, even thought researchers claim that neural networks are potentially able to absorb some degrees of vulnerability due to their natural resilience properties, it is essential to evaluate the resilience of the entire system, especially in safety-critical domains, where it might be needful to assess the robustness of a system during the design phase, *before* the fabrication process. It is therefore needed to move from pure software approaches to lower-level ones. Nevertheless, the big issue when working in simulation at RTL or, even worse, gate level, is the time required to run single inference phases. As an example, a state of the art neural network, i.e., ResNet-32, counts on about 68 million of MAC operations to do a single inference cycle: a value that, in simulation, becomes easily

unfeasible. To understand the neural network behaviour in presence of a fault, a fault must be injected at low level (e.g., RTL), and then the inference has to be performed. This process is very time-consuming, even though using commercial fault simulation tools [55].

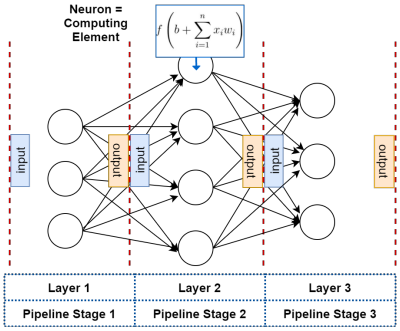
To fill this gap, this work proposes a methodology to drastically reduce simulation and fault injection time when running DNNs at RTL [97]. It presents the architecture of a multi-level fault injector that, by reproducing the flow of a pipelined process, is capable of handling many inferences in parallel. As a matter of fact, layers in a neural network process the information in a sequence of steps: when a layer has completed its elaboration, it produces an output that becomes the input for the subsequent layer, and so on. The main idea has been to take advantage of idle times and to allow the DNN layers to work in parallel on different inputs.

As illustrated in Figure 4.1a, the key principle of this research is that each layer in a DNN may function as a pipeline stage, allowing them to elaborate on several inputs simultaneously. It means that the instruction-level parallelism of a CPU pipeline scenario, becomes an image-level parallelism (Figure 4.1b). To enable this parallelism, the architecture of the proposed fault injector, highlighted in Figure 4.2, consists of three vertically stacked levels (from the bottom):

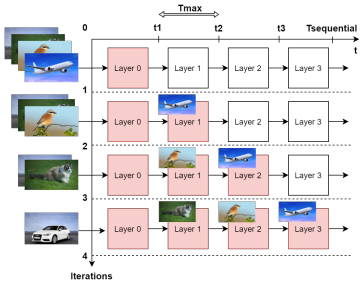
1. The lower abstraction level;
2. The application level;
3. The synchronization level.

The bottom level includes the hardware model of the platform running the DNN under assessment, either at the RTL or gate level representation, and the Fault Injection Unit (FIU). The physical device is not physically available, but the design is provided as HDL. Note that the target hardware may be an embedded device, a GPU, a ASIC, for which the reliability evaluation is needed. The FIU is the unit in charge of introducing faults in the RTL device.

On top of the lower level, there is the application one, where the DNN application runs. It can be implemented in a high-level language (e.g., C or Python). At this level, the framework allows splitting the inference in several stand-alone blocks corresponding to the layers of the neural network. Each layer must communicate with the lower and the upper level and, to this end, the input and the output neurons



(a) The Pipeline concept applied to Neural Networks



(b) Image-Level Parallelism

Figure 4.1 Exploiting the pipeline technique to reduce the DNN simulation time.

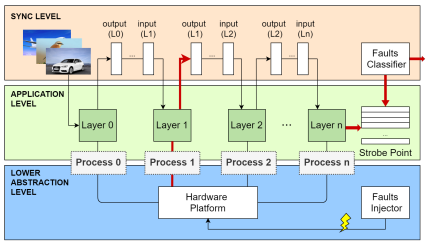


Figure 4.2 Proposed Pipelined Multi-Level Fault Injector Tool

are extracted for each layer. At the lower level, layers are simulated as individual application processes. By coordinating the running of the distinct stand-alone layer simulations, this architecture allows the higher level to obtain control over the inference.

The purpose of the synchronization level is to coordinate the activity of the DNN layers and, overall, advance the inference process. In this level, the following units are included: the database holding the DNN test set, and the Fault Classifier Unit (FCU). The latter evaluates the impact of the injected fault on the network by observing, at the application level, the strobe point (i.e., the results of the inferences). The synchronization mechanism is illustrated in Figure 4.2 with a red arrow linking the hardware low level with the output of L1.

To work at full capacity, the proposed fault injector requires an initial Set-Up Time (T_{setup}) which is computed by multiplying the amount of layers or stages (x) by the simulation time of the "slowest" one (T_{max}), i.e., the most computational-intensive (Equation 4.1). As an example, in Figure 4.1b, x is equal to 4 and T_{max} corresponds to the HDL simulation time of Layer1. Once the T_{setup} is done, the

pipeline works at full capacity and the tool is able to provide an inference result every T_{max}. This time is much slower than the sequential time $T_{sequential}$. Therefore, as described in Equation 4.2, the time required to run a pipelined fault injection (T_{pipe}) with y input images is computed by adding the initial T_{setup} with the inferences of the (y-1) images. Without resorting to the proposed pipelined framework, the simulation time (T_{no_pipe}) is computed as described in Equation 4.3.

$$T_{setup} = x * T_{max} \tag{4.1}$$

$$T_{pipe} = T_{setup} + (y-1) * T_{max} \tag{4.2}$$

$$T_{no\_pipe} = y * T_{sequential} \tag{4.3}$$

4.2 Experimental Results

To prove the effectiveness of the proposed fault injector, a system including an open-source Parallel Ultra-Low-Power (PULP) hardware platform [98] running a CNN used for image classification has been exploited. The selected hardware device includes a cluster of 8 RISC-V cores together with a classic microcontroller for communication and security purposes. Moreover, it includes a Hardware Convolution Engine (HWCE) for running DNN-based applications. The CNN running on it has been trained on CIFAR-10 and the code relies on the PULP-NN open-source library [99]. PULP-NN incorporates a full set of kernels and utilities to support the inference of quantized neural networks (8,4,2 and 1-bit) on a RISC-V based processor.

Using a commercial HDL simulator program from Mentor Graphics, the time of a single inference simulation at RTL has been calculated given the mentioned CNN operating on PULP. It corresponds to 25 minutes ($T_{sequential}$). Next, for each single layer the simulation times have been extracted: the most computational-intensive layer establishes the speed of the proposed tool (i.e., the throughput of the pipeline). In our case study, it turns out to be Layer0 (CV1) and T_{max}=$\sim$10 minutes, as shown in Table 4.1. The pipelined multi-level fault injector was built based on these per-layer analysis. More in details, the last four layers (MP2, CV3, MP3, FC) have been joined in a single pipeline stage to reduce the T_{setup} and to optimize the throughput of the pipeline. Hence, the pipelined fault injector framework is made up of 4 layers and

Table 4.1 Pipelined Fault Injector timing details.

Neural Network			**Fault Injector Framework**		
Layer	**Name**	**Sim Time (min:sec)**	**Layer**	**Name**	**Sim Time (min:sec)**
Layer 0	CV1	10:11	**Layer 0**	CV1	10:11
Layer 1	MP1	4:00	**Layer 1**	MP1	4:00
Layer 2	CV2	9:39	**Layer 2**	CV2	9:39
Layer 3	MP2	1:19	**Layer 3**	MIX	8:04
Layer 4	CV3	4:23			
Layer 5	MP3	1:07			
Layer 6	FC	1:38			

its T_{setup}=$\sim$40 min. After T_{setup}, every T_{max}=$\sim$10 minutes, the framework provides the results of an inference, working at full capacity.

Table 4.2 A comparison between the performances of the sequential fault injector wrt the pipelined.

Layers (x)	**Images (y)**	$\mathbf{T_{no\_pipe}}$	$\mathbf{T_{pipe}}$	**Time Reduction**
4	100	$\sim$42 h	$\sim$17 h	60%

The synchronization level buffers and synchronizes the results of the other layers if they finish their calculation before T_{max}. Therefore, to demonstrate the performances and the gains in terms of simulation time, a comparison is made between a full sequential single fault injection with a pipelined one. The experimental results are shown in Table 4.2. As can be seen, a significant time reduction is achieved; the proposed methodology reduces sequential fault injection timings by 60%. Clearly, such an advantage comes at a cost in terms of memory use. The RAM memory usage goes, on average, from 2.1 up to 4.7 GB for running y inferences with the sequential fault injector. On the contrary, the pipelined uses from 8.4 up to 18 GB for the same set of y images, although for a shorter time.

Next, to validate the proposed tool, a total of 10,000 RTL injections campaigns have been executed by injecting permanent faults (i.e., stuck-at faults) on the PULP platform running the targeted CNN. A total of 100 permanent faults have been placed both on CPUs and randomly distributed over the multi-core system (SYS). Specifically, for each fault, the inference of 100 images from CIFAR-10 dataset have been run.

Table 4.3 Fault Injection Results

	Detected [%]					Masked [%]
	SDC-1	SDC-10	SDC-20	Hang	Crash	
CPU	2.06	8.12	9.84	21.82	38.14	20.02
SYS	3.96	2.02	5.96	14.02	26.16	47.88

Then, faults have been classified depending on their effect and according to the ranking proposed in [43]. A fault is *Detected* whether one of the following situations occur:

- **SDC-1**: A Silent Data Corruption (SDC) failure is a deviation of the network output from the golden network result, leading to a misprediction. Hence, the fault causes the image to be wrongly classified.
- **SDC-10%**: The faulty network correctly predicts the result but assigns a confidence score which varies by more than +/-10% of its fault-free execution.
- **SDC-20%**: The faulty network correctly predicts the result but assigns a confidence score which varies by more than +/-20% of its fault-free execution.
- **Hang**: The fault causes the system to hang and the HDL simulation never finishes.
- **Crash**: It is the opposite situation of the previous one. The HDL simulation immediately stops as a consequence of the fault.

To deal with the *Hang* category, i.e., faults producing an infinite loop, the framework has been equipped with a Timeout. If the framework, after T_{setup}, does not yield any inference results, the fault is classified as detected by system hanging. On the other hand, a fault is *Undetected* and classified as **Masked** whether the network prediction does not belong to any of the previous described classification and the output results are equal to the fault-free execution ones. Experimental results are shown in Table 4.3.

Furthermore, the pipelined multi-level fault injector, as outlined in Table 4.4, allows for a 78% drop in fault simulation time. This gain exceeds the one you would obtain if applying Equation 4.2 (it has been computed, and it corresponds to $\sim$1,716 h). This improvement is due to all the faults belonging to the Hang and Crash categories, that, once discovered at the beginning of the pipeline process, avoid the

Table 4.4 Sequential Framework vs Pipelined Multi-Level Framework

Sequential Framework	**Total Injections**	**Tsequential [min]**			**y**	**Duration [h]**
	10,000	25			100	$\sim$ 4,166
Pipelined Multi-Level Framework	**Total Injections**	**Tmax [min]**	**Tsetup [min]**	**x**	**y**	**Duration [h]**
	10,000	10	40	4	100	$\sim$ 881

fault simulation of the entire image set for that fault. It is important to underline that this mechanism has not been implemented in the sequential framework.

4.3 Chapter Summary

In summary, a pipelined multi-level methodology for analysing the resilience of DNNs operating on a target device is presented in this research. It came from the need to supplement pure software techniques with lower-level approaches that encompass the hardware description of the target platform as well. Indeed, software-based injection approaches can only target a restricted number of elements such as weights, biases, and input images. A hardware-level approach broadens the range of options for evaluating DNN-based system reliability. At the same time, it is known that the lower the abstraction level, the higher the simulation time. To address this issue, this research presents the architecture of a fault injector that is able to drastically reduce the simulation time at RTL by mimicking the flow of a pipelined process. Thanks to this, it was experimentally demonstrated that the inference time reduces by about 78% in our case study.

Future research will focus on enhancing the capability of the pipelined multi-level framework to handle other types of fault models, such as transient faults or Multiple Bit Upsets (MBUs). Second, by adopting a multiprocess environment, for example, further decreasing HDL simulation time.

Chapter 5

Reliability Assessment at the Physical Level

5.1 Proposed Approach

Recent studies have demonstrated that hardware faults induced by external perturbations (i.e., in a harsh environment) can significantly jeopardize the correct functionalities of neural networks, leading to CNNs prediction failures [100, 50]. It starts to be crucial to guarantee a high reliability level of CNN-based systems, due to their widespread use in various fields (e.g., automotive, robotic, avionic).

Fault Injections (FIs) have long been known as appealing methods for assessing the reliability of systems under test among all available testing approaches. They can be carried out at very different abstraction levels: in simulation (at the software or architectural level), or physically by means of radiation tests. Above all, it is worth mentioning their principal characteristics, along width their main advantages and disadvantages. Radiation-based FIs expose the system implementation to the same external conditions with respect to the in-field application, and therefore they guarantee a precise reliability assessment. However, they are expensive in terms of hardware resources, facility access and exposure time. On the contrary, software-based FIs modify the behaviour of the software to simulate hardware fault occurrences. The cost is lower compared to physical-based FIs, because they do not require purchasing specific electronic devices to run the tests, with a higher

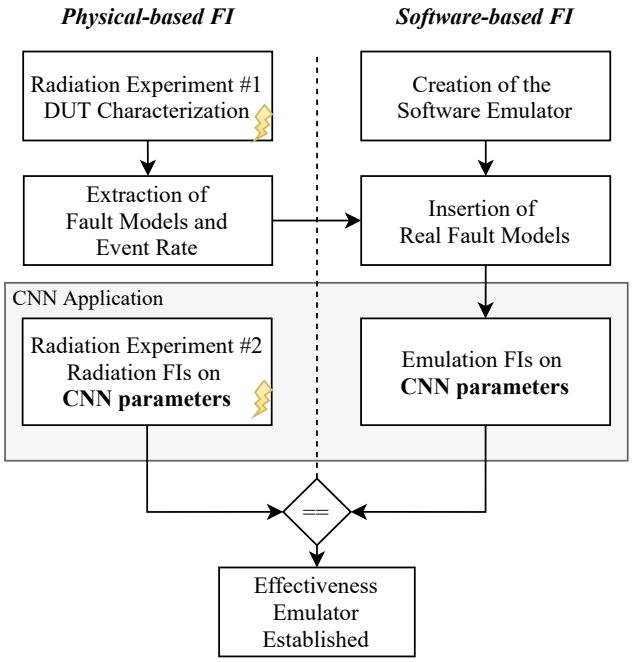


Figure 5.1 Diagram of the Proposed Approach.

degree of controllability[1] and observability[2]. However, the precision assessment of a software-based fault injection strictly depends on the adopted fault models.

This work [54] presents a reliability assessment methodology which is able to achieve a reliability assessment precision close to radiation test experiments with the flexibility of a software-based fault injector and related costs. It describes the architecture of a software emulator which is configured to inject real faults retrieved from radiation tests on neural networks. Since the occurrence of such faults is independent of the running application, the designed emulator can reproduce their incidence and access their effects on CNN applications with high fidelity. A diagram of the proposed flow is shown in Figure 5.1.

From the device characterization of a DUT in realistic environmental conditions (through an accelerated neutron beam), event rates and fault models are extracted. These are used to feed a software emulator capable of introducing such faults on the parameters of CNNs (i.e., weights and biases). In parallel, radiation FIs are conducted by targeting the DUT storing the CNN parameters, to establish the effectiveness of the methodology. The results evidence the correlation between the physical-based

[1]The ability to control where and when a fault is injected.
[2]The ability to identify internal events within the system circuit (not only primary output).

FIs and software-based ones. The proposed methodology will be detailed in the following sections.

5.1.1 Radiation Experiments

The radiation tests have been conducted in the United Kingdom at the Rutherford Appleton Laboratories, where an atmospheric-like neutron spectrum is delivered in the ChipIr beamline at the second target station of the ISIS neutron source. We can split the radiation experiments in two different phases: the DUT characterization and the radiation tests on the targeted CNNs. They are detailed in the following subsections.

5.1.1.1 DUT Characterization

The DUT is the S27KS0642GABHI020, a 64 Mib HyperRAM self-refresh DRAM manufactured by Cypress Semiconductor. In [50], this memory device was characterized under an atmospheric-like neutron beam, by performing static and dynamic tests. The former apply known patterns, and after radiation exposure, a readback operation is performed to collect radiation-induced faults. The latter is done by means of known memory tests algorithms like March C-, Dynamic Stress, Dynamic Classic, and mMats+. In the end, three different fault types were identified: Single Bit Upsets (SBUs), Stuck-at Bits, and Block Errors (BE). Among these, the most worrying is the block error event due to its extension. Block errors affect different regions of the memory and can derive by a temporary malfunction of the sense amplifier or register that serves a column of the affected memory addresses. Four different patterns have been identified, with intermittent word errors in vertical and horizontal sequential addresses, affecting up to 2,048 addresses with the vertical pattern and 1,024 addresses with the horizontal pattern. Given the results of static and dynamic tests, it was possible to (i) evaluate the memory sensitivity to the mentioned faults, (ii) compute the event cross-section (σ), and (iii) compute the Soft-Error Rate (SER). Regarding the event cross-section (σ), it is important to distinguish between the cell-related (SBUs and stuck-at bits) and device-related (BEs) faults. In the first case, it is computed as:

$$\sigma_{\text{bit}} = \frac{N}{F \times M} \tag{5.1}$$

where N is the number of events, F is the cumulative fluence in particles/cm^2, and M is the number of bits [101].

On the contrary, since BEs are related to the control logic of the device, the cross-section is defined as:

$$\sigma_{\text{device}} = \frac{N}{F} \tag{5.2}$$

where the memory size is removed from the equation, and the cross-section is device-based.

Finally, the Soft-Error Rate (SER) for SBUs and stuck-at bits is defined as:

$$SER_{FIT/Mb} = \sigma_{bit} \times (1,024 \times 1,024) \times 10^9 \times \phi \tag{5.3}$$

where $1,024 \times 1,024$ (bits) is the Mb coefficient, 10^9 is the Failures in Time (FIT) definition, and ϕ (13 particles/cm^2/h) is the neutron energies' ($>$ 10 MeV) flux at New York (sea level) outdoors for a mean solar activity. Concerning the SER for BEs, the Mb coefficient is removed from the equation, and it becomes:

$$SER_{FIT} = \sigma_{device} \times 10^9 \times \phi \tag{5.4}$$

5.1.1.2 Radiation Tests on CNNs

Radiation tests on CNNs applications have been performed to validate the effectiveness of the software emulator. To this end, the same CNNs architectures have been selected for the assessment and the same portion of data has been corrupted in both procedures. More in details, based on the same LeNet-5 architecture [4], three different CNNs with different data types (32-bit Floating Point, 16-bit Integer, 8-bit Integer) have been used: *Float 32*, *Int 16*, and *Int 8* CNNs. They have been trained by using the open-source framework N2D2 [33] on MNIST dataset and, on the test set (10,000 images), their accuracy was equal to 99.07%, 99.05%, and 99.05% respectively. Next, the same N2D2 framework was used to export the trained networks as C code using the three different data representations. Concerning *Int 16* and *Int 8* CNNs, weights and biases are quantized after the training.

As shown in Figure 5.2, during the radiation tests, only the HyperRAM memory was under an atmospheric-like neutron beam (our DUT). The system is a Xilinx Zynq-7000 SoC with an ARM Cortex™A9 processor attached to a 28 nm Artix®7 FPGA. Three external memories were used to deploy the CNN application on this embedded device: two units of the MT41K128M16JT-125, which is a 2 Gb SDRAM DDR3L from Micron Technologies, and the HyperRAM memory (our DUT). Furthermore, to enhance the reliability of the system, the SoC configuration memory (CRAM) was monitored by the commercial Xilinx scrubber, the Soft Error Mitigation (SEM) core, which reports detected SBUs, and, when possible, corrects them [102].

The memory layout for an application is usually split in sections: the *text* section that includes executable instructions; the *data* section includes constants and statically allocated variables, *heap* includes dynamically allocated variables, and the *stack* stores parameters for function calls, return addresses, and local variables. In the previous research project [50], the same DUT (the HyperRAM) stored all data related contents (i.e, *data* and *heap*). In this study, only the *.rodata* (read-only data segment) section of the code was allocated into the HyperRAM memory (and so under radiation) to restrict the source of errors. This section holds static constant data and is mainly composed of CNN layers' weights and biases. This section uses approximately 505 kB in the *Float 32* CNN, 273 kB in *Int 16* CNN, and 154 kB in *Int 8* CNN. The choice of this setup enabled a higher capability analysis by restricting the source of errors (only the DUT was exposed to the beam).

Henceforth, run is defined as the inference of a set of images (e.g., 2,000 or 1,000 images), beginning with image '0' and ending with the final image inside the dataset or when the execution is terminated. This is due to the fact that certain runs did not complete their execution due to a functional interruption that did not impact the DUT (HyperRAM), but directly affect other components of the system. Only the processed images within a run are taken into account in these cases. For the sake of clarity, it is necessary to better specify the reasons behind the incomplete runs. The radiation experiments were performed under an atmospheric-like neutron beam, and the ionization in silicon was not directly generated by the neutron interaction with the matter but through neutron-induced silicon recoils and/or nuclear reactions byproducts. The created free charges of electron-hole pairs generated from the neutron events might lead to single-event effects in the devices [103, 104]. For example, the scattering of a neutron by collision with an atomic nucleus will lead to a neutron scattering at an angle as well as the nucleus recoils [105]. In this case,

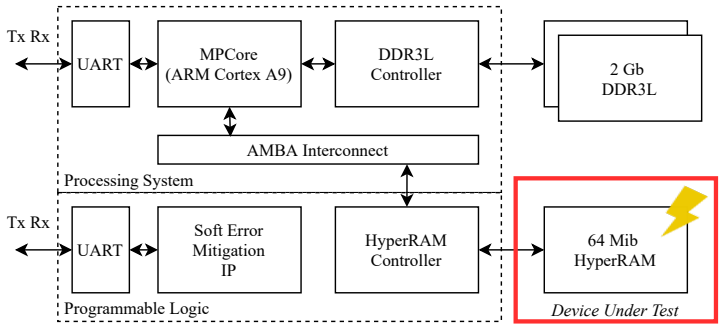


Figure 5.2 Top-level diagram of the system. The DUT is highlighted.

some scattered particles may hit not only the device under test, but also some other electronic components of the system that are located in the irradiation room. Then, considering that the execution time of the *Float 32* version is higher when compared to the other versions (*Int 8* and *Int 16*), and adding it to the time needed to complete the 2,000 inferences, the probability to have some particles hitting other parts of the setup electronics increases within a run with these characteristics.

During the initial phase of the test, we target, for each CNN under test, the inference of a total of 2,000 images per run. At the end of each run, a readback operation of the content of the *.rodata* is done to identify possible SBUs, stuck-at bits, and block errors. In a second phase, to cope with the above-mentioned functional interruptions of the system due to single-event effects originated by free charges of electron-hole pairs, the dataset was reduced to 1,000 images, to raise the likelihood of having complete runs. The total number of inferences were 44,370 for *Float 32*; 57,680 for *Int 16*, and 52,595 for *Int 8*. Experimental results are discussed in Section 5.2.1.

To obtain the expected number of events within a run, based on the Soft-Error Rate, we define the Execution Soft Error Rate (E-SER) as:

$$E-SER_{\text{ SBU | stuck-at}} = \sigma_{\text{bit}} \times M \times \bar{\phi} \times t \tag{5.5}$$

where σ_{bit} is the calculated cross section for SBUs or stuck-at bits, M is the memory size in bits used by the application (stored in the DUT), $\bar{\phi}$ is the average neutron flux during the test campaigns, and t is the application execution time in seconds. The same reasoning is applied to define the E-SER related to BEs. However, since the

Table 5.1 Estimated event rate for both test scenarios.

Test Scenario	Dataset Size	CNN version	E-SER [events/run] SBUs	Stuck-at bits	Block Errors
1	2,000 images	Float 32	1.53	0.79	0.59
		Int 16	0.48	0.25	0.34
		Int 8	0.17	0.08	0.21
2	1,000 images	Float 32	0.84	0.43	0.32
		Int 16	0.24	0.12	0.17
		Int 8	0.08	0.04	0.10

BEs are device related, the rate is defined as:

$$E - SER_{\text{BE}} = \sigma_{\text{device}} \times \bar{\phi} \times t \tag{5.6}$$

From the E-SER definition, it is possible to calculate the expected event rate based on the presented radiation tests. Table 5.1 provides the calculated values for every CNN under study.

5.1.2 Software Emulator

From the device characterization of a DRAM memory, three types of faults have been extracted (SBUs, stuck-at bits, and BEs) together with event rates. Since their occurrence was independent of the running application, a software emulator was designed to reproduce their incidence on CNN applications. It is illustrated in Figure 5.3. To mimic the process of the radiation campaigns and the neutron flux, it is structured with a multi-threading structure. Indeed, together with the main process, it makes use of two threads: the inference (*thread1*) and the injector (*thread2*). Once created, they run independently to fulfil their purposes. The task of the injector thread is to introduce faults in memory locations, while the goal of the inference thread is to run a given number of inferences (N). Threads can share the same resources, such as memory locations, even if they exist as independent entities within a process. For example, the CNN parameters (weights, biases, and images) are reachable by all threads. Therefore, to avoid parallel read and write of shared data, a synchronization method is used based on mutexes to guarantee safe communications. For the sake of clarity, when *thread2* wants to inject a memory fault (for example, a stuck-at 0 in a weight), it acquires the mutex, injects the fault, and then releases it. While the

mutex is being taken, *thread1* is not authorized to read the shared memory, causing the inference process to stall.

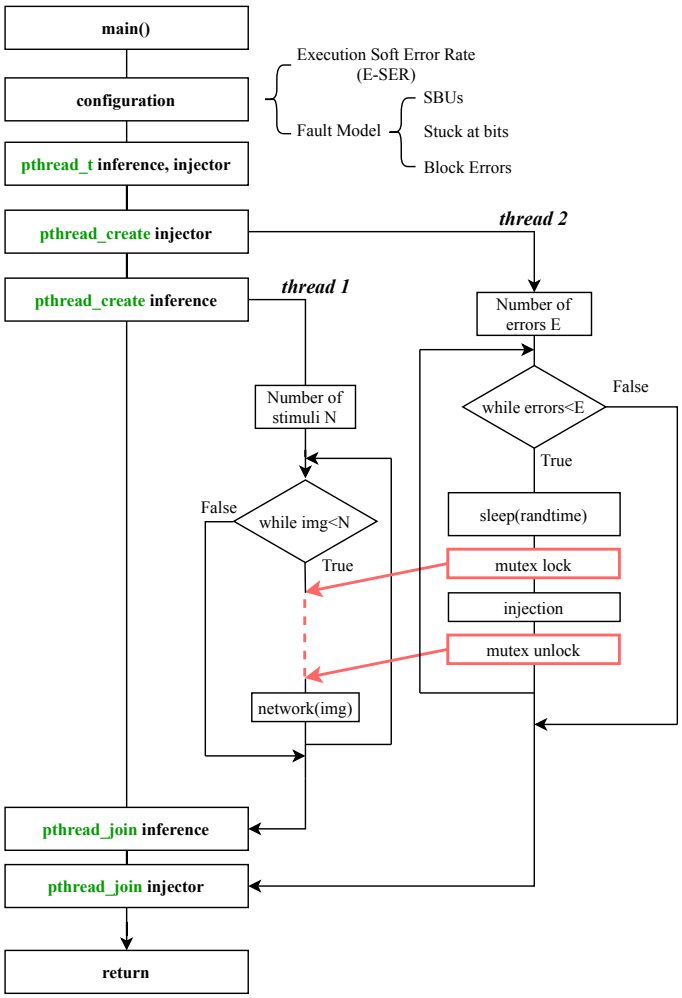


Figure 5.3 A diagram of the proposed emulator.

Furthermore, to get realistic injections and, above all, to emulate a neutron flux, the injection frequency is not fixed: *thread2* introduces faults with a random frequency driven by the *randtime* variable. Until this time has elapsed, the injector thread sleeps.

To work properly, the emulator is first configured to deal with a specific E-SER and a specific fault model (i.e., SBUs, stuck-at bits, BE), as shown in Figure 5.3.

For SBUs and stuck-at bits the faulty locations are chosen randomly, and the injection time is tuned according to the E-SER. On the other side, replicating block errors necessitates the use of a particular parser. Indeed, a BE is injected into the

application parameters only if the virtual addresses of the software application match with the physical addresses returned from the radiation test campaigns where the targeted BE was found.

For the sake of clarity, at the end of the radiation test campaign, faults in the HyperRAM memory are reported specifying, along with other information, the physical address affected by the fault and the corresponding error data. Then, knowing the virtual addresses of the CNN application, a parser was created and included in the architecture of the emulator to deal with BEs injection: if there is a match, BEs are injected in the corresponding memory addresses of the CNN application. It can inject in both weights and biases, *the same portion of data allocated in the rodata memory section under radiation.* Experimental results and analysis are presented in Section 5.2.2.

5.2 Experimental Results

In this section, the experimental results are presented and the effectiveness of the methodology is validated through accurate analysis and comparisons.

5.2.1 Radiation Tests Results

In this section, the principal results of the radiation tests on weights and biases of the three CNNs (i.e., *Float 32*, *Int 16*, and *Int 8*) are presented. In the two test scenarios (i.e., 2,000 images and 1,000 images from the MNIST dataset), the golden accuracy of the CNNs was the same: 99.15% and 99.30%, respectively. From the total runs under radiations, only the 57.96% were completed, and, among these, only the 6.37% presented a difference in the final accuracy. To optimize the use of the beam time, also incomplete runs were considered for the analyses: indeed, data related to each inference are independent, so all the data coming up to the last non-failing inference can be a good candidate for our purposes. In Table 5.2 a summary of the faulty runs is given, by presenting the first image with a faulty inference and the final accuracy. Note that all the runs with the *Int 8* version returned the expected values, while data in Table 5.2 show degradation in the *Float 32* and *Int 16* versions.

Table 5.2 Summary of the runs that return a faulty accuracy at the end of the radiation tests on CNNs.

Faulty run	CNN version	Dataset size	First faulty inference [image no.]	Run accuracy [%]	Expected accuracy [%]
1	Int 16	2,000	1,345	99.10	99.15
2*	Float 32	2,000	1,384	84.36	99.09
3	Int 16	2,000	583	99.10	99.15
4*	Float 32	2,000	120	19.96	99.24
5*	Float 32	1,000	720	98.91	99.24
6*	Int 16	1,000	988	98.00	99.30
7	Int 16	1,000	102	99.20	99.30
8	Int 16	1,000	720	93.60	99.30
9	Float 32	1,000	673	70.40	99.30
10	Int 16	1,000	189	99.50	99.30

* Incomplete runs.

Very interesting, in one case radiation-induced faults lead to an increase in the number of correct inferences (faulty run number 10, Table 5.2): the final accuracy increased from 99.3% to 99.5%. This is a fascinating scenario that demonstrates how CNN resilience evaluation differs from traditional applications. The point is that CNNs are always an approximation of a function, in this case, a classifier. As a result, a fault might cause the CNN to behave differently, not just from good to wrong prediction (which is the majority of the time), but also from wrong to good prediction. A further interesting outcome is observed in faulty run number 5: starting from image 720, the output vectors returned all the values as a floating-point exception (NaN, "Not a Number"), leading to the invalidation of the top-1 scores. This example is fascinating since it is not a conventional example of silent data corruption (that is, by definition, non-detectable). Worthy of note is faulty run number 9, where, the output vectors, starting from the first faulty inference, return always the same wrong value for all subsequent inferences, having all the digit *3* as a result of the top-1 score. Similar behaviour was observed for faulty runs number 2, 4, and 6. An additional analysis is needed for faulty run number 7: the output vector had the correct top-1 score, but the output application returned a faulty inference, giving a digit '0' instead of '2'. This may be due to a register or a variable allocated in the external memory (DDR3). The remaining faulty runs did not present particular behaviours and were caused by SBUs, stuck-at bits, and BEs.

5.2.2 Software Emulator Results

Fault injections with the software emulator aim at reproducing the same scenarios observed during the radiation test campaigns. Indeed, it can introduce faults in CNNs parameters, and this reproduces the configuration of the radiation test where only the *.rodata* section is under test. The fault injection procedure was guided by the E-SER defined in Section 5.1.1.2 and whose values are reported in Table 5.1 (hereinafter referred to as *nominal* E-SER, i.e., 1x). To investigate worst-case scenarios, those values are tuned and amplified by 25 times (25x), up to 100 times (100x). This was useful to evaluate the CNNs resilience when a growing number of SBUs, stuck-at values, and BEs affect the CNN application.

SBUs and Stuck-at Bits: In the following, we will first focus on the results and the analysis of the injections related to SBUs and stuck-at bits. From the implementation point of view, SBUs are injected through bit-flips and remain active only during a single run; stuck-at bits as a permanent '0' or '1' in the targeted fault location and accumulate over the runs. Detailed figures of injected SBUs and stuck-at bits for each CNN are given in Table 5.3. Considering *Float 32 CNN*, Columns 2 and 3 from Table 5.3 report the number of SBUs and stuck-at injected during each run. On the other hand, Column 5 presents the total amount of stuck-at faults that have been injected at the end of all runs, because they are accumulated over the runs. For each E-SER value (Column 1), the experiment was reproduced 50 times (50 runs/E-SER) with SBUs or stuck-at bits corrupting random layers, fault locations (weights or biases), and bit positions. The same reasoning is applied to the *Int 16* and *Int 8* CNNs. It is worth reminding that: a run is defined as the inference of a set of 1,000 images from MNIST, second, a golden run over this set achieves the 99.30% accuracy for all the CNNs.

Concerning the nominal E-SER (i.e., the row *1x*), since injecting a non-integer number of faults during a run was not feasible, an approximation was done as follows:

- *CNN Float 32*: For SBUs, one single random fault was injected during a single run; for stuck-at bits, one single fault was injected every 2 runs.
- *CNN Int 16*: For SBUs, one single random fault was injected every 4 runs; for stuck-at bits, one single fault was injected every 8 runs.

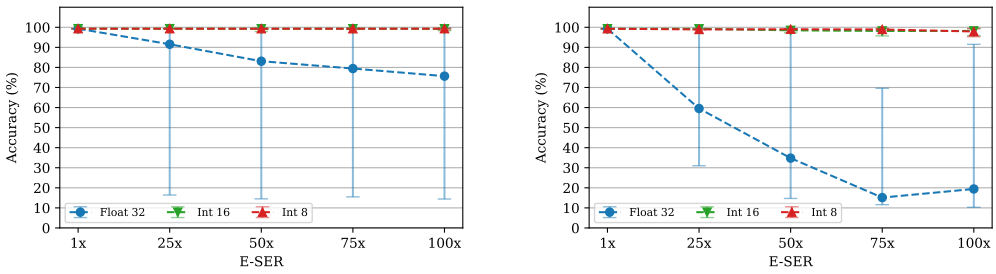
Table 5.3 Details of SBUs and stuck-at bits injection for the *Float 32*, *Int 16*, *Int 8* CNNs with an increasing E-SER: 1x, 25x, 50x, 75x, 100x

E-SER	Float 32 CNN			Int 16 CNN			Int 8 CNN		
	SBUs/run	Stuck-at bits		SBUs/run	Stuck-at bits		SBUs/run	Stuck-at bits	
		Stuck-at/run	Total		Stuck-at/run	Total		Stuck-at/run	Total
1x	0.84	0.43	25	0.24	0.12	6	0.08	0.04	2
25x	21	11	550	6	3	150	2	1	50
50x	42	21	1,050	12	6	300	4	2	100
75x	63	32	1,600	18	9	450	6	3	150
100x	84	43	2,150	24	12	600	8	4	200

- *CNN Int 8*: For SBUs, one single random fault was injected every 16 runs; for stuck-at bits, one single fault was injected every 32 runs.

Figures 5.4a and 5.4b report the experimental results of the FI campaigns performed with the proposed emulator. For the three CNNs under assessment, the average value achieved during the 50 runs is represented, together with the minimum and the maximum value (the small error bars). It is apparent that, the less resilient CNN to the occurrence of SBUs or stuck-at bits is the *Float 32*: as the number of faults increases, its average accuracy considerably reduces. This first outcome is in line with the radiation test results, where, overall, *Int 16* and *Int 8* show higher resilience. Focusing on the nominal E-SER (*1x*), data show that the final accuracy for the three CNNs is not affected by stuck-at bits, staying at 99.30%. When it comes to the *Float 32* CNN, only one run over 50 presented a slight degradation (the final accuracy was equal to 99.29%). A similar scenario was observed during the radiation tests: only 1 complete run over 17 was faulty, despite the occurrence of SBUs or stuck-at bit faults. On the contrary, the two integer CNNs (*Int 16* and *Int 8*) keep the golden accuracy (99.30%) over the 50 runs. Indeed, the accuracy degradation that we observe during radiation tests for the *Int 16* in Table 5.2 is related to the occurrence of block errors.

Moreover, it is worth adding that the accuracy of the *Float 32* CNN considerably decreases as the E-SER increases. On the contrary, the resulting accuracy value is slightly influenced by increasing the E-SER for *Int 16* and *Int 8*: only with a E-SER *100x* a small degradation (respectively 1% and 2%) is noticed when they are affected by stuck-at bits. Overall, it is evident that stuck-at bits are more critical than SBUs, and this is reasonable, since they are permanent faults and therefore, once they appear, they accumulate over the runs.



(a) SBUs

(b) Stuck-at bits

Figure 5.4 Accuracy variation based on the increase of the E-SER with SBUs or stuck-at faults for the three CNNs.

A last interesting observation in the *Float 32* CNN is associated to the scenarios *75x* and *100x* for stuck-at bits. Indeed, the *100x* injects a greater number of stuck-at bits and the average accuracy is higher than *75%*. This outcome implies that the choice of faulty bit locations (random in our experiments) is crucial in the final accuracy measurement. With an in-depth analysis, it turned out that faults affecting the exponent bits are the most critical, leading in many cases to NaN ("Not a Number") values, in line with the state-of-the-art knowledge [43, 83].

Block Errors: Radiation tests on the HyperRAM memory revealed the presence of block errors, i.e., vertical or horizontal sequential addresses of faulty cells in the memory. This is a known effect in the literature, and it is well documented in [106]. A total of 37 BEs have been identified in our DUT at the end of the readback operations. Knowing the virtual addresses of the CNNs application, and their physical addresses on the board, it is possible to inject exactly the same faults via software on the same piece of data. To this end, the software emulator is equipped with a parser that looks for this correspondence. Contrary to SBUs and stuck-at bits, where the faulty locations are randomly generated, for BEs faults are injected as detailed by the fault lists retrieved from radiation tests' reports. As a result of the readback operation, a list of faulty addresses in memory is returned, as well as their expected content and their respective faulty one. This log is converted in a fault list that is used by the software emulator to inject stuck-at bits in the corresponding virtual addresses of the CNN parameters.

For each CNN application, Table 5.4 reports the number of BEs matching the addresses, the total faults that can be introduced into the CNN parameters, and

the memory footprint of the given CNN application in the HyperRAM memory. Moreover, in Figure 5.5 the following information are given:

- y-axis: The number of injected faults for each single BE. As emerging, their incidence can also vary from many faults injected (e.g., BE29 with 972,960 in the *Float 32* CNN) up to few ones (e.g., BE1 and BE2 with only 8 faults in the *Int 8* CNN).
- x-axis: The impact in terms of accuracy of the specific BE on the targeted CNN when it occurs during a run in a random time. The experiment has been repeated 50 times: therefore, in the graph the average accuracy is depicted with the minimum and the maximum value reached (the error bars).

Table 5.4 BE Injection Details

CNN Application	**Total Injected BEs**	**Total Injected Faults**	**Memory Footprint (kB) *(.rodata)***
Float 32	17	3,278,656	505.81
Int 16	11	1,714,007	273.17
Int 8	15	605,922	154.11

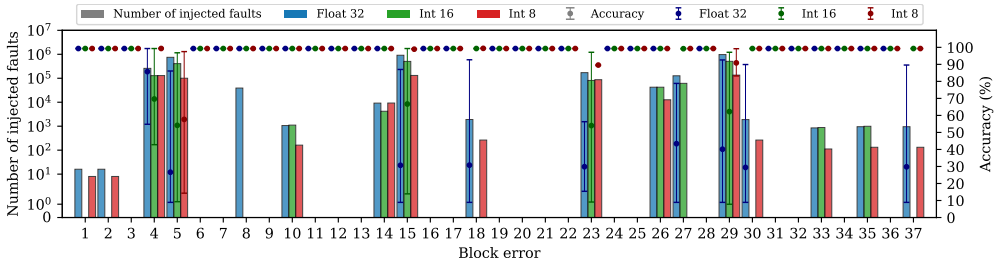


Figure 5.5 Number of injected stuck-at faults for each single BE for each CNN application (y-axis on the left). Average accuracy value for each single BE affecting a run at a random time (y-axis on the right).

Worthy of note in Figure 5.5 is the minimum value: it is calculated at time zero (before running the 1,000 inferences). As shown, it leads an accuracy lower than 10% in ten cases (seven in *Float 32* and three in *Int 16*). It is apparent that the impact of the BE is dependent on the injection time. Because the overall accuracy is determined by the outcome of 1,000 inferences, fault injection closer to the start of the run will have a greater influence on the final accuracy, and vice versa.

Furthermore, by analysing the outcome of the software fault injections for each single inference, it turned out that, in particular cases, once a BE is injected during a run, the CNN starts predicting *all* the subsequent images wrongly. For this reason, we define them as critical. This scenario was observed in the following injections:

- *Float 32*: BE5, BE15, BE18, BE27, BE29, BE30, BE37
- *Int 16*: BE5, BE15, BE23, BE29
- *Int 8*: BE5

The listed BEs critical for the *Float 32* CNN inject stuck-at-1s in the exponent part of the floating point representation, in one or many bits of the corrupted weights and/or biases. This produces several floating point exceptions (Not a Number) that propagate through the network and corrupt all the subsequent computations. This was observed for the faulty run number 5 during the radiation test campaigns, as detailed in Table 5.2. The BEs that in *Float 32* produce a degradation in the final accuracy (Figure 5.5) but are not included in the list of critical do not generate NaN numbers. As an example, BE4 and BE23 inject stuck-at-0 values, and, after their injection, the CNN is still able to yield some correct prediction. The leftovers do not affect the final accuracy at all (staying at 99.30% despite the injection). This is due to the following: they do not inject any faults into the *Float 32* parameters, or they inject stuck-at faults into the mantissa part of the floating-point values, which do lead to a change in the final CNN prediction [83].

Concerning the *Int 16*, we can identify two scenarios: first, BE15 and BE23 inject respectively stuck-at-1s or stuck-at-0s in all bits of the matching weight/bias addresses. After their injection, the output vectors of the subsequent images assumes fixed values (all 0xffff for BE15 and all zeros for BE23). Second, BE5 and BE29 are also considered critical because fixed random values are observed in the output vectors after their injection (contrarily to BE15 and BE23, we do not observe specific patterns). Next, as shown in Figure 5.5, also the occurrence of BE4 degrades the performances of the CNN, but, the CNN can still yield some correct predictions after its injection.

The most crucial BE in the *Int 8* application is BE5: when it is injected, the network begins to generate incorrect results. The CNN's output vectors, in particular, take on random values once BE5 is injected. Overall, this CNN is extremely resistant

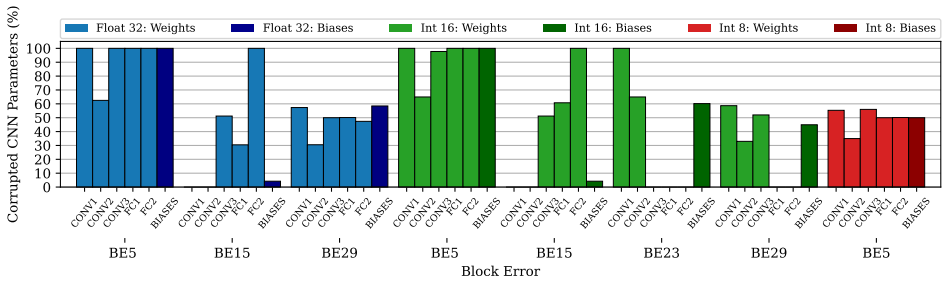


Figure 5.6 The incidence of the most critical BEs on CNNs weights and biases.

to BEs. Only BE23 and BE29 exhibit a modest average decline in the final accuracy value, as seen in Figure 5.5. Despite the inserted stuck-at bits, the remaining BEs had no effect on CNN performance.

While for *Float 32* the origin behind the problem raised by the critical BEs (the corruption of the exponent bits) was clear, for *Int 16* and *Int 8* it needed a further investigation. Indeed, thanks to the emulator, it has been possible to analyse the impact of BEs on CNN's layers and parameters.

In Figure 5.6, the percentages of the corrupted weights or biases matching the physical addresses of the BEs are illustrated for the most critical BEs, for every layer and biases, for every CNN. The reader should note that the critical BEs affecting the CNNs by less than 10% are not represented in the graph (i.e., BE18, BE27, BE30, BE37 for *Float 32*).

The CNNs behaviour may be categorized into three categories based on the inserted BE. As stated before, we define a BE as critical if once it occurs (during the CNNs inferences), all the output vectors containing CNNs predictions may assume the following values:

- Random Values: In all layers and biases, the percent of corrupted weights or biases is less than 100%. As a result, the network continues to transmit incorrect values that are still dependent on inputs (and vary accordingly). This happens for BE5 in *Int 8* and BE29 in *Int 16*. This particular scenario was observed during radiation tests for the faulty run number 8 in Table 5.2: starting from the image number 720, the report of the readback operation shows the vector outputs returning all random values (i.e., wrong predictions).

- Fixed Random Values: The amount of corrupted weights or biases in one or more layers is 100%. Two scenarios exist in this circumstance. First, if a BE

forces a stuck-at-0 in all layer's weights (BE23 *Int 16*, CONV1) the network's output no longer depends on the inputs but just on the values taken by the biases (corrupted at 60% with stuck-at-0s). Due to the occurrence of BE23, only 0x0s are reported in all the subsequent output vectors. This result is identical to the faulty run number 6 (Table 5.2) achieved during the *Int 16* CNN's radiation test. All 0x0s are reported in the radiation test's log. The second scenario is related to those BEs injecting stuck-at-1s in all weights of a layer: in this case, the numbers become negative and they are filtered by the rectified linear unit (ReLU), i.e., the activation function. The dependency on the inputs is removed once more: the output vector will be constant for all subsequent inferences and will only be dependent on the bias values of the final layer. This scenario covers BE5 and BE15 in *Int 16*. Lastly, during radiation tests, fixed random values are returned during run number 9 (Table 5.2) for the *Float 32* CNN.

- Saturated Values: Stuck-at-1s are placed in the exponent part of the floating point representation, leading to NaN exceptions. This class covers all the critical BEs for *Float 32* CNN. The analysis of Figure 5.6 may guide to different conclusions (e.g., if the occurrence of BE5 and BE29 in *Float 32* may lead to *fixed random* values). This is not possible because of the faulty bit positions and the NaN exceptions.

Interestingly, contrarily to what is claimed in the literature [45], these results show a lower resilience values of biases against the occurrence of block errors. If a great number of biases is affected by stuck-at faults, the accuracy of the network is highly influenced, and their higher resilience with respect to weights is not confirmed with block errors. The computations turn into *unbiased* values with the 100% of corrupted biases (e.g., BE5 (*Int 16*) corrupts the 100% of total biases by injecting a stuck-at-1 in the 36% of bits of the matching ones).

Generally, as illustrated in Figure 5.5, the final accuracy of the networks affected by non-critical BEs can also increase: as observed during radiation tests for the faulty run number 10, the final accuracy moderately increases (Table 5.2). Clearly, it can also slightly decrease as a consequence of radiation-induced errors (faulty run number 7, Table 5.2).

Until now, the effects of single injected BEs are discussed. A further experiment was configured to inject multiple BEs during a run, by tuning and increasing the

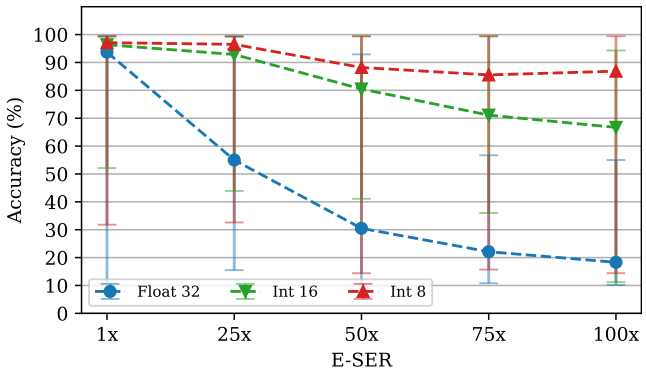


Figure 5.7 Multiple block errors affecting the three CNNs under assessment.

E-SER reported in Table 5.1 for the 1,000 images dataset. The purpose was to assess worst-case scenarios: when a growing number of BEs corrupt the weights or biases of a CNN. The results of the experiments are shown in Figure 5.7, and the details about the E-SER parameters are reported in Table 5.5. As done for SBUs and stuck-at bits, the nominal E-SER (*1x*) was increased up to 100 times (*100x*). As Figure 5.7 illustrates, the final accuracy of the CNNs can vary consistently depending on the injection time, the BE sorting, and the number of injected BEs. This high variation is highlighted by the error bars: the difference between the minimum and the maximum values are non-negligible. Particularly, the minimum was always achieved when the first injected BE belonged to the set of critical ones. To conclude, as shown in the graph, as the E-SER increases, the final average accuracy reduces. Once again, as a confirmation of the radiation tests results, the *Int 16* and the *Int 8* CNNs show a greater resilience against the occurrence of block errors.

Table 5.5 Details of block error injections for the CNNs *Float 32*, *Int 16*, *Int 8* with an increasing E-SER: 1x, 25x, 50x, 75x, 100x

E-SER	**Total Runs**	**CNN**		
		Float 32	**Int 16**	**Int 8**
1x	50	0.32	0.17	0.1
25x	50	8	4	3
50x	50	16	9	5
75x	50	24	13	8
100x	50	32	17	10

5.3 Chapter Summary

This research work presents a methodology to perform fault injections on CNN applications based on real data coming from radiation tests campaigns. It relies on the adoption of a software emulator, which has been designed and developed to deal with different types of radiation-induced faults, and with a configured frequency of injection. First, a characterization phase was performed on a DRAM memory and three types of faults have been identified: SBUs, stuck-at bits, and block errors. Then, event rates have been extracted and used to set up the injection frequency of the software emulator. It is, thus, capable of reproducing real scenarios and injecting exactly the same faults (as for BEs) in CNN applications. The efficacy of the proposed emulator was assessed by means of (i) radiation test campaigns on three CNNs exploiting different data types, and (ii) software injections on the same CNNs' portion of data that was under test during the radiation experiments (i.e., *.rodata*).

By comparing the outcomes of both experiments, it was shown that the software emulator can reproduce the faulty behaviours observed during the radiation tests for all three CNNs. As an example, the same output vectors are returned in both radiation- and software-based FI campaigns, like NaN for *Float 32* CNN, or all zeros for *Int 16* CNN. Moreover, for a more in-depth analysis, the causes of specific behaviours have been investigated thanks to the adoption of the software emulator. Indeed, it was used to get information that cannot be extracted from physical-based fault injections, such as the impact of faults on CNN internal structures (e.g., layers, channel or even kernel) or worst-case scenarios. Additionally, it is known that radiations experiments are extremely costly. In this work, we show that a software strategy could be complementary to physical testing and may allow optimizing time and costs. In other words, we would like to underline that the aforementioned evaluations can not be done only using physical-based FIs, and, at the same time, pure software-based FIs may produce insignificant findings if the injected fault models do not accurately match real-world hardware malfunctioning behaviour.

Finally, the following considerations are worth mentioning. There has been a lot of research into the reliability of CNNs against transient and persistent faults in the literature so far (e.g., SBUs and stuck-at bits). At the same time, it is worth noting that the single-fault assumption is indeed not entirely accurate. Convolutional neural networks, as many other computational-intensive applications, mainly suffer the occurrence of multiple faults. Nevertheless, establishing an effective multiple

fault injection technique remains an open concern, and it is not a simple task: the number of fault locations might rapidly grow. As a consequence, based on the results of radiation testing, we feel that block errors injection may be regarded as a very interesting attempt to conceal numerous defects.

In the future, the software emulator will be used and modified to assess the reliability of applications different from convolutional neural networks or, deeper and modern CNN models, having different memory sections.

Chapter 6

Mitigation Strategies

The intent of this section is to illustrate the mitigation strategies that have been proposed to improve the reliability of AI-based systems. In the first part (Section 6.1), a methodology to redistribute neuronal computations on a MPSoC is proposed. Next, in Section 6.2, a study on the coexistence between Software Test Libraries (STLs) and ANNs running on resource-constrained embedded devices is described. The principal achievements of the two studies are discussed in both sections.

6.1 Neurons Redistributions on AI-oriented MPSoCs

Being compute-intensive applications, Artificial Neural Networks are typically executed on programmable high-performance GPUs. Unfortunately, due to the significant power consumption required for running an inference job, they are impractical for applications demanding low-cost and, in particular, low-power devices, despite their remarkable performance. For their flexibility, Application Specific Integrated Circuits (ASIC) devices are gaining increasing interest for this class of applications, especially those intended for the Internet of Things (IoT) and for the edge computing paradigm, e.g., [19, 20].

To cope with the computational complexity of ANNs, ASICs destined to the AI world require the parallelization of many computational units or processing elements (PEs), which can correspond to an entire processor core (e.g., [98]), or a subunit including only the multiplier, the accumulator and an on-chip memory for weights storage. Data parallelism is achieved with the single-instruction multiple-

data (SIMD) computing paradigm, where a single instruction is applied to multiple data items. In other words, each PE elaborates the same instructions simultaneously but on different data.

Clearly, the principal drawback of using ASIC devices for AI-based applications is their limited storage capacity. Being resource-constrained systems, they can hold a limited amount of data, approximately from a few kilobytes [98] up to, in the best case, megabytes [107]. This implies that only bounded-sized ANNs can be deployed on resource-constrained embedded devices. To reduce their memory footprint, compression and quantization methodologies are proposed to move from a full precision representation (i.e., floating-point), to optimized ANN models exploiting reduced bit-width data types (i.e., fixed-point) [45, 87].

This work [89] focuses on AI-oriented multiprocessor SoCs (MPSoCs). They are typically made of two distinct areas: a microcontroller unit that is used for keeping the boot code, managing the peripherals and all the interconnections; and a cluster of PEs that enables the SIMD computing paradigm. This cluster can also include a dedicated hardware for accelerating the convolutional operations, which might account for more than the 90% of operations [108].

The one-to-many paradigm is used to perform an inference process: *one* PE is assigned the computation of *many* neurons. Additionally, to ease the inference phase and to optimize memory accesses, always the same range of neurons is assigned to each PE. It will be referred to as *static scheduling*. An example is provided in Figure 6.1, where neurons $N=\{0,...,n\}$ are distributed in an orderly fashion among all the $P=\{0,...,p\}$ PEs: it is straightforward to identify which neurons a PE handles when launching the inference of a L-layer neural network, where $L=\{0,...,l\}$. Neurons are split in definite chunks made of fixed groups of neurons assigned to a specific PE p during the inference of each layer l. The number of chunks in each layer is always equal or lower to the total number of PEs available in a given hardware device. The number of critical neurons in each chunk is not allocated evenly throughout the P computer resources. This might raise severe problems in terms of trustworthiness and safety. Indeed, a physical fault affecting the hardware may negatively influence the computation of numerous neurons at the behavioural level (software). If this happens for critical neurons, the effects will be greatly emphasized, as analysed in Chapter 3.2, leading to a significant drop in accuracy.

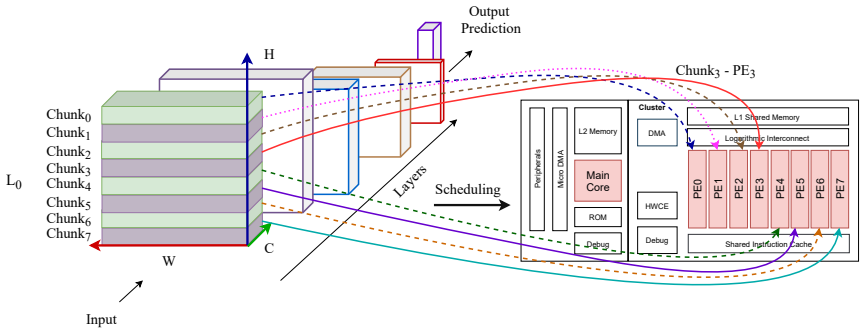


Figure 6.1 Neurons assignment in a AI-oriented SoC exploiting the SIMD configuration.

6.1.1 Proposed Approach

This research work introduces a technique for balancing the chunk assignment to the P processing elements, lowering the risk that a physical fault on a PE may compromise the right functionality of many critical neurons. This research presents a scheduling strategy for allocating portions of neurons to the available PEs. The trend of neurons cannot alter, since no retraining of the ANN is being considered. The only method to avoid this is to redistribute their allocation among the available PEs, so that the computation of the most critical neurons is not given to just one or a subset of them.

The methodology described in 3.2 assigns a resilience score to each neuron in the neural network: the higher the score, the higher the criticality. The variance metric is used to measure the criticality of a group of neurons (i.e., chunks). Hence, measuring the variance for each chunk of neurons means measuring the number of critical neurons included in that chunk. Mathematically, it is computed as the average of the squared differences from the mean μ. The variance of the $chunk_{l,p}$ can be computed as described in (6.1), for each subset of neurons $x_{i,i\in[1,N]}$ assigned to a $\mathrm{PE}_{p,p\in[1,P]}$. N represents the total number of neurons in the chunk, while x_i is the score assigned to the neuron by applying the methodology proposed in Section 3.2.

$$\sigma^2{}_{\mathrm{l,p}} = \frac{\sum_{i=1}^{N}(x_i-\mu)^2}{N} \tag{6.1}$$

The larger the variance, the higher the number of critical neurons that are enclosed in that chunk. In contrast, a small variance value indicates that the chunk holds a few critical neurons.

6.1.1.1 Integer Linear Programming based Methodology

To avoid that hardware faults insist on many critical neurons, the proposed approach aims at redistributing chunks of neurons over the existing PEs. Based on Integer Linear Programming (ILP), the method finds the optimal deterministic scheduling solution to map ANN elaborations on the available hardware resources. It is inspired by the existing scheduling techniques on parallel machines [109–111], where a set of identical machines $M=\{1,...,m\}$ has to handle in parallel a set of jobs $J=\{1,...,j\}$. Jobs can be divided into multiple sections which are processed on several machines at the same time: each job $j \in J$ has weight w_j and processing time pt_j. In conformity with this, the machines $m \in M$ can be considered as the processing elements $p \in P$ of our hardware system and the jobs $j \in J$ as the layers $l \in L$ of our neural network. The terms machines and processing elements as well as jobs and layers are equivalent, i.e., $m=p$ and $j=l$ from this point on. As a result, the problem appears to be extremely comparable while modifying the criterion used to pick the optimum solution. Indeed, based on the criteria that define the problem, an optimum scheduling solution can be found by using integer linear programming. As an example, if the purpose is to minimize the maximum completion time of machines, a scheduling provides a solution for that scope by assigning those jobs $j \in J$ to the machines $m \in M$. Though our problem is very similar, our goal is not to reduce the maximum completion time of machines, but to balance the number of critical neurons that each PE must elaborate. Then, instead of considering the weights w_j or the processing time pt_j, the criterion on which our scheduling is built is the variance $\sigma^2_{\mathrm{j,m}}$ of the job's sections (i.e., the chunks), that measures their criticality (Equation 6.1). In other words, the objective of the proposed method is to uniform the variance of the jobs over the machines.

Using optimization solvers, it is possible to find an optimal and deterministic solution to improve the reliability of the system. Our strategy formulates the problem as an ILP problem that can be described using mathematical equations: an ILP model is set up by specifying the decision variables, the objective function, and the constraints, all of which were defined using the following equations. The best

solution is the one that reduces the difference between the cumulative variance of the machines and the average variance.

Let us make the following definitions, assuming that $1 \leq l \leq L$ and $1 \leq p \leq P$, where L is the total number of layers and P is the total number of available PEs. Then, we need to introduce a third index $1 \leq k \leq K$ that refers to the order of the chunks. Such parameter indicates also how many chunks you can get by distributing the workload of layer l over the available PEs (if P is equal to 8, K will correspond to 8). In a static scheduling, the index k is always equal to p: for instance, the chunk_1 is always assigned to PE_1. With the proposed ILP and variance-based mapping we are going to change this order, and so we need to differentiate among p and k.

As decision variables, integer variables $x_{(l,p)k}$ are used to indicate whether the chunk k of the layer l is assigned to the processing element p or not.

Specifically, $x_{(l,p)k}$ is a binary variable and is equal to:

$$x_{(\text{l,p)k}} = \begin{cases} 1 & \text{if chunk } k \text{ of layer } l \text{ is assigned to} \\ & \text{processing element } p; \\ 0 & \text{otherwise.} \end{cases} \tag{6.2}$$

The variance of the chunk k of the layer l is fixed ($\sigma^2{}_{(\text{l,p)k}}$), regardless of the PE to which it is associated. Hence, we can avoid the index p and only refer to as $\sigma^2{}_{(\text{l,k})}$. The objective function of our ILP problem is:

$$\text{Minimize} \sum_{k=1}^{K} \sum_{l=1}^{L} \sum_{p=1}^{P} \sigma^2{}_{(\text{l,k})} * x_{(\text{l,p)k}} \tag{6.3}$$

This is subject to the following constraints:

- Each chunk k must be assigned to a single processing element p, multiple assignments of sections of the same layer to a certain machine are not allowed:

$$\sum_{p=1}^{P} (x_{(\text{l,p)k}}) = 1, \forall l \in L, \forall k \in K \tag{6.4}$$

- Each processing element p must compute the same amount of chunks k equal to the total amount of layers L:

$$\sum_{l=1}^{L}\sum_{k=1}^{K}(x_{(\mathrm{l,p)k}}) = L, \forall p \in P \tag{6.5}$$

- Each processing element p in each layer l has to process a single chunk k:

$$\sum_{k=1}^{K}(x_{(\mathrm{l,p)k}}) = 1, \forall l \in L, \forall p \in P \tag{6.6}$$

- The cumulative variance elaborated by every PE must be closed to the average one:

$$\sum_{k=1}^{K}\sum_{l=1}^{L}(\sigma^2{}_{(\mathrm{l,k})} * x_{(\mathrm{l,p)k}}) \sim \frac{\sum_{p=1}^{P}\sigma^2{}_{\mathrm{p}}{}^{(\mathrm{TOT})}}{P}, \forall l \in L \tag{6.7}$$

- The cumulative variance of each layer must stay the same:

$$\sum_{k=1}^{K}\sum_{p=1}^{P}(\sigma^2{}_{(\mathrm{l,k})} * x_{(\mathrm{l,p)k}}) = \sigma^2{}_{\mathrm{l}}{}^{(\mathrm{TOT})}, \forall l \in L \tag{6.8}$$

6.1.2 Experimental Results

The targeted MPSoC is an open-source AI-oriented RISC-V device named GAP-8 [98]. As shown in Figure 6.1, it is made of two separate domains: the fabric controller that handles the principal SoC functionalities, and a cluster of eight RISC-V cores used for offloading highly computational-intensive SIMD operations. The RISC-V cores in the cluster are identical and run the same binary code on different data (SIMD configuration). In this work, they will be referred to as PEs. The fabric controller includes a 512 kB of L2 memory and a ROM storing the primary boot code. The cluster hosts a 128 kB of shared L1 memory: every PE can make access to it. The transfers between L2 and L1 are managed by the DMA unit. At the beginning, the DNN parameters (i.e., weights and biases) are stored in the L2 Memory; before running every layer, the DMA moves the current layer's data from L2 memory to the cluster's shared L1 memory.

The targeted DNN running on the above-described MPSoC is a CNN based on the architecture of LeNet-5, trained and tested on MNIST [90], a dataset used to recognize handwritten digits including a training set of 60,000 28x28 gray-scale images, and a test set of 10,000 examples. It is a custom version of the original LeNet-5 and is composed of 7 layers (i.e., 3 convolutional, each one followed by max pooling and the last fully connected) with an input size of 28x28x1. After each convolutional layer, the Rectified Linear Unit (ReLU) activation function is exploited. The CNN model running on the MPSoC has been created in C programming language by exploiting the kernel functions of the open-source PULP-NN library [112]. Since the RISC-V cores of the cluster do not have a hardware floating-point unit, all computations are done in fixed-point arithmetic. Therefore, the neural network's parameters were quantized to 8-bit signed integers. The accuracy of the new quantized network was computed by running the MNIST test set, and it slightly decreased, moving from the 99.31% of accuracy to the 97.64 %.

Considering this system including the mentioned DNN and the MPSoC, a complete CNN inference cycle (a single prediction) takes 276,529 clock cycles (15,772 ms at 18 MHz).

The reliability improvements of the proposed ILP scheduling are demonstrated by comparing two different approaches:

- **Static Scheduling:** The same range of neurons gets always assigned to the same PE, as shown in Figure 6.1.
- **Proposed ILP-based Scheduling:** Chunks of neurons are assigned to PEs depending on their criticality with the aim of distributing them among the different hardware resources.

First, in a static scheduling the number of critical neurons for each PEs is measured with the variance equation, and is reported in Table 6.1. In more details, we rely on the methodology described in Chapter 3.2 to assign resilience scores to individual neurons. Based on that, we determine the criticality of each chunk by computing the variance metric (Equation 6.1). It is worth to underline that to comply with the ILP methodology, numbers have been converted to integers.

Scheduling statically DNN inferences on a MPSoC means that the first chunk of the first layer is assigned to the first PE_0, the second chunk to the second (PE_1),

Chunks Variance - Static Scheduling									
Layer	**Neurons**	**PE_0**	**PE_1**	**PE_2**	**PE_3**	**PE_4**	**PE_5**	**PE_6**	**PE_7**
L0	32,768	12	10	8	6	5	5	6	4
L1	8,192	31	11	12	11	21	10	18	5
L2	4,096	18	15	17	9	13	8	11	9
L3	1,024	19	7	3	2	5	6	2	3
L4	2,048	1	3	3	4	4	4	4	4
L5	512	1	1	2	2	1	1	2	2
L6	10	1	1	1	1	1	1	1	1
Total	**48,650**	**83**	48	46	35	50	35	44	**28**

Table 6.1 Figures of variance when the chunks of neurons are assigned following a static scheduling.

Chunks Variance - Proposed optimal scheduling									
Layer	**Neurons**	**PE_0**	**PE_1**	**PE_2**	**PE_3**	**PE_4**	**PE_5**	**PE_6**	**PE_7**
L0	32,768	8	6	6	4	5	5	12	10
L1	8,192	21	5	10	31	18	12	11	11
L2	4,096	9	9	15	8	13	17	11	18
L3	1,024	3	19	7	2	5	6	3	2
L4	2,048	4	4	3	1	3	4	4	4
L5	512	1	2	1	1	1	2	2	2
L6	10	1	1	1	1	1	1	1	1
Total	**48,650**	**47**	**46**	**43**	**48**	**46**	**47**	**44**	**48**

Table 6.2 Figures of variance when the chunks of neurons are assigned following the proposed ILP and Variance-based optimal scheduling.

and so on. The targeted MPSoC is equipped with 8 PEs: therefore, the workload of each layer is split in 8 chunks of neurons and statically assigned to the 8 PEs of the cluster. For each layer, the total number of neurons is provided in the second column of Table 6.1: each chunk is composed of that number divided by the available PEs. This can not be extended to the last fully connected layer, since there is not a precise division during the inference: having the neurons all connected between them, every PE elaborates all neurons. From the third to the last columns, the variance numbers are provided for each chunk of the layer. Overall, it is evident that the highest quantity of critical neurons is processed by PE_0: the sum of the variances is equal to 83, the highest. On the other side, PE_7 is the one with the least critical load: the sum of the variances is the lowest among the PEs and is equal to 28.

The proposed approach is used to uniform the distribution of critical neurons among the different hardware resources (8 PEs in our case study). An ILP model

was developed by applying the Equations (6.2)-(6.8), and by taking as inputs the numbers shown in Table 6.1. Given our targeted system, the constants were tuned: *P=8* and *L=6*. To find the optimal solution, the ILP engine was set up with all the equations and constraints, in format acceptable for the solver. An open-source optimizer, i.e., OpenSolver [113], was used, and the optimization engine was CBC (COIN-OR Branch-and-Cut). The optimizer returned as outcome the scheduling reported in Table 6.2. As shown, the optimizer arranges the chunks in such a way that the cumulative variance attributed to each PE over the inference cycle is equally distributed. Better solutions are not consistent with the integer constraints, crucial to comply with (6.4) and (6.6). As a result, the kernel of the PULP-NN library was modified to match the ILP solver's optimal scheduling order. As previously stated, the allocation of chunks to different PEs has no impact on the final classification results.

It must be underlined that numbers of variance for each chunk are fixed, regardless the PEs they are assigned to. Moreover, the row *L6* in Table 6.1 was excluded from the ILP formulation: it was referred to the last fully connected layer for which the chunk assignment does not make sense for topological reasons.

The efficacy of the two different scheduling (Table 6.1 and Table 6.2) are compared by means of fault injection (FI) campaigns at the architectural level (RTL). The same set of permanent faults (stuck-at-0 or stuck-at-1) was injected into the RTL design of the PULP platform in two scenarios: first, when the CNN application was compiled by following a static scheduling, then, when it was compiled with the proposed ILP-based optimal one.

To run FIs at the RTL, a specific FI framework was developed based on a commercial simulator from Mentor Graphics. It must be said that FIs at the RTL are computationally intensive and extremely time-consuming. To run a complete inference cycle, it took about 25 minutes. Therefore, massive FI campaigns were out of our computational possibilities. To speed up the simulation, the pipelined fault injector presented in Chapter 4 was used. In this way, it was possible to get an inference result about every 10 minutes. As described before, the pipeline concept applied to neural networks was used to parallelize the inference cycles.

Permanent faults have been injected, one at a time, into the 8 RISC-V cores belonging to the cluster domain of the GAP architecture. As specified, the inference process is completely performed by the cluster's cores (PEs) in a SIMD configuration.

The main core sitting in the fabric controller domain is only in charge of turning on the cluster, so assessing its reliability is out of the scope of this paper. Specifically, stuck-at faults have been injected on the inputs and outputs of the Flip-Flops composing the registers. It is known that when working at the architectural level, the injection locations are limited to some data-path units, micro-architectural units such as registers or memories.

In line with the ranking proposed in [43], faults have been classified depending on their effects. Additionally, we add a component-level metric which is typically more connected to the hardware but, as suggested in [114], can be interestingly applied to classification problems: the Mean Squared Error (MSE) of the output vector.

Therefore, a fault is *detected* in the following cases:

- **SDC-1**: A Silent Data Corruption (SDC) failure is a deviation of the network output from the golden result, leading to a wrong prediction. Hence, the fault causes the image to be wrongly classified.
- **Masked with MSE>0**: The network correctly predicts the result, but the MSE of the faulty output vector is different from zero. It means that the top score is correct, but the fault causes a variation in the outputs compared to the fault-free execution.
- **Hang**: The fault causes the system to hang and the HDL simulation never finishes.

In the remaining cases, the fault is said to be **Masked with MSE = 0**.

In total, 164,000 RTL injections have been performed. The same group of 2,050 stuck-at-faults have been injected in the cluster area of the GAP-8 RTL design; specifically, the PEs' register file. FI campaigns were performed in the following way: a sample of 40 images was randomly selected from the MNIST test set. Then, a stuck-at-fault was injected into one of the PEs and the inference of the selected 40 images was performed by compiling the kernel functions of the CNN application with a static scheduling (a total of 82,000 inferences). Then, by keeping the same stuck-at-fault, the inferences of the same images were executed by compiling the application's kernel with the proposed scheduling (a total of 82,000 inferences). Concerning the simulation time, the 164,000 injections took about 41 days thanks to

the parallelization of the environment and the usage of the pipelined FI framework. The reader should note that for faults producing a simulation hang (i.e., 71,840 and 65,040 images in Table 6.3), a timer was used to avoid useless inferences of the full set of images.

The data in Table 6.3 demonstrate the capability of the proposed ILP and variance-based scheduling in improving the reliability of the system. As shown, it is able to reduce by 24.74 % the neural network incorrect predictions (SDC-1 %). Moreover, as expected, the amount of correct predictions with MSE greater than zero (Masked, MSE >0) increased by 97.80 %. In other words, the new scheduling is able to reduce the risk of wrong predictions, by creating evidence of faults in the output vector (MSE >0) but keeping the prediction correct. A third good point concerns the last row of the table (Masked, MSE $= 0$): the suggested scheduling improves the masking ability of the neural network by 59.53 %.

To conclude, we computed the impact of the proposed ILP-based scheduling at run-time in terms of simulation time and memory footprint. Compared to a traditional scheduling, the adoption of the new ILP scheduling leads to an increase of 3.2% in simulation time for a single inference cycle and 0.6% in memory occupation, due to the additional code included to modify the chunks' assignment. However, despite this small increase, the results demonstrate the improvement in the reliability of the target neural computing system, which starts to be crucial in safety-critical areas.

Table 6.3 RTL Fault Injection Results.

Fault Injection Results	**Static Scheduling**		**Proposed Scheduling**		**[%] Variation**
	Images	**[%]**	**Images**	**[%]**	
SDC-1	1,338	1.63	1,007	1.23	-24.74
Hang	71,840	87.61	65,040	79.32	-9.47
Masked, MSE >0	4,910	5.99	9,712	11.84	+97.80
Masked, MSE = 0	3,912	4.77	6,241	7.61	+59.53
Total	82,000	100	82,000	100	

6.2 Software Test Libraries for ANNs

Artificial Neural Network based applications currently play a key role in everyday life. They are used as decision-making models in several areas such as banking operations,

self-driving cars, robotics. As a consequence, assessing their reliability is becoming of a paramount importance. In the literature, several **fault mitigation** techniques have been proposed to adequately protect (passive fault tolerance) [115, 116], or correct (active fault tolerance) [21, 117] the system running neural networks based applications in presence of faults. On the other side, detecting in field the occurrence of faults has not yet been properly explored for ANN-based applications, expect from a preliminary study [118].

Among the different **fault detection** techniques used in field, the usage of Software Test Libraries (STLs) is a widely adopted solution for testing microcontrollers to perform periodic on-line tests during the system mission [73]. A background in the area is given in Section 1.2.5. In this work, the stuck-at fault model [26] is considered, a widely used fault model for digital integrated circuits and processors.

6.2.1 Proposed Approach

This research work provides a comprehensive analysis of the use and the integration of STLs for the on-line test of embedded systems running ANN-based applications [119]. A recent trend is to exploit low-power, low-cost, and resource constrained hardware devices, especially for the Internet-of-Things (IoT) field and for the edge computing paradigm. As described in Section 6.1, AI-based microcontrollers are typically designed with two main domains: the main domain with a main core used to handle the peripherals and manage the interconnections, and a cluster of processing elements (e.g., MAC units or entire processor cores) used for speeding up the arithmetic operations.

The paper presents a fault detection technique based on the adoption of an STL that must coexists with the requirements and the limitations of such resource-constrained microcontrollers.

The relevance of incorporating in-field testing methodologies stems from the fact that, unlike general applications, ANN-based applications are computationally costly, requiring millions of arithmetic operations in a single inference cycle. Furthermore, because these predictive models can make decisions, proving their accuracy in the field is becoming an important requirement for safety standards. A set of possible scenarios where the on-line execution of test programs is interleaved with the ANN-based application is described. For each of them, the impact introduced by the STL

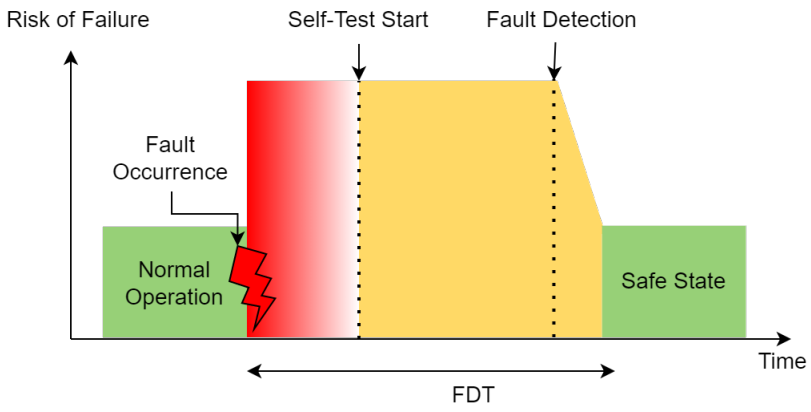


Figure 6.2 Fault Detection Time

execution on the performance of an ANN inference time is evaluated; in addition, the Fault Detection Time (FDT) of the STL is also considered. When monitoring for the incidence of faults during run-time, two primary characteristics must be taken into account, regardless of the self-test safety technique used.

The first is the *availability* of the system, which is related to the system's performance. All system functions (interrupts reaction, application software, and others) are delayed when the in-field self-test mechanism is enabled.

The second one is the *Fault Detection Time*, defined as the time the self-test mechanism takes to detect a detectable fault in the worst-case scenario from the time it occurs. In practice, it may be thought of as the time it takes to indicate whether the system is fault-free, assuming that all problems are equally likely (with a given confidence level expressed by the fault coverage). A graphic illustration is provided in Figure 6.2. As evident, the parameters *availability* and *FDT* are mutually exclusive: the higher the required system availability, the lesser the time window available for the self-test. It's worth mentioning that the parameters are heavily influenced by two factors: the application's real-time limitations, and the application's safety requirements (the higher the criticality of a failure, the more often the system must be tested).

The proposed study applies to AI-based microcontrollers, and, depending on whether the adoption of a software test library may degrade or not the ANN's performances, identifies six possible scenarios. In the following, they are distinguished in two principal categories:

1. **STL execution without ANN performance penalty.**

   Three cases that have no effect on performance are detailed in this paragraph as they are carried out during the ANN's idle intervals. Clearly, because the aim is to optimize system availability without affecting the ANN's performance, executing the full run-time STL at a single idle moment is impossible. Because the STL is made up of a series of separate self-test programs, this is simple to accomplish. The run-time programs are made to be the least intrusive as possible. As a result, their duration is limited in order to fit inside the idle times.

   - A1: Ram Data Transfer.

     The scheduling of the self-test programs is done during the data transfer between the main memory and the innermost level of cluster data RAM memory. A Direct Memory Access (DMA) unit is usually assigned the transfer. The PEs are idle at this period as the responsibility is offloaded to the DMA.

   - A2: Group-based Round robin Self-test during low intensive computation layers.

     It might happen that during the inference process, some PEs are not employed for the computation, and they can be used for the self-test. In contrast to A1, the technique concentrates on just evaluating a few PEs. Obviously, an efficient and complicated scheduler based on the round-robin paradigm is required for this technique. The scheduler must be able to calculate the non-parallelizable layers of the ANN application using several PEs.

   - A3: A1 combined with A2.

     The last class in this category combines A1 and A2 by exploiting the idle times during the RAM data transfer for running test programs on all PEs. Moreover, it executes self-test programs on unused PEs.

2. **STL execution with ANN performance penalty**

   In this category, the execution of self-test programs negatively impact the ANN performances while improving the FDT.

   - B1: PE-level Round Robin Self-test execution.

While one PE is performing the self-test, the others share the task that would have been given to the PE who is being tested. This invariably causes a delay in the execution of inference. However, this approach is inaccurate since the FDT is drastically lowered only for single PEs, and the FDT must refer to the entire cluster.

- B2: Parallel Self-test at the end of the inference.

  The prior technique is ineffective in terms of FDT reduction and has a negative impact on ANN performance. A preferable option would be to do the self-test at the conclusion of each inference. In practice, the system examines the state of the PEs in the cluster before initiating the next inference. It is worth mentioning that the self-test is run in parallel on all the PEs that are active.

- B3: Combining idle times with interlayer execution.

  The last method entails interlayer execution and idle times during RAM data transfer. In reality, the self-test is launched simultaneously on all PEs during their idle intervals. As a result, since the aim is to lower the FDT, the best is to complete the run-time STL in one inference or fewer. As a result, extra test sessions are scheduled at the conclusion of each layer.

6.2.2 Experimental Results

Experiments are carried out on two different neural computing systems: a 7-layer CNN running on GAP-8 [98], and ResNet-18 running on HERO. As for the first AI-based microcontroller (i.e., GAP8), it is made of two separate domains: the main area with a main core and a cluster of 8 processing elements (i.e., cores). All the 9 cores are identical RISC-V cores (named *ri5cy*) and feature a 4 stage in-order single-issue pipeline and support the RV32IMC instruction set plus extensions to boost performance for digital signal processing. The main domain includes 512 kB of L2 memory, while the cluster cores can access a shared 128 kB of L1 memory. When GAP-8 runs CNN-based applications, computational-intensive operations are offloaded to the cluster together with, if desired, the accelerator engine. All the MPSoC cores execute the same binary following the SIMD paradigm, with the aim of maximizing the power efficiency and increasing the instruction-level

Table 6.4 CNN timing details

CNN Operations	Clock Cycle	Cores Active
L2-L1 mem transfer	756	Fabric Controller
Convolutional (layer 1)	116,193	Cluster's Cores 0 - 7
ReLU	2,281	Cluster's Cores 0 - 7
Max Pooling (layer 2)	9,245	Cluster's Cores 0 - 7
L2-L1 mem transfer	1,162	Fabric Controller
Convolutional (layer 3)	100,532	Cluster's Cores 0 - 7
ReLU	1,316	Cluster's Cores 0 - 7
Max Pooling (layer 4)	4,564	Cluster's Cores 0 - 7
L2-L1 mem transfer	1,171	Fabric Controller
Convolutional (layer 5)	24,111	Cluster's Cores 0 - 7
ReLU	1,264	Cluster's Cores 0 - 7
Max Pooling (layer 6)	2,718	Cluster's Cores 0 - 7
L2-L1 mem transfer	801	Fabric Controller
Fully Connected (layer 7)	10,418	Cluster's Cores 0 - 4

parallelism. A complete inference cycle (a single prediction) takes 276,529 clock cycles (15,772 ms, 18 MHz). At the beginning, all the network data are stored in the 512 kB L2 Memory in the main domain. Layer by layer, they are transferred to the 128 kB cluster's memory by the DMA controller. Interestingly, this transfer may be considered a downtime for the network inference, and therefore it could be exploited for implementing safety mechanisms (e.g., on-line self tests). For the sake of completeness, in Table 6.4 we report the precise timing in terms of clock cycles for each inference step. In Column 3, the active cores are highlighted. The CNN under assessment is made of 7 stacked layers, varying from convolutional, max pooling and fully connected. It has been trained and tested on the CIFAR-10 database, a dataset consisting of 60,000 32x32 coloured images grouped in 10 output classes. All the convolutional layers are followed by the ReLU activation function.

As a second case study, a different neural network was considered, i.e., ResNet-18 running on the HERO [120], a heterogeneous SoC that integrates a multi-cluster of RISC-V PEs. It is made of 4 replicas of the RISC-V cluster; each of them is equipped with 8 cores for a total of 32 PEs. A ResNet-18 inference requires about 850 ms (with the HERO microcontroller operating at 18 MHz), including also the DMA cycles used to load the parameters into the memory of each layer. Consistently, the inference process is the similar to the one described for the previous CNN; the main difference is that this DNN is done on a multi-cluster system and the workload is shared between the active clusters.

The STL was developed for the in-field test of the RISC-V cores of the microcontrollers (the *ri5cy* cores). It comprises a total of 29 test programs, 21 intended for the execution at run-time. For each of them, Table 6.5 reports the test program duration (in terms of clock cycles), the stack frame size, and the code memory occupation. On the whole, the entire STL takes 60,155 clock cycles and occupies 17.7 kB of memory. Considering only the run-time tests (the one that will be interleaved with the ANN inference), the duration corresponds to 11,736 clock cycles while the memory occupation is around 5.8 kB.

Table 6.5 Software Test Library Details

Test Program	**Duration [cc]**	**Stack Frame Size [Byte]**	**Test Program Size [Byte]**	**Test Classification**
Mull 1	584	24	258	Run-time
Mull 2	587	24	258	Run-time
Mull 3	581	24	258	Run-time
Mull 4	583	24	258	Run-time
Div 1	585	24	258	Run-time
Div 2	582	24	264	Run-time
Div 3	587	24	264	Run-time
Shifter	454	24	264	Run-time
Logical	458	24	210	Run-time
Adder 1	553	24	220	Run-time
Adder 2	561	24	242	Run-time
Adder 3	586	24	242	Run-time
Mull dotp 1	588	24	242	Run-time
Mull dotp 2	581	24	384	Run-time
Mull dotp 3	588	24	384	Run-time
Mull dotp 4	586	24	384	Run-time
General ALU 1	489	24	384	Run-time
General ALU 1	524	24	180	Run-time
Vectorial ALU 1	558	24	180	Run-time
Vectorial ALU 2	574	24	358	Run-time
Vectorial ALU 3	549	24	358	Run-time
Register block 1	5,291	52	986	Boot-time
Register block 2	5,843	52	1,121	Boot-time
Load Store	2,912	52	1,658	Boot-time
Decode	9,584	76	2,226	Boot-time
Exception	8,796	76	1,156	Boot-time
Hardware loop	6,891	76	1,522	Boot-time
Branch control	4,978	76	1,187	Boot-time
Prefetch	4,122	76	2,284	Boot-time
TOTAL	60,155		17.7 kB	
TOTAL (run-time)	11,736		5.8 kB	

Table 6.6 RI5CY stuck at faults details and test coverage

Unit	#faults before ASSFC	#faults after ASSFC	FC [%] boot-time & run-time	FC [%] run-time
cs_registers_i	29,657	25,350	84.05	26.57
ex_stage_i	69,214	68,151	97.96	95.73
id_stage_i	66,348	63,237	85.42	67.26
if_stage_i	18,618	15,145	86.57	65.23
load_store_unit	5,774	5,365	85.47	44.35
RI5CY_pmp_unit	77,784	425	91.51	88.87
glue logic	1,093	985	77.69	76.51
TOTAL	268,488	178,658	90.08	71.59

The number of stuck-at faults that could affect the RISC-V core is provided in the 2nd column of Table 6.6 and corresponds to 268,488. The faults are grouped according to the functional unit they belong to. These figures refer to a synthesis performed with the 45nm NangateOpenCell Library. Additionally, given the workload, 89,830 safe faults were identified and safely removed from the fault list. Such faults were found according to the Autonomous Systems Safe Faults Classification Process (ASSFC) described in [118]. As it can be seen from Table 6.6 (3rd column), a significant number of safe faults are identified in the memory protection unit (RI5CY_pmp_unit), which is normally used by the operating system. Since the target application does not make use of it, such unit is unused. Given this, the STL Test Coverage considers exclusively the remaining 178,658 faults (considered as unsafe). It is worth to underline that the gate-level descriptions of all the considered processor cores within the system are equal (they underwent the same synthesis process). Hence, the Fault Coverage (FC) reached by the STL is valid for all the RISC-V cores. The FC is provided in Table 6.6. In particular, Column 4th reports the cumulative test coverage achieved by both boot-time and run-time tests, equal to 90.08%. Contrarily, when considering exclusively the run-time tests, the test coverage drops to 71.59% (5th Column).

To study the feasibility of using an STL to on-line test an embedded system oriented to execute AI-based applications, a commercial simulator was used to derive the CNN execution time and FDT. For the experiments, the cluster was operating at 18 MHz and the instruction caches were enabled for the entire duration (including the self-test program execution). In the A1 scenario, it is possible to derive a test window of 3,886 clock cycles for each inference due to the DMA. As the run-time

STL lasts 11,736 clock cycles, the FDT is equal to three full inferences plus some few additional clock cycles of the following inference. In other words, once every four inferences, the run-time self-test diagnostic can provide an answer concerning the integrity of all eight cluster cores. Instead, in the A2 scenario, the cores not interested in the computation of a CNN layer are exploited. Table 6.4 shows that five of the eight cores are involved in the computation of the last fully connected layer. The remaining three can run the STL in a test window of 10,418 clock cycles. Given the duration of the run-time STL, two full inferences are required. It is worth noting that after these two inferences, only three cores were tested. For achieving the target fault coverage on the entire cluster, six full inferences (and consequently also the FDT) are required. Clearly, every two inferences, the last fully connected layer must be scheduled on those cores that have been fully tested during the last test session. In this way, every two inferences, three new cores are available to be tested. The scenario A3 combines the two previous scenarios by exploiting all possible IDLE times. With this policy, it is possible to execute an entire self-test on three cores per inference. Therefore, from the FDT point of view, within three inferences each core has been tested at least once without any performance loss. In scenario B1, a core is excluded from the ANN calculation and executes the STL exclusively. The remaining seven cores run the ANN application, including the work of the excluded core. Specifically, with the cluster clock running at 18 MHz, the inference time increases from 15.772ms to 18.231ms. However, this reasoning is flaw as the FDT (equal to the duration of the run-time STL, 11,736 clock cycles) is greatly reduced exclusively for one core. Indeed, as the FDT must refer to the entire cluster and the core are interleaved at the end of the inference, the FDT is actually eight full inferences. In the B2 scenario, the overall performance of the CNN is reduced but the FDT of the STL is improved. If a detectable fault (i.e., a fault covered by the run-time STL) is occurring at the beginning of an inference, then the system would react after an amount of time equal to one full inference plus the time for executing the run-time STL. When completing the first inference, the next one can start after the cluster completes the self-test only. It is worth underlining that since the STL is scheduled after the CNN, the CNN still completes the computation in time. The last loss-of-performance scenario is a combination of A1 and B2 methods. The aim of this approach is to execute all the test programs in a single inference to reduce the FDT. At the same time, we want to reduce the ANN's loss of performance by exploiting the DMA data transfer. Test programs not executed during DMA are

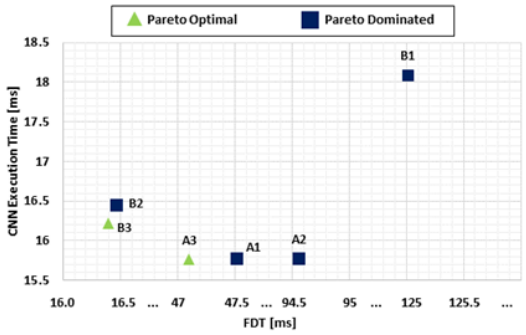


Figure 6.3 CNN performance versus FDT

scheduled in different test windows arranged between the layers' computation. As said before, the test window available during the DMA operation is of 3,886 clock cycles. Since the run-time STL lasts 11,736 clock cycles, the remaining 7,852 clock cycles have to be allocated in seven test windows of 1,122 clock cycles each. As a consequence, the CNN requires 7,852 additional clock cycles to complete.

The obtained results are plotted on a bi-dimensional Cartesian space in Figure 6.3, where the x-axis represents the FDT (expressed in ms), and the y-axis the CNN execution times (still in ms). Each point represents one of the six possible scenarios. Since approaches A1-3 do not alter the CNN execution time, they ideally lie on a straight line parallel to the x-axis. When the CNN performance is paramount, any of these implementations would be in principle appropriate for including also an STL as safety mechanism. However, when considering also the safety of the application, A3 represents the Pareto optimal solution with respect to the FDT. Indeed, it is the closest implementation to the y-axis. This means that when considering the CNN performances, A3 is the optimal solution with respect to the FDT. As a matter of facts, there is not any other point below it with the same CNN performances. When considering the safety of the application as the most important parameter, B3 is the Pareto optimal solution with respect to the CNN execution times. Indeed, B3 worsens the CNN performance of 2.71% while B2 of 4.83%. However, it is worth noting that B3 also has a better FDT with respect to any other point. Since A3 and B3 are Pareto optimal points, they represent the frontier for the considered implementations. Finally, it is interesting to observe that even though B1 is probably the most intuitive and immediate implementation, it is undoubtedly not beneficial for any of the considered parameters. Indeed, it is dominated by any other implementation under any circumstance.

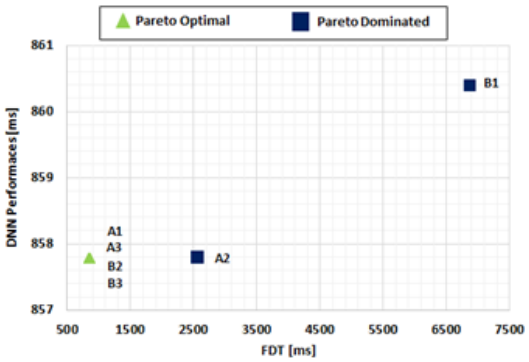


Figure 6.4 Resnet-18 performance versus FDT

Regarding the second case study, all the above-described analyses have been replicated on ResNet-18 running on HERO. Experimental results with the six different scenarios are illustrated in Figure 6.4. In the A1 scenario, the entire STL is performed during DMA cycles in a single CNN inference; similarly, in scenarios A3 and B3, all the STL is performed in a single inference exploiting the DMA activity time. Instead, the A2 scenario needs three inferences to execute all the STL. The A2 scenario exploits three PEs not used during the last fully connected layer of the ResNet-18. The B1 scenario has a significant impact on the performance of the system, as depicted in Figure 6.4. Finally, the B2 scenario, in which the STL is performed after each inference, introduces a minimal effect on the CNN performance. The B2 scenario performances are very similar to that of scenario A1.

To conclude, it is worth underlying that such scenarios do not represent exhaustively all the possible existing cases. They have been selected since they provide useful insights on the potentiality of software test libraries as reliability mechanism for ANN-based applications. Future work aims at exploring different architectures, e.g., GPUs, as well as deepen all the not-yet examined scenarios. To conclude, the experimental results demonstrate that the optimal scenario matches with B3 for both the targeted CNNs.

6.3 Chapter Summary

To summarize, this chapter describes two methodologies proposed to either improve the masking ability of neural networks by redistributing neuronal computations in the available PEs, and detect on-line the occurrence of faults. The first technique

experimentally proves that, without recurring to retraining or redundancy schemes, it is possible to mitigate the effects of hardware faults by redistributing the neural computations. Indeed, conventional mitigation techniques that are based on redundancy do not fit well with the compute-intensive nature of neural networks, introducing huge overheads. Clearly, this comes at the cost of a minor increase in simulation time. To the authors' knowledge, this is the first time that an ILP solver is used to find out an optimal scheduling for artificial neural networks. In the literature, ANNs are used as they are for solving multiple or real-time task scheduling problems [121]. A reliability-aware scheduling on heterogeneous multicore processors is proposed in [122] for general applications. While our approach is tailored to ANN-based applications on embedded systems and is performed at compile-time, in [122] the scheduling is done dynamically and applies to multithreaded workloads.

The second approach presents an in-depth analysis on the coexistence of software test libraries and neural networks. Particularly, two main scenarios are illustrated: the on-line execution of the STL *with* and *without* ANN performance penalty. Experimental results show that, by exploiting the application's idle times, it is possible to reduce the fault detection time without greatly impacting the neural network performance.

In the future, the feasibility of the ILP-based scheduling methodology will be studied to schedule neuronal computations on different architectures, such as GPUs. Moreover, the coexistence of STLs with deep neural networks will be investigated to reduce even more the fault detection time.

Part II

Security of modern devices

Chapter 7

Background and Related Works

Hardware-based vulnerabilities are becoming a serious threat in the Integrated Circuit (IC) industry. Current SoC designs are comprised of many Intellectual Properties (IP) blocks coming from third-party vendors. These can maliciously insert additional hardware, commonly known as Hardware Trojans (HTs), aiming at degrading performance, altering functionality or even leaking secret information. Moreover, they can lower the device reliability by changing physical parameters. Specific measures need to be taken for detecting, avoiding, and mitigating potential threats. Since the security needs are driven by the evolving types of attacks, i.e., new adversary models, types, and the intended use of the device, there is no solution that is able to provide complete protection. A common stance both in academia and industry is that the best approach is a set of flexible, technologically-driven solutions that are to be applied during the whole life-cycle of the device: development, deployment and operation.

Understanding the SoC supply chain is the first necessary step for delineating the possible attack scenarios. In [22], the authors provide an interesting overview of the SoC development flow and all the entities involved. They identify three main phases: the *Intellectual Property (IP) Development*, the *SoC Integration* and the *Foundry*. The first one involves all the IPs providers. An SoC is typically comprised of more than one IP unit. To reduce research and development costs, some of them are built in-house, others bought from third-party IP vendors. Once all the IPs are available, they are joined to build an SoC. This phase is known as *SoC Integration*. Both SoC designers and IP providers rely on EDA tools for facilitating the design

process. At this point, all the side structures are already integrated into the SoC, for example, Design-For-Testability modules, Debug Units, and Built-In Self-Test blocks are typically entrusted to third-party specialized vendors. Once the SoC post-layout phase is done, it is sent to the foundry for the IC fabrication. The fabrication process is usually the most costly stage of the flow, thus is usually granted to external foundries. A malicious actor present in any stage can insert an HT at various levels of abstraction. The key issue lies in understanding which of these entities are trusted and which are not. Once it has been established, a threat model can be drawn. In [22, 7], the authors provide a comprehensive list of adversarial models showing exactly when, where, and how a Trojan can be placed into an IC. The field of hardware security given the practically unlimited number and type of the attacks is quite vast. [123] systematizes the classification of threat models, state-of-the-art defences, and evaluation metrics for important hardware-based attacks.

Hardware Trojans (HTs) are defined as malicious and intended alteration of a circuit that endanger the trustworthiness and the security of the hardware, leading to unexpected behaviours. In the research community, a great effort has been spent on classifying them according to factors such as their insertion phase, the activation mechanisms, the location, and the effects. The authors in [7, 6] provide a broad description of HTs as well as a wide category of benchmarks.

A typical HT (*trigger activated*) is composed of a trigger and a payload circuit. The trigger usually monitors specific signals or series of events under some internal or external conditions. When the trigger condition is met, it informs the payload circuit, which executes the malicious function. The trigger is usually hidden under rare conditions, so the HT is dormant for most of the time and the payload inactive. In that case, the circuit acts as a Trojan-free circuit. If the activation does not depend on the trigger circuit, the Trojan belongs to another category, denoted as *always-on*. Such Trojan gets activated as soon as its host design is powered-on. Both of them are silent and accurately hidden into the design to avoid all the pre- and post-silicon verification, validation and testing mechanisms. However, their effect could seriously compromise the correct functionality of the target device, and therefore it is of a paramount importance to develop accurate strategies to detect them as soon as possible.

Regarding the state-of-the-art Hardware Trojans detection techniques, much of the research focuses on discovering the first class of Trojans: the triggered-type [124–

Table 7.1 RTL Hardware Trojan benchmarks available on Trust-Hub [6, 7].

Design Class	AES	b19	BasicRSA	MC8051	memctrl	PIC	RS-232	wb_conmax	TOTAL
Number	21	3	4	7	1	4	10	2	52

126]. Most of these works try to trigger malicious logic by exploiting formal methods such as theorem proving and equivalence checking. Clearly, both the effort and the time required for implementing formal techniques grow as the complexity of the target device increases. Concerning always-on Hardware Trojans, existing techniques mainly rely on side-channel analyses. A design hosting an always-on Trojan does not change its functionalities, producing apparently the correct outputs. Therefore, it is hard to detect them without observing side-channel parameters. Indeed, an infected design evidences a change of physical characteristics, in the form of power, delay or current. Techniques addressing this class of Trojans, [127–129], usually depend on a Trojan free golden reference model that is not always available. The second main drawback is that side-channel analyses are carried out after fabrication. A pre-silicon verification and validation methodology is suggested in [130], but from a formal perspective. They demonstrate the use of theorem proving methods for providing high-level protection of IP cores, as well as the use of symbolic algebra in equivalence checking.

During the last years, huge effort has been invested not only in *developing detection methodologies*, but also in *designing benchmark* circuits to favour the advancements in research. Indeed, the research community has received a strong drive to adopt **open benchmarks** for validating their detection techniques. In this light, many HT models have been proposed [131, 132]. However, the growing complexity of modern devices as well as more mature and elaborate detection methodologies call for more complex benchmark circuits. As mentioned, some authors in their studies proposed different HT taxonomies based on the insertion phase, location, abstraction level, activation mechanism, effects, etc. However, it is complicated to create a HT model given the whole spectrum of constantly evolving attacks and adversaries that are gaining access to more and more phases of the IC development.

A common trend is to use benchmarks released from the Trust-Hub platform[1] [6, 7]. Considering HTs at RTL, only 8 typologies of benchmarks are currently

[1] https://trust-hub.org/benchmarks/chip-level-trojan

available in Trust-Hub (Table 7.1), and none of them is applied to a pipelined processor similar to the ones used in the real life, as for example the ones in the automotive applications. This is even more concerning, given a higher flexibility for implementing different kinds of malicious functions at RTL. The available HTs are injected on a small 8-bit 8051 microprocessor, and a detection technique has already been proposed in [133]. Hence, even the state-of-the-art HT detection techniques are validated on **obsolete benchmarks** that do not reflect the true complexity of the modern embedded devices. As stated in [132], in order to further support the development of appropriate detection methods, the design and implementation of practical HTs needs to be considered.

Chapter 8

Main Contributions

To address this problem, the main contributions addressed in this chapter are two-fold:

- To present the release of RTL Hardware Trojan Benchmarks targeting a pipelined RISC microprocessor core. The developed HT Benchmarks are publicly available for the research community[1]. Their design follows the guidelines for creating a hard-to-detect Trojan, presented in [134]. This is covered in Section 9.1.
- To propose a novel technique for detecting HTs in a pipelined microprocessor design at the RTL. This methodology combines both static and dynamic properties of the RTL design for building a comprehensive detection methodology at the pre-silicon stage, resorting to robust machine learning algorithms. This is presented in Section 9.2.

[1] https://github.com/ale-dam/HT-uP

Chapter 9

Hardware Trojans

9.1 Proposed Benchmarks

The proposed benchmarks are IP-level Hardware Trojans conceived for a pipelined CPU. Such Trojans are implanted into an individual IP core of the SoC and can affect only the specific IP in which they are embedded [135]. The benchmarks comply with the taxonomy and the classification scheme outlined in [22, 6, 7]. Furthermore, the following attributes are outlined for each benchmark: abstraction level, insertion phase, location, activation mechanism, trigger, payload, effect. For the sake of completeness, the insertion phase of the HTs is the Design phase, while the abstraction level is the Register-Transfer level for all the introduced benchmarks. Concerning the effects, the benchmarks might prove to be disastrous or introduce minor damage. Three different categories have been identified:

1. **Degrade Performance (DP)**: The availability of the system under attack might not be affected, remaining fully operational. However, the HT might damage the performance of an IC and, in a worst-case, cause it to fail.

2. **Denial Of Service (DoS)**: The HT when activated stops all the activities of the system.

3. **Change the Functionality (CF)**: The HT alters the functionalities of the system, causing it to perform malicious, unauthorized operations. The CF might also lead to a DP or DoS.

Table 9.1 Hardware Trojan Benchmarks Description

.

Name	Location	Trigger	Payload	Cat
OR1K-**T1**00	Decode Unit	Sequence of instructions	Periodically forcing signal values	DP
OR1K-**T2**00	Control Unit	Counters monitoring read accesses to SPRs	Entering the supervisor mode	DoS
OR1K-**T3**00	PIC Unit[1]	Counters for mask and status register write access	Disabling external interrupts	CF
OR1K-**T4**00	Control Unit	Three counters for monitoring instructions	Disabling control flag bit	CF
OR1K-**T5**00	Decode Unit	A specific sequence of instructions	Introducing "bubbles" to stall the pipeline	DP
OR1K-**T6**00	Data Cache	Counters monitoring Data Cache Final State Machine (FSM) transitions	Invalidating dcache content	DP
OR1K-**T7**00	Load & Store Unit	Instruction type, order and number	Exception on the data bus	DoS
OR1K-**T8**00	Instr. Cache	Counters monitoring Instr. Cache FSM transitions	Invalidating icache content	DoS

Regarding the trigger part in the introduced HT benchmarks, they can be grouped into two main categories. The first one is represented by a sequence of events that, when triggered, enable the payload. Such events can be related to different signals in the model, for instance an exact sequence of instructions, or a set of consecutive values observed on a given bus. There are different possibilities for implementing it; however, two main parts can be identified: a set of conditions that activate or deactivate a targeted flag, and the second one for registering that flag with some auxiliary combinational or sequential logic. Given the complexity of the condition, this type of trigger may be difficult to activate, and therefore may escape the standard verification approaches. The second category of triggers is used to create and check sub-conditions. Once all of them are satisfied, the payload is activated. They can be implemented by monitoring different CPU resources, for example, by observing certain values on the bus, the order and/or the number of certain instructions, the read/write access to the registers, or tracking the value of control signals between different stages of the pipeline. Sub-conditions may also check the state of counters in charge of monitoring different activities in the CPU. The implemented counters

may be the part of a separate process observing the aforementioned activities or be hidden, for instance, in an already existing state machine. In fact, this type of trigger gives the possibility to create a wide-range of complex conditions. A HT would generally be expected to be as much controllable as possible from the attacker's perspective. However, working with a microcontroller, i.e., a SoC that integrates additional components such as peripherals, memories, etc., renders such access more difficult. Given that all the benchmarks are developed for a processor core, and that there are no mechanisms relying on the user input, i.e., component output, such as switches, keyboards or keywords/phrases in the input data stream to activate a Trojan, all the proposed hardware trojans are considered internally triggered. Moreover, they are activated either depending on the time-based events or on the instructions that are being executed. Table 9.1 reports all the essential details related to the newly developed benchmarks: their name, location, trigger and payload brief description and their category.

9.1.1 Experimental Analysis and Implementation

The proposed RTL Hardware Trojans are implemented in the mor1kx CPU, whose architecture and HTs' respective faulty location being depicted in Figure 9.1. The mor1kx is an open-source core provided by the OpenRISC community[2]; it is a configurable 32/64-bit load and store RISC architecture, written in Verilog Hardware Description Language (HDL). Due to the high design flexibility, it is possible to customize the CPU by choosing the best trade-off between area and performance. The version selected in this work (*Cappuccino*) has a pipeline with 4 stages, supports delay slot and is tightly coupled with the caches. It also integrates a Programmable Interrupt Controller (PIC), a Tick Timer (TT) and Debug units. In this work, HTs are injected in the original HDL design, one at a time, by directly modifying the RTL code. On top of eight primary HT designs, detailed in Table 9.1, we performed modifications concerning the complexity of trigger conditions and coding style to expand our benchmark library and to obtain additional 20 HT designs. In the following, the principal 8 designs are discussed and detailed.

Trojan T100: This Trojan is located in the Decode Unit of the CPU. An if-then-else nested structure controls the opcode value originating from the decode unit. Each

[2]https://github.com/openrisc/mor1kx

time an instruction gets decoded, if the sequence is correct, a counter is incremented; if the sequence is interrupted, the counter is reset. The sequence of instructions is ORI-ADDI-AND-ORI-SUB-XOR-AND-XORI-ADD-OR. Once the counter reaches a predefined value, the payload gets activated. In this case, the pipeline is stalled indefinitely, thus, disrupting the service.

Trojan T200: This implementation is located in the control unit of the processor. Eleven counters in the newly added process monitor read and write access of special purpose registers (CPUCFGR, EPCR0, SR, DCCR, PCCR0, PCMR0, PMR, PICMR, PICSR, TTMR, TTCR). With each access, a corresponding counter is incremented. When all the counters reach predefined values, a trigger is activated. The payload in this case is integrated into the existing code by adding a single OR condition to go from user to supervisor mode. Such behaviour is typical when an exception occurs. The effect is interrupts and timer exceptions being disabled, as well as Data and Instruction MMU. Additionally, a device that is in the supervisor mode enables access to some sensitive registers.

Trojan T300: This HT is located in the programmable interrupt controller. Two counters are inserted to count write accesses to *picmr* (PIC mask) and *picsr* (PIC status) special-purpose supervisor-level registers. Once the trigger part is activated, and there are no pending interrupts, the payload gets activated to perform its role by masking all maskable interrupts, which may result in disastrous consequences in safety-critical systems. Reset needs to be performed to unmask such interrupts and disable the HT.

Trojan T400: Malicious trigger-part of this HT consists of three counters, counting the number of 3 instructions in the control stage (e.g., *rfe* – return from exception, *mfspr* – move from special purpose register, *mtspr* – move to special purpose register).

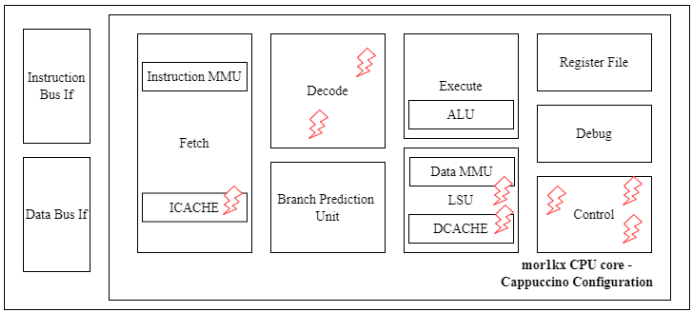


Figure 9.1 RTL Hardware Trojans benchmarks inserted in the mor1kx CPU.

The payload is activated when all three counters count up to a defined value. Once activated, the malicious function is designed to prevent the first succeeding setting of the compare-conditional branch flag by adding a simple condition in the assign statement. However, the effect can be severe, given that often a processor when dealing with some instructions uses exactly this flag to calculate the address or/and choose the operand, which may disrupt the desired flow and cause serious problems depending on the application. Once the request for setting the flag arrives, the HT performs its malicious function and gets deactivated. Additionally, a reset signal resets the counters and deactivates the Trojan.

Trojan T500: In the decode unit, this Trojan is implanted to monitor consecutive instructions. Once the sequence of instructions corresponds to the sequence of a fixed number of predefined instructions, a trigger is activated. The difference with respect to some others HTs introduced in this work is that this HT introduces two processes for registering the activation signal and producing a pulse. In that manner, the payload gets activated periodically. The payload is added to the condition to form the *decode_bubble_o* signal and insert periodically a bubble into the pipeline. The effect is no change in functionality of the processor. However, due to the introduced stalls, it becomes slower and degrades the performance.

Trojan T600: This HT is inserted into the data cache module. The trigger part consists of three counters inserted in the state machine. The Cache FSM has five states: IDLE, WRITE, READ, REFILL, INVALIDATE. The counters have been inserted to count the transitions between the states: IDLE to INVALIDATE, READ to REFILL, WRITE to READ. Once all the three counters reach certain values, cache invalidation is forced.

Trojan T700: This HT is located in the load-store unit. Trigger part consists of nested if-else examining the sequence of consecutive multiple load i.e., store operations with three different types of access: byte (8), half-word (16) and word (32). Once the complex condition gets satisfied, a pulse signal is generated to activate the payload. The payload is integrated into the process dealing with the data bus exceptions. In this regard, once the payload becomes active, it will execute its malicious function by simulating a data bus exception and stepping into the exception routine. As a result, the processor proceeds to the next instruction in the pipeline, skipping the current one at the moment when the exception occurred. Such event may definitely disrupt the normal operation of the processor.

Trojan T800: This HT is implanted into the instruction cache unit. Its trigger part is incorporated within the FSM with counters by following FSM state transitions. Once all the counters get set to predefined values, a payload is activated: the internal hit signal is tied to zero, therefore, every time a request is sent, the instruction cache reports a miss, i.e., not found in cache memory. Consequently, a refill operation is performed, thus significantly slowing down the performance of the CPU.

To demonstrate the feasibility of the proposed modifications, we synthesized all of our 8 HT designs, including the original one, with a 65 nm industrial technology. Successively, we collected reports regarding area, power and frequency. The results are shown in Table 9.2, and demonstrate that the trojan insertions are negligible in terms of area and power overhead. The relative area difference is below 9.8×10^{-4}, while the total power relative difference is below 7.6×10^{-4}. Moreover, we have confirmed that the critical path in the design does not change by introducing the proposed modifications.

Starting from the structure of these original eight HTs, we derived 20 additional benchmarks by mainly modifying the trigger part (complexity of trigger conditions, changing the comparison values, and changing them structurally). For instance, if the trigger looks for a specific sequence of instructions, this has been shortened or extended. Additional wire signals for controlling the conditions are introduced, and the position and number of counters is changed together with comparison values. Furthermore, if the trigger sequence was hosted in a single RTL process, it has been split up to use two or more processes, clearly maintaining the same sequence. For example, Trojan T200, originally uses the value of 11 counters to control the trigger condition. A modified version of this Trojan uses 14 counters for its activation. Their values are incremented within two separate processes (10 + 4). The aforementioned changes are especially useful for evading detection by some methodologies that rely on one particular coding style. On the whole, the benchmark set finally contains a total of 28 Hardware Trojans.

Functional testing is quite unlikely to detect malicious circuitry based on instruction or access sequences, as the input space is too large. The number of instructions in the 32-bit version of the processor is 96 (including custom ones). Therefore, the probability of activating Trojan T100 is 10×10^{-20} an order of magnitude. Moreover, functional verification/testing is statistically useless trying to detect HTs observing multiple counter values. It is not only because of the large number of conditions, but

Table 9.2 Synthesis Results

Design	Size						δArea[%]	Power *mW*			Total	δPower[%]
	Ports	Nets	Cells	Comb.	Seq.	Area		Intern.	Switch.	Leak.		
Orig.	9,679	931,538	924,619	601,116	323,252	4,777,062.18	-	257.62	4.93	69.42	331.99	-
T100	9,679	931,567	924,648	601,145	323,252	4,777,100.66	0.1×10^{-2}	257.62	4.93	69.42	331.99	-1.81×10^{-4}
T200	9,679	932,716	925,797	602,038	323,508	4,781,729.10	9.8×10^{-2}	257.82	4.94	69.48	332.24	7.60×10^{-2}
T300	9,679	931,899	924,980	601,412	323,317	4,778,437.10	2.9×10^{-2}	257.67	4.93	69.44	332.06	2.03×10^{-2}
T400	9,679	932,033	925,114	601,515	323,348	4,779,015.82	4.1×10^{-2}	257.70	4.93	69.45	332.09	3.07×10^{-2}
T500	9,679	931,793	924,874	601,333	323,290	4,777,998.70	2.0×10^{-2}	257.65	4.93	69.43	332.03	1.35×10^{-2}
T600	9,679	932,056	925,137	601,535	323,351	4,779,066.78	4.2×10^{-2}	257.69	4.93	69.43	332.06	2.11×10^{-2}
T700	9,679	931,787	924,867	601,330	323,286	4,777,932.66	1.8×10^{-2}	257.65	4.93	69.43	332.03	1.16×10^{-2}
T800	9,697	932,052	925,043	601,441	323,351	4,779,032.98	4.1×10^{-2}	257.70	4.94	69.45	332.09	3.18×10^{-2}

also given the large comparison values and limited time required to run the simulations. All the listed HTs can get excited and are not completely dormant/silent in terms of activity. Nevertheless, the probability of activating the payload is extremely low without the knowledge of the structure of the malicious circuit inserted by the attacker.

9.2 Pre-silicon Detection methodology

9.2.1 Proposed Methodology

The proposed methodology relies on a supervised learning scheme. It is necessary to underline that, apart from [136], most ML-based techniques are applied at the gate-level. However, more and more examples of Hardware Trojans inserted at the RTL are available, due to the flexibility for implementing various malicious functions. Hence, there is a pressing need for more RTL detection techniques. To fill the above-mentioned gaps, this paper presents a ML-based methodology for detecting triggered-type Hardware Trojans. It combines a *dynamic* approach with a *static* analysis of the RTL model. Indeed, if a static approach analyses the structure of the model by looking for a similarity with the structure of a Trojan, a dynamic method considers the true activity of the circuit. For this reason, the proposed work picks up the best of the two methods to cover a greater set of HTs and, thus, generalize the detection approach.

The proposed flow is shown in Figure 9.2. The input of the framework is the design that is about to be processed; it is the behavioural RTL model description. The output is a report indicating suspicious parts in the design, i.e., the code fragments

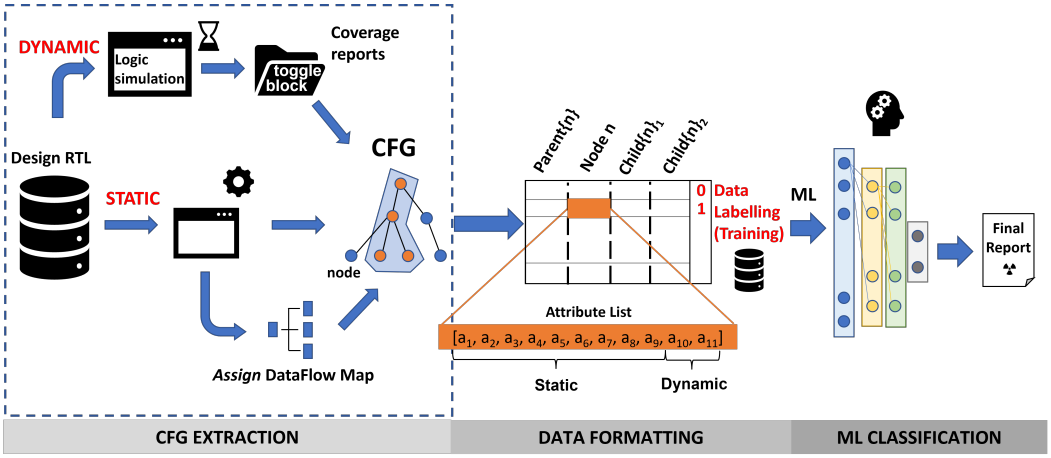


Figure 9.2 Flow of the proposed methodology

that should be checked more thoroughly for malicious HTs insertion. The RTL design is processed in order to extract both dynamic and static information. While the dynamic is derived from observing the model behaviour under different stimuli, the static is obtained without any code execution and is related to the structure and control/data dependency in the code. The data extracted from the RTL model are embedded in CFGs. Static/Dynamic data are used as *attributes* to create input samples out of node sets for the classification task. At the end, a ML-based binary classification is used for distinguishing between input samples originating from the CFGs. The proposed approach is based on the following steps:

1. Control Flow Graph Extraction:
   (a) Static Attributes
   (b) Assign DataFlow Map
   (c) Dynamic Attributes
2. Data Formatting
3. Classification

In the following subsections, each of the steps is detailed.

9.2.1.1 Control Flow Graph Extraction

At the initial stage, the RTL design is represented in the form of a CFG, which incorporates key properties of the design: the static, dynamic, and dataflow map. A CFG is a directed graph $G = (V, E, in, out)$, where V is a set of vertices (nodes) and E set of edges. For each process P in the RTL design D, a CFG G can be extracted. A node $v \in V$ of the graph G can be:

- a single non-blocking statement – allow scheduling assignments without blocking the procedural flow;
- a conditional statement/loop (IF-ELSE, CASE, FOR, WHILE).

E is a finite subset of $V \times V$; e is an edge between the nodes $v1$, $v2$ if and only if $v2$ can be executed after $v1$ in the process P. *in* and *out* are the first and the last node in a CFG, respectively, used to mark entering the process and leaving the process. Then, each node in the CFG holds an attribute list, which will be created as described in the following.

Static Attributes: The static attributes have been extracted from the RTL design by parsing the source code files. Given the complexity of modern designs, this task requires an automated tool that provides as an output an abstract syntax tree (AST). AST is a convenient hierarchical tree-like representation of the abstract syntactic structure of source code. Then, syntax trees generated by the parser are traversed to perform the extraction of the CFGs in accordance with the definition that was previously introduced. It is worth noting that each of the source files may contain more than one process, which are all elaborated sequentially. The algorithm extracts the list of input signals, registers, wires, output signals, and parameters. A CFG node is identified by its unique name and a unique line number that get assigned inside the processes while creating nodes and attaching them to the corresponding graph. Since one node can represent either a conditional statement, i.e., a loop, or a non-blocking statement, it is possible to extract static properties from such constructs. These include the number of input signals, the number of output signals, the number of logic operators, relational and equality operators, arithmetic operators, and numbers (constants). Additionally, each node has its depth in the CFG (level - the number of edges in the path from the root to the node).

Assign DataFlow Map: To deal with the combinational logic (e.g., the `assign` statements in Verilog), the proposed flow introduces an auxiliary structure. Creating an Assign DataFlow Map allows the information outside the (sequential) processes to be captured and incorporated later into the CFGs. The left part of the assign statement is used as a key to identify an item in such structure, while the corresponding value is in a form of a list. Its first element is an array of properties that coincide with the ones for the statements inside the process (*static attributes*). The second one is a list of used signals, either inputs (*input*), registers of integers (*reg*, *integer*) or other wires (*wire*). The map is searched recursively for all of its key elements, summing up the attributes for a corresponding signal list. It stops when there are no more wire signals, i.e., if the remaining ones are a register, integer, input or output. While creating the CFGs and extracting their nodes' static attributes, the influence that a signal (in the Assign DataFlow Map) has on a statement inside the process is taken into account by adding its attributes from the corresponding value in the map entry.

Dynamic Attributes: Logic simulations of the design under assessment are performed to collect code coverage reports, based on standard metrics such as statement and toggle coverage. The idea is to gather information from a set of programs that thoroughly exercise the design under analysis. It is essential to outline that such set of programs may have been written either as a part of pre-silicon or post-silicon verification, validation, or even manufacturing tests etc., targeting different parts and different features of the system. For every instance in the design, the uncovered sequential statements belonging to a process are listed with their line number, source code, and type (*if* and *case* conditional structures, *for* and *while* loops together with non-blocking assign statements). The second type of reports focuses on toggle activity of the signals that are being used outside sequential processes as inputs/outputs, to model combinational logic in assign statements. For each and every program in the library, a statement-coverage report is generated, while only one merged report for all runs regarding the signal toggling. Hence, two additional fields have been created in the attribute list for such purpose: one for execution probability and one for signal toggling activity. Regarding the former, a category is decided for each node (statement) based on the number of executions, i.e., how many times it was covered. This technique is an important tool for preparing numerical data for ML and is referred to as unsupervised discretization [137]. It consists of transforming data from continuous to discrete, using, e.g., equally wide intervals. Typical use case

is having many unique values to model effectively. In Eq. 9.1 that shows the range for deciding a category, n_{exec} is a number of times a statement has been covered out of M runs. N is the number of intermediate categories, set to 5. Consequently, apart from the two extreme categories *never* (N) and *always* (A), there are other five: *almost never* (XS), *rarely* (S), *sometimes* (M), *often* (L), and *almost always* (XL).

$$i\frac{M}{N} \leq cat(n_{exec}) < (i+1)\frac{M}{N}, i \in \{0,1,2,\cdots,N\} \tag{9.1}$$

As for the latter, toggle reports from all the runs are merged into a unique report, showing if a wire signal has toggled in at least one run, fully or partially (rise and fall). The algorithm embeds such information into a node belonging to a process statement in the following manner: wire signals are listed in such statement, if any, otherwise score 0 is set; based on their total number t and the number of those that toggled d a ratio $R = \frac{d}{t}$ is calculated; R falls into one of the ranges, 0, $(0, \frac{1}{4}]$, $(\frac{1}{4}, \frac{2}{4}]$, $(\frac{2}{4}, \frac{3}{4}]$, $(\frac{3}{4}, 1)$, 1, and gets assigned a value from 6 to 1.

9.2.1.2 Data Formatting

By capturing the structural and functional dependency between the nodes in a CFG, the context and neighbourhood information is brought into the predictions. To do so, a node with its closest neighbours is selected to form a set, i.e., to obtain an input sample. Clearly, such sets may vary in size, given the bound that is chosen for grouping the nodes. It is desirable not to be too generic neither too specific, since this action will have an impact on the learning capabilities. For this reason, we have considered a set of 4 nodes. Therefore, each node that has at least one parent and at least one child is processed. For nodes with more than one parent P and more than two children C, all the possible combinations are extracted $P \cdot \binom{C}{2}$. A child having no siblings is included in the selection two times. For all the CFGs, the algorithm implementing a set of above-mentioned rules extracts a set S of node selections $t_i = (p_i, n_i, c_{1i}, c_{2i})$. Subsequently, by expanding its nodes with their incorporated attributes, gets transformed into $t_a i = (a(p_i)[\,], a(n_i)[\,], a(c_{1i})[\,], a(c_{2i})[\,])$. For the training, such input data have to be labelled relying on the set of Trojan Benchmarks introduced in [138]. If a central node n_i for which we select its environment belongs to the malicious insertion then, the set of 4 nodes is marked as positive. Otherwise, it is marked as negative, i.e., non-suspicious.

9.2.1.3 Classification

Once the data have been extracted, the problem may be tackled as a pure Machine Learning classification problem. The learning phase, i.e., the training process, relies on the features obtained from the data formatting. Here, we apply different paradigms to perform the classification and confront their performance in the following subsections. The first one is using SVM algorithm, while the second is based on a fully connected feed-forward neural network.

Support Vector Machine: SVM algorithm is used with different kernels to choose the one that fits best for the problem in question. Often, the differences in the scales across input variables may affect the training process and therefore the final result. A model might become unstable, meaning that it would suffer from poor performance in both learning and validation/test phases as a result of high sensitivity to input data and higher generalization error. Therefore, using pre-processing techniques such as scaling or normalizing input data is preferred when working with many ML algorithms. Normalization is a scaling of the data from the original range so that all values are within the new range between 0 and 1. It can be performed on each individual data sample (row-wise) or across data features (column-wise). Standardization, on the other hand, includes transforming data to change its distribution of values: the mean of the observed values becomes 0 and the standard deviation 1. For this particular purpose, we perform scaling across the features: $\overline{X} = \frac{X - \mu}{\sigma}$.

Feedforward Neural Network: Given the number of attributes, the number of inputs for a fully connected feed-forward neural network is set to 60, after expanding some features with one-hot-encoding. Following the common experience of ML experts, having too many layers when dealing with a limited number of training data (an order of magnitude of 1,000 samples) may result in underfitting. Furthermore, the number of NN inputs is a limiting factor when defining the number of nodes in layers. Given the previous consideration as well as an empirical analysis, the following topology has been adopted: (64, *tanh*), (32, *tanh*), (32, *relu*), (2, *sigmoid*). For the sake of clarity, the first number indicates the number of neurons that constitutes the fully-connected layer, while the second parameter specifies the activation function, e.g., hyperbolic tangent, rectified linear unit, sigmoid. For a fixed topology, tuning training parameters may significantly enhance the NN learning capabilities. Hence,

K-fold cross-validation method is employed to find the best optimizer and select optimal parameters such as batch and number of epochs. One of the challenges faced in ML is memorizing the input samples, especially when having a small training dataset. However, the NNs have shown to be more resilient to those problems. In any case, to reduce the generalization error, i.e., to prevent overfitting, a Gaussian noise is added to the input. In this way, the training process is made more robust.

9.2.2 Experimental Results

The selected platform is AutoSoC [139], an open-source SoC benchmark suite, conceived to serve the needs for standardization and benchmarking in the automotive area.

For every HT benchmark, the following experimental procedure was exploited:

1. Parsing of the design model using a set of Python tools and an in-house developed tool to generate CFGs;
2. Performing the logic simulation and report generation using state-of-the-art commercial tools; then, adding the information originating from the coverage and toggle reports to the CFGs;
3. Node extraction: a selection of nodes with their neighbourhood is made (parents and children) to create textual files whose rows contain the attributes for each of the four nodes. For the training process such data have to be labelled manually; repeating items, if any, are eliminated.

The whole setup has been developed to perform logic simulation and generate reports in Linux environment on a server equipped with a dual Intel Xeon CPU E5-2680 v3 and 256 GB of RAM. The process itself is managed by a set of bash scripts taking care of design elaboration, design simulation, calculating the coverage and merging the reports.

Given the fact that for the training, RTL designs with different type and implementation of HTs have to be simulated multiple times, the time required for obtaining the reports can become significant. To speed up the execution time, a multiprocess environment has been developed. To this end, a library of test programs for *mor1kx* CPU has been simulated on all the 28 RTL *trojan models*. The test

program library includes 46 programs for a total of 64 kB. Launching a set of 46 programs, simulations on one design in this configuration requires 22 minutes on average. By merging the contribution of each single program, the entire test program library achieves 85% of statement and toggle coverage on the golden design model. It is worth underlying that the test program library is not able to activate the HTs, being coherent with the assumption that HTs hide under rare trigger conditions.

In our approach, the tool for performing the task of parsing is Pyverilog [140]. It is a Python-based hardware design processing toolkit for Verilog HDL. The tool relies on Icarus,[3], an open-source tool for performing the preprocessing. It flattens the hierarchy by implementing the `include` and `define` directives, producing the equivalent output related to such directives. Successively, Pyverilog reads the source code and generates Abstract Syntax Tree (AST) in the form of Python nested class objects. The parser is built upon PLY[4] which is used as a parser generator (compiler-compiler). PLY is a Python implementation of the Lex-Yacc lexical analyzer.

The first set of experiments is intended to utilize SVM as a model to perform classification of code sections given in the form of attributes belonging to the family of nodes. Common practice when working with supervised learning and data classification is to split the data set into three exclusive sets: training set, validation set, and test set. However, by partitioning the available data into three sets, we drastically reduce the number of samples used for the learning phase. Consequently, such action might have a negative impact on the model's performance. Furthermore, the results can depend on a particular random choice when choosing/creating training and validation sets. A solution to this issue is using k-fold cross-validation. It consists in splitting the training set into k smaller sets. The following procedure is followed for each of the k "folds": a model is trained using $k-1$ folds as input data for the training; the resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure). Training/validation data and test data contain, respectively, 80% and 20% of the complete data set. The average recall, precision, accuracy and F1-score [141] were calculated on cross-validation sets with 10 folds for each of the four classifiers and reported in the first four columns of Table 9.3. Subsequently, the model was trained on the whole training data set (80%), with a particular model configuration. Next, we examined the models' strength by

[3] http://iverilog.icarus.com/
[4] http://www.dabeaz.com/ply/

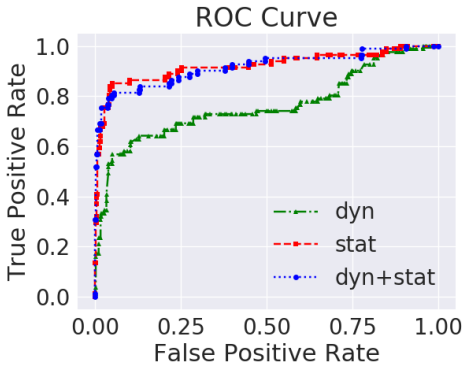
Table 9.3 Experimental results of the four SVM classifiers

.

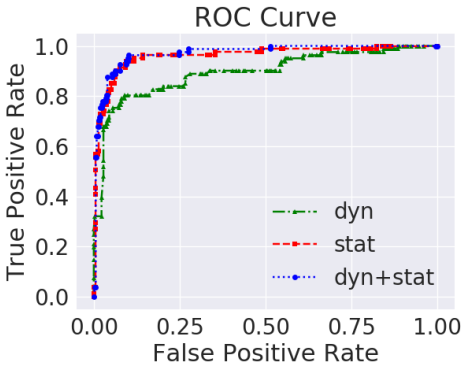
Kernel	Cross-Validation $10-fold$				Training [80%]		Test [20%]					
	Rec.	Prec.	Acc.	F1-sc.	Rec.	Prec.	Rec.	Prec.	Acc.	F1-sc.	TN FN	FP TP
Linear	0.79	0.60	0.87	0.69	0.80	0.64	0.81	0.61	0.87	0.70	314 15	42 66
Polynomial	0.49	0.90	0.90	0.63	0.57	0.97	0.64	0.95	0.93	0.76	353 29	3 52
RBF	**0.82**	**0.81**	**0.93**	**0.82**	**0.88**	**0.90**	**0.91**	**0.87**	**0.96**	**0.87**	**345** **7**	**11** **74**
Sigmoid	0.67	0.42	0.77	0.52	0.68	0.4	0.64	0.41	0.76	0.5	280 28	76 53

applying test data that had not been used previously, i.e., the remaining 20% of the initial complete set.

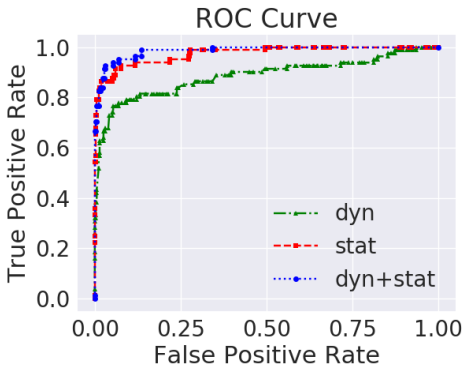
Receiver Operating Characteristic (ROC) curve is a graphical plot showing the influence of the threshold margin on the performance of the binary classifier system; it gives a trade-off between *sensitivity* (true positive rate) and *specificity* (1 - false positive rate). Classifiers with corresponding ROC curves closer to the top-left corner indicate a better performance. On the other hand, the closer the curve comes to the 45-degree diagonal of the ROC space, which is used as a baseline for the random classifier, the less powerful the classifier becomes. Four Receiver Operating Characteristic (ROC) curves for linear, polynomial, rbf and sigmoid kernels are given in Figure 9.3. They provide enough information to analyse the predictive power of a classifier and find the optimal threshold. Based on the aforementioned analysis, the threshold was set to 0.19. Moreover, the RBF kernel was chosen as the best one in terms of performance when compared to the others. This claim can be supported by observing the numerical values in Table 9.3, where we report recall and precision on the training set, and successively, recall, precision, accuracy and F1-score on the test set, together with the corresponding confusion matrix. Additionally, here we decided to split the attributes extracted from the set of nodes and examine their partial influence on the performance of the classifiers. As shown in the Figure 9.3, we performed training using exclusively static attributes (*stat*), then dynamic attributes (*dyn*) and finally, latter and former combined (*stat+dyn*). All the classifiers clearly underperformed when relying only on the dynamic attributes. In case of the classifier



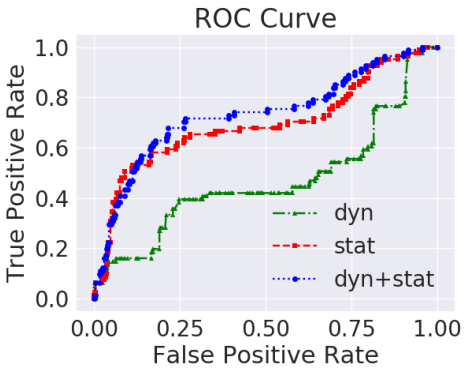
(a) ROC curve for linear kernel



(b) ROC curve for polynomial kernel



(c) ROC curve for rbf (gaussian) kernel



(d) ROC curve for sigmoid kernel

Figure 9.3 ROC curves for 4 different kernels including different set of extracted attributes (farther from the 45-diagonal, i.e., closer to the upper-left corner, the better)

with the RBF kernel, using the complete set of attributes instead of static attributes only resulted in improved classification power; in particular, 0.91 instead of 0.88 for recall, 0.87 instead of 0.8 for precision, 0.96 instead of 0.94 for accuracy and 0.87 instead of 0.84 for F1-score.

The second set of experiments is related to training the NN and evaluating its performance. For selecting its parameters during the training process, exhaustive experiments were run using LazyGrid[5], an open-source package that eases hyperparameters tuning and compares different machine-learning models. To evaluate its effectiveness, eight different experiments have been conducted, one for each group of Hardware Trojan. To determine how the NN will generalize for an independent data set, we used cross validation technique. In other words, the NN has been trained

[5]https://github.com/glubbdubdrib/lazygrid

on a set completely independent of the test one. The results show that even though the NN learns only on a category of Trojans, it is able to discover different types as well.

In Table 9.4, we report, for each training data set, the results obtained by evaluating the learning capabilities of the NN on the corresponding test sets. For a subset of benchmarks T_{k*} that is used later for test, we first train the NN on the whole set of all benchmarks ($\bigcup T$) *excluding* that one particular subset T_k and benchmarks derived from modifying it (T_{k*}). Confusion matrix terminology is used to present training and test performance given the predicted and expected classes for binary classification. The number of CFGs in a design (a set) is equal to the number of processes it contains. Finally, a false positive rate ($\frac{FP}{FP+TN}$) is given in the penultimate column of the table.

Elements of the confusion matrix in context of HT detection are given in Table 9.5, with their corresponding explanation and the effect from the user's point of view. The number of FPs (a non-trojan detected as trojan) should ideally be 0, i.e., in practice it should be kept as low as possible, together with the FNs. However, the obtained numbers (FP and FN) are still significantly low, given the total number of samples that have been evaluated ($\sim$1.5k).

It is essential to outline that, first, the number of FPs remains significantly lower than the number of TNs, while being comparable to TPs. Therefore, checking all samples marked as positive (TPs + FPs), does not represent a huge effort. Secondly, even though there are FNs, it does not mean some parts of malicious code escape the final analysis and remain undetected. As it can be seen from Figure 9.4, a set of nodes marked in orange belongs to the HT (inserted malicious code), while those in blue are not. Those nodes covered in red polygon are detected as malicious, therefore enter in TP category, while those in blue polygon are left undetected, belonging to the FN. By revealing one, others can be examined and by tracing back all TPs, a verification engineer is able to completely discover all the maliciously inserted code. As a conclusion, we can confirm that all the Trojans in the test set have been discovered.

Table 9.4 Experimental results of the Neural Network

Training Dataset $\cup T \setminus$	Training performance				Test Dataset (n_{CFG})	Test performance				FP rate [%]	Det.
	TP	TN	FN	FP		TP	TN	FN	FP		
T_1*	260	1781	42	8	$T_1(183)$	**23**	1493	7	1	1.1	✓
					$T_{11}(183)$	**18**	1493	6	1	1.1	✓
					$T_{12}(183)$	**27**	1493	9	1	1.1	✓
					$T_{13}(184)$	**24**	1493	7	1	1.1	✓
					$T_{14}(185)$	**23**	1493	8	1	1.1	✓
T_2*	308	1764	13	14	$T_2(182)$	**19**	1490	15	5	0.1	✓
					$T_{21}(183)$	**24**	1491	16	6	0.1	✓
					$T_{22}(182)$	**19**	1493	11	5	0.1	✓
T_3*	358	1769	22	11	$T_3(182)$	**8**	1487	1	10	0.3	✓
					$T_{31}(183)$	**6**	1489	2	10	0.3	✓
					$T_{32}(184)$	**5**	1490	7	12	0.3	✓
T_4*	346	1770	25	16	$T_4(182)$	**5**	1485	9	10	0.3	✓
					$T_{41}(182)$	**4**	1487	10	10	0.3	✓
					$T_{42}(182)$	**40**	1494	11	10	0.3	✓
					$T_{43}(183)$	**3**	1487	11	10	0.3	✓
T_5*	305	1770	16	13	$T_5(184)$	**35**	1487	7	8	1.4	✓
					$T_{51}(184)$	**28**	1489	6	8	1.8	✓
					$T_{52}(184)$	**35**	1491	8	13	1.8	✓
					$T_{53}(185)$	**38**	1489	7	8	1.8	✓
T_6*	343	1641	30	9	$T_6(181)$	**7**	1489	9	5	1.2	✓
					$T_{61}(181)$	**9**	1492	5	5	1.1	✓
					$T_{62}(181)$	**9**	1486	15	5	1.3	✓
T_7*	358	1781	32	7	$T_7(183)$	**23**	1494	2	3	1.0	✓
					$T_{71}(183)$	**21**	1496	2	1	1.0	✓
					$T_{72}(184)$	**29**	1496	4	1	1.1	✓
T_8*	340	1656	34	5	$T_8(181)$	**8**	1490	13	2	0.4	✓
					$T_{81}(181)$	**10**	1493	8	2	0.4	✓
					$T_{82}(181)$	**10**	1493	8	7	0.5	✓

Table 9.5 Meaning of the confusion matrix in context of HT detection

Classification	Explanation
True positive (TP)	Trojan code correctly recognized as malicious
True negative (TN)	Circuit code correctly considered safe
False positive (FP)	Safe circuit code believed to be malicious (i.e., a false alarm)
False negative (FN)	Malicious code that escaped detection (i.e., a major error)

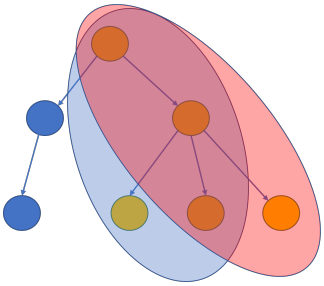


Figure 9.4 Set of nodes belonging to Hardware Trojans as TP and FN

9.3 Chapter Summary

Two principal contributions are presented in this chapter: the introduction of a set of HT benchmarks at RTL, and a machine learning-based detection technique at the pre-silicon stage. Specifically, we proposed a set of HTs for a pipelined processor core. The proposed HTs have been injected in different parts of the processor design. They differ in the trigger and payload. The synthesis reports show the negligible impact that the introduced modifications have on area, power and frequency. We advocate that the set of benchmarks could be extremely useful for validating dynamic HT detection methodologies, since the core is open-source, and they are publicly available. The proposed set of HTs are easily modifiable and allow creating even more complex set of trigger conditions, while the space for inserting payloads is quite vast and allows executing different type of malicious functions. Although most of the detection techniques work at the gate level, shifting their detection to the gate level would result in increased design and verification costs.

Therefore, in the next work we present an efficient method for detecting such Trojans at the RTL, based on both static and dynamic analyses of the circuit. Specifically, we have addressed the problem of detecting RTL Hardware Trojans resorting to ML-based techniques in a pipelined CPU. A mixed approach consisting of static and dynamic model analysis is described, where robust machine learning algorithms

are used to perform classification. Experimental results prove the efficacy of the technique: no HT was left undetected, showing that this technique could be used with similar complex industrial designs, in an automatized manner, reducing both effort and time. The in-house tool was built and integrated into the whole flow: it is adjustable for other commercial tools that can simulate the design and generate a code coverage report. Additional items and rules can be introduced for feature extraction, as well as different CFG node environments to create the classification input. The final flow includes logic simulation, CFG extraction and annotation, and input formatting. The final result of the evaluation is the list of suspicious locations in the code. By *out-of-sample* testing, it is shown that the method using the artificial neural network is capable of identifying all the benchmarks embedded in a complex design, aggravating the detection process. Additionally, we evaluated the performance of four different SVM classifiers. The one using RBF kernel was shown to generalize very well. Comparing two models in terms of performance, SVM RBF kernel is more successful in discovering the set of nodes that is marked as malicious, and also takes less time to train. Nevertheless, both approaches in the end detect each Hardware Trojan as an entity, following the discussion that a set of nodes might represent only one section of a Trojan. The relatively small amount of training data might be responsible for a poorer performance of the neural network. It should be acknowledged that there are certain limitations to this approach. Since it is both static and dynamic, it requires input data for the simulation, either high-level software code, e.g., C code, or directly on the hardware side, e.g., an RTL testbench. Nowadays, in the industrial practice of chip design and development, writing such (verification) programs can start early in the life-cycle, which makes it suitable for this application. Trojans evolve in structure and their location is unpredictable. A lot of effort is being invested into their classification and development. Since the supervised type of learning is used to train both SVM and artificial neural networks, it is uncertain how their classification performance will change with new types of HTs. However, such new malicious insertions may be included into the training set. Moreover, it is important to specify that the proposed approach cannot be directly applied at the post-silicon level because it relies on the availability of the RTL design to extract static and dynamic properties. Typically, at the post-silicon level, the availability of the RTL model description is not guaranteed. Another limitation is the number of false positives. As it has been discussed, false positives at the output of the classifier mean that a part of the benchmark has not been identified as malicious. This does

not mean that the whole Trojan escaped detection: it implies that some additional manual effort is required to decide. The efficiency of the approach nevertheless remains high, given the size and complexity of modern designs.

Future works will be focused on examining and extending the set of properties used for the analysis, and validating the approach further with some other Hardware Trojans.

Chapter 10

Conclusions and Achievements

This PhD thesis addresses the problem of ensuring and improving the reliability and the security of modern embedded systems. It is organized in two main parts. The first one is intended to describe and present all the work that has been done in the artificial neural networks reliability field. As stated, the ever-increasing complexity of modern devices has called for enhanced computing paradigms, and artificial intelligence algorithms currently represent the best answer to cope with this problem. Artificial Neural Networks for their outstanding computational capabilities are today appealing models in different fields and areas: from automotive, avionic, to robotic. Since many of these are considered safety-critical areas, ensuring their reliability has become of paramount importance.

On the other side, the complexity of modern embedded devices has also raised security concerns. Indeed, current SoC designs are made of many IP blocks coming from third-party vendors. This outsourcing may affect the trustworthiness of these devices. External IPs can come with additional malicious circuits aiming at leaking secret information or altering the normal functionality. Therefore, the main effort in these years was put in these two directions.

In the field of artificial neural networks reliability, the principal contributions of this thesis are the following. First, the investigation of the principal vulnerabilities and fault models existing in ANNs and AI-based devices (Chapter 1). Based on that, the proposal of reliability assessment methodologies and the release of specific tools to assist the analysis at different abstraction levels. Chapters 3, 4, and 5 describe the methodologies that have been provided to carry out specific reliability assessments

at different abstraction levels (by considering only the neural network model, or the system including the hardware device and the architectural description). In more details, the following methodologies and frameworks have been proposed: a fault injection framework at the software level to inject errors on ANN weights and biases; a multi-level pipelined FI framework at the architectural level (RTL) able to drastically reduce the fault injection time in simulation; a multi-threading emulator at the software level able to inject faults based on the occurrence of real faults retrieved from test radiation campaigns. All these tools have enabled us to conduct a wide-ranging reliability assessment of AI-based systems. Specifically, they are the bases for allowing us to draw guidelines to make these systems more reliable. As an example, thanks to the several fault injections analysis on the bit width and the data type representation used for storing the ANN parameters, it was possible to identify the best trade-offs between reliability and memory footprint. Next, in Chapter 6, the proposed mitigation strategies developed to improve the safety and the reliability of AI-based devices are given. A reliability-oriented methodology based on an ILP optimizer to schedule ANN elaborations on ASIC MPSoCs is released.

A very big problem today remains the impossibility of generalizing: each neural network must be analysed separately, and, worse, every little changes in the model (such as updating ANN weights) could change the assessment. For this reason, the major impact of my Ph.D. I feel may be regarded as an opening push toward generalization and the definition of guidelines, due to the wide-ranging analysis and proposals that have been done. Furthermore, additionally to the reliability assessment methodologies, mitigation proposals have been made.

In the security field (Part II in this manuscript), two main contributions are described. First, an overview of existing works in the literature as well as background knowledge are given in Chapter 7. Then, in Chapter 9, Hardware Trojan (HT) benchmarks for pipelined processors are proposed. In the state of the art, very few HT benchmarks are available at the RTL, and none of them is applied to pipelined processors: the previously available Hardware Trojans were injected on a small 8-bit 8051 microprocessors, which do not reflect the true complexity of the modern embedded devices. To fill this gap, we released a total of 28 Hardware Trojans Benchmarks targeting a pipelined RISC microprocessor core, and they are available on GitHub. Finally, in Chapter 9.2, a detection technique has been proposed: unlike common approaches, the proposed one combines both static and dynamic

properties for building a comprehensive detection methodology at the pre-silicon stage, resorting to robust machine learning algorithms.

10.1 Future Directions

The aforementioned achievements of this thesis open the way for further investigations in the fields. They allow deepening and extending the research work to address related topics. One of the main effort was put in classifying and identifying the vulnerabilities and weaknesses in artificial neural networks based systems: this study can be used to improve and propose ad-hoc techniques to raise their safety. It is important to highlight that the research conducted in recent years has always viewed the problem with a top-down view, always trying to consider the entire hardware software system. In fact, when it comes to safety, it is difficult to untie the two entities (ANN model and hardware device), which have different vulnerabilities and together can mitigate the effect of faults but also amplify it. The presented research shows that neural networks, although they are mimics of the human brain, cannot be considered inherently robust. Their resilience must be evaluated, preferably in conjunction with the hardware device finally running it.

The ongoing research, which has not been included in this manuscript, follows specific directions. One of them starts from the analysis of critical neurons and related concepts with the idea of exploring the connections between *group* of neurons, and explain the per-class information flow. Indeed, one of the main problem today remains the impossibility of *explaining* the behaviour of such predictive models: to comply with safety standards it is crucial to understand the reasons behind their choices, and to move beyond the *black box* view of neural networks.

In the security field, the ongoing research focuses on the proposal of more sophisticated detection methodologies, aiming at facing the ever-growing complexity of modern embedded devices.

To conclude, it is worth saying that there is still work to be done, and hopefully my work and my results would provide both means and motivation for further research on these directions.

Acronyms

AI Artificial Intelligence.

ALU Arithmetic Logic Unit.

ANN Artificial Neural Network.

ASIC Application Specific Integrated Circuits.

ASSFC Autonomous Systems Safe Faults Classification.

AxC Approximate Computing.

BE Block Error.

CF Change the Functionality.

CFG Control Flow Graph.

CNN Convolutional Neural Network.

CoA Class Oriented Analysis.

CPU Central Processing Unit.

DL Deep Learning.

DMA Direct Memory Access.

DNN Deep Neural Network.

DoS Denial of Service.

DP Degrade Performance.

DRAM Dynamic Random Access Memory.

DUT Design Under Test.

E-SER Execution Soft Error Rate.

ECC Error Correction Code.

FC Fault Coverage.

FDT Fault Detection Time.

FI Fault Injection.

FP Floating Point.

FPGA Field Programmable Gate Array.

FSM Final State Machine.

FxP Fixed Point.

GPU Graphics Processing Unit.

HDL Hardware Description Level.

HERO Open Heterogeneous Research Platform.

HLS High Level Synthesis.

HT Hardware Trojan.

IC Integrated Circuit.

ILP Integer Linear Programming.

IoT Internet of Things.

IoU Intersection Over Union.

IP Intellectual Property.

MAC Multiply and ACcumulate.

ML Machine Learning.

MPSoC Multiprocessor System-on-a-Chip.

MSE Mean Squared Error.

NaN Not a Number.

NN Neural Network.

NoA Network Oriented Analysis.

PE Processing Element.

PIC Programmable Interrupt Controller.

ReLU Rectified Linear Unit.

RISC Reduced Instruction Set Computer.

ROC Receiver Operating Characteristic.

RTL Register Transfer Level.

SBST Software Based Self Test.

SBU Single Bit Upset.

SDC Silent Data Corruption.

SEU Single Event Upset.

SFAD Safe Faults Application Dependent.

SIMD Single Instruction Multiple Data.

SoC System-on-a-Chip.

STL Software Test Library.

SVM Support Vector Machine.

TT Tick Timer.

Bibliography

[1] M. Kooli, F. Kaddachi, G. Di Natale, and A. Bosio. Cache- and register-aware system reliability evaluation based on data lifetime analysis. In *2016 IEEE 34th VLSI Test Symposium (VTS)*, pages 1–6, April 2016.

[2] Annachiara Ruospo, Ernesto Sanchez, Marcello Traiola, Ian O'Connor, and Alberto Bosio. Investigating data representation for efficient and reliable convolutional neural networks. *Microprocessors and Microsystems*, 86:104318, 2021.

[3] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 979–984, 2018.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. doi: 10.1109/5.726791.

[5] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[6] H. Salmani, M. Tehranipoor, and R. Karri. On design vulnerability analysis and trust benchmarks development. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 471–474, 2013.

[7] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security*, 1, 04 2017.

[8] Ito Takuya et al. Constructing neural network models from brain data reveals representational transformations linked to adaptive behavior. *Nature Communications*, 13, 2022.

[9] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, Cham, 2018.

[10] W. Pitts W.S. McCulloch. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

[11] ISO. Road vehicles – Functional safety, 2011.

[12] Terry Sejnowski and Toby Delbruck. *The language of the brain*. Howard Hughes Medical Institute United States, October 2012.

[13] Steve Lawrence, C. Giles, and Ah Tsoi. What size neural network gives optimal generalization? convergence properties of backpropagation. Technical Report UMIACS-TR-96-22 and CS-TR-3617, Institute for Advanced Computer Studies, Univ. of Maryland, 1996.

[14] Simon S. Haykin. *Neural networks and learning machines*. Pearson Education, third edition, 2009.

[15] El Mahdi El Mhamdi and Rachid Guerraoui. When neurons fail. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1028–1037, Orlando, FL, USA, 2017. IEEE.

[16] Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. Convolutional networks for fast, energy-efficient neuromorphic computing. *CoRR*, abs/1603.08270, 2016.

[17] Kung and Hwang. Parallel architectures for artificial neural nets. In *IEEE 1988 International Conference on Neural Networks*, pages 165–172 vol.2, 1988.

[18] U. Ramacher, J. Beichter, N. Bruls, and E. Sicheneder. Architecture and vlsi design of a vlsi neural signal processor. In *1993 IEEE International Symposium on Circuits and Systems*, pages 1975–1978 vol.3, 1993.

[19] Danilo Cappellone, Stefano Di Mascio, Gianluca Furano, Alessandra Menicucci, and Marco Ottavi. On-board satellite telemetry forecasting with rnn on risc-v based multicore processor. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2020.

[20] Gianmarco Cerutti, Renzo Andri, Lukas Cavigelli, Elisabetta Farella, Michele Magno, and Luca Benini. Sound event detection with binary neural networks on tightly power-constrained iot devices. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '20, page 19–24, New York, NY, USA, 2020. Association for Computing Machinery.

[21] Cesar Torres-Huitzil and Bernard Girau. Fault and error tolerance in neural networks: A review. *IEEE Access*, 5:17322–17341, Aug. 2017.

[22] K. Xiao et al. Hardware trojans: Lessons learned after one decade of research. *ACM Trans. Des. Autom. Electron. Syst.*, 22(1), May 2016.

[23] K. Hasegawa, M. Yanagisawa, and N. Togawa. Trojan-feature extraction at gate-level netlists and its application to hardware-trojan detection using random forest classifier. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–4, 2017.

[24] P. Zhao and Q. Liu. Density-based clustering method for hardware trojan detection based on gate-level structural features. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–4, 2019.

[25] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec. 1943.

[26] M. Bushnell and Vishwani Agrawal. *"Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits,"*. Frontiers in Electronic Testing. Springer, Boston, MA, USA, 2013.

[27] Carlo H. Sequin and R. D. Clay. Fault tolerance in artificial neural networks. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 703–708 vol.1, San Diego, CA, USA, 1990. IEEE.

[28] Pravin Chandra and Yogesh Singh. Fault tolerance of feedforward artificial neural networks- a framework of study. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 1, pages 489–494, Portland, OR, USA, 2003. IEEE.

[29] G. Bolt. Fault models for artificial neural networks. In *Proceedings of the 1991 IEEE International Joint Conference on Neural Networks*, volume 2, pages 1373–1378, Singapore, Singapore, 1991. IEEE.

[30] Annachiara Ruospo, Angelo Balaara, Alberto Bosio, and Ernesto Sanchez. A pipelined multi-level fault injector for deep neural networks. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, Frascati, Italy, 2020. IEEE.

[31] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. ACM, USENIX Association.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance

deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[33] CEA-LIST. N2D2. [Online]. Available: https://github.com/CEA-LIST/N2D2.

[34] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[35] Zitao Chen, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. Tensorfi: A flexible fault injection framework for tensorflow applications. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 426–435, Coimbra, Portugal, Oct. 2020. IEEE.

[36] Michael Beyer, Andrey Morozov, Kai Ding, Sheng Ding, and Klaus Janschek. Quantification of the impact of random hardware faults on safety-critical AI applications: CNN-based traffic sign recognition case study. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 118–119, Oct., 2019. IEEE.

[37] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. Binfi: An efficient fault injector for safety-critical machine learning systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1 – 23, New York, NY, USA, 2019. ACM.

[38] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 27–38, Luxembourg City, Luxembourg, 2018. IEEE.

[39] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. PyTorchFI: A runtime perturbation tool for DNNs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 25–31, Valencia, Spain, 2020. IEEE.

[40] Behzad Salami, Osman S. Unsal, and Adrian Cristal 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)*, pages 322–329, Lyon, France, 2018. IEEE.

[41] Yi He, Prasanna Balaprakash, and Yanjing Li. Fidelity: Efficient resilience analysis framework for deep learning accelerators. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 270–281, Athens, Greece, 2020. IEEE.

[42] nvidia. Nvdla, 2021.

[43] Guanpeng Li et al. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov. 2017. doi: 10.1145/3126908.3126964.

[44] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, page 367–379, Seoul, South Korea, 2016. IEEE.

[45] Brandon Reagen et al. Ares: A framework for quantifying the resilience of deep neural networks. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, San Francisco, California, USA, 2018. Association for Computing Machinery.

[46] François Chollet et al. Keras, 2015.

[47] James Bergstra, O. Breuleux, Frederic Bastien, Pascal Lamblin, Razvan Pascanu, G. Desjardins, David Warde-Farley, and Y. Bengio. Theano: a CPU and GPU math expression compiler. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 18–24, Austin, Texas, USA, 2010. SciPy.

[48] Corrado De Sio, Sarah Azimi, and Luca Sterpone. An emulation platform for evaluating the reliability of deep neural networks. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–4, Frascati, Italy, 2020. IEEE.

[49] Chen Chen, Jun Xia, Wenmin Yang, Kang Li, and Zhilei Chai. A pynq-compliant online platform for zynq-based dnn developers. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 185, Seaside, CA, USA, 2019. ACM.

[50] Lucas Matana Luza, Daniel Söderström, Georgios Tsiligiannis, Helmut Puchner, Carlo Cazzaniga, Ernesto Sanchez, Alberto Bosio, and Luigi Dilillo. Investigating the impact of radiation-induced soft errors on the reliability of approximate computing systems. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2020.

[51] Md Mehedi Hasan, Md Raquibuzzaman, Indranil Chatterjee, and Biswajit Ray. Radiation tolerance of 3-d nand flash based neuromorphic computing system. In *2020 IEEE International Reliability Physics Symposium (IRPS)*, pages 1–4, Dallas, TX, USA, 2020. IEEE.

[52] Fernando Fernandes dos Santos, Lucas Draghetti, Lucas Weigel, Luigi Carro, Philippe Navaux, and Paolo Rech. Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. pages 169–176, 2017.

[53] F. Libano, P. Rech, B. Neuman, J. Leavitt, M. Wirthlin, and J. Brunhaver. How reduced data precision and degree of parallelism impact the reliability of convolutional neural networks on fpgas. *IEEE Transactions on Nuclear Science*, 68(5):865–872, 2021.

[54] Lucas Matanaluza, Annachiara Ruospo, Daniel Soderstrom, Carlo Cazzaniga, Maria Kastriotou, Ernesto Sanchez, Alberto Bosio, and Luigi Dilillo. Emulating the effects of radiation-induced soft-errors for the reliability assessment of neural networks. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2021.

[55] A. Floridia, E. Sanchez, and M. Sonza Reorda. Fault grading techniques of software test libraries for safety-critical applications. *IEEE Access*, 7:63578–63587, 2019.

[56] Mohamed A. Neggaz, Ihsen Alouani, Smail Niar, and Fadi Kurdahi. Are CNNs reliable enough for critical applications? an exploratory study. *IEEE Design & Test*, 37(2):76–83, April 2020.

[57] Mohamed A. Neggaz, Ihsen Alouani, Pablo R. Lorenzo, and Smail Niar. A reliability study on CNNs for critical embedded systems. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, October 2018.

[58] Younis Ibrahim, Haibin Wang, Junyang Liu, Jinghe Wei, Li Chen, Paolo Rech, Khalid Adam, and Gang Guo. Soft errors in DNN accelerators: A comprehensive review. *Microelectronics Reliability*, 115:113969, December 2020.

[59] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang. Resiliency of automotive object detection networks on gpu architectures. In *2019 IEEE International Test Conference (ITC)*, pages 1–9, 2019.

[60] Daniele Palossi, Antonio Loquercio, Francesco Conti, Eric Flamand, Davide Scaramuzza, and Luca Benini. A 64mw dnn-based visual navigation engine for autonomous nano-drones. *IEEE Internet of Things Journal*, page 1–1, 2019.

[61] Majid Sabbagh, Cheng Gongye, Yunsi Fei, and Yanzhi Wang. Evaluating fault resiliency of compressed deep neural networks. In *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–7, Las Vegas, NV, USA, 2019. IEEE.

[62] Boyang Du, Sarah Azimi, Corrado de Sio, Ludovica Bozzoli, and Luca Sterpone. On the reliability of convolutional neural network implementation on SRAM-based FPGA. In *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, Noordwijk, Netherlands, 2019. IEEE.

[63] Yann Le Cun, John S. Denker, and Sara A. Solla. *Optimal Brain Damage*, page 598–605. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[64] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

[65] Jianjun Wang, Leshan Liu, and Ximeng Pan. Pruning algorithm of convolutional neural network based on optimal threshold. In *Proceedings of the 2020 5th International Conference on Mathematics and Artificial Intelligence*, ICMAI 2020, page 50–54, 2020.

[66] K. Lee, H. Kim, H. Lee, and D. Shin. Flexible group-level pruning of deep neural networks for on-device machine learning. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 79–84, 2020.

[67] Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Axnn: Energy-efficient neuromorphic systems using approximate computing. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 27–32, 2014.

[68] Shubo Liu, Xu Wang, Jing Wang, Xin Fu, Xingyao Zhang, Lan Gao, Weigong Zhang, and Tao Li. Enabling energy-efficient and reliable neural network via neuron-level voltage scaling. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 410–413, 2019.

[69] Grégoire Montavon, Sebastian Lapuschkin, Alexander Binder, Wojciech Samek, and Klaus-Robert Müller. Explaining nonlinear classification decisions with deep taylor decomposition. *Pattern Recognition*, 65:211–222, 2017.

[70] Fuxun Yu, Zhuwei Qin, and Xiang Chen. Distilling critical paths in convolutional neural networks. *CoRR*, abs/1811.02643, 2018.

[71] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 979–984, 2018.

[72] Muhammad Hanif and Muhammad Shafique. Salvagednn: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 378:20190164, 02 2020.

[73] Paolo Bernardi, Riccardo Cantoro, Sergio De Luca, Ernesto Sánchez, and Alessandro Sansonetti. Development flow for on-line core self-test of automotive microcontrollers. *IEEE Transactions on Computers*, 65(3):744–754, 2016.

[74] Thatte and Abraham. Test generation for microprocessors. *IEEE Transactions on Computers*, C-29(6):429–441, 1980.

[75] C. Gursoy, M. Jenihhin, A. S. Oyeniran, D. Piumatti, J. Raik, M. Sonza Reorda, and R. Ubar. New categories of safe faults in a processor-based embedded system. In *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 1–4, 2019.

[76] Xiaoqing Wen and Hsin-Po Wang. A flexible logic bist scheme and its application to soc designs. In *Proceedings 10th Asian Test Symposium*, pages 463–, 2001.

[77] D. Piumatti, Ernesto Sanchez, P. Bernardi, R. Martorana, and M.A. Pernice. An efficient strategy for the development of software test libraries for an automotive microcontroller family. *Microelectronics Reliability*, 115:113962, 12 2020.

[78] Mihalis Psarakis, Dimitris Gizopoulos, Ernesto Sanchez, and Matteo Sonza Reorda. Microprocessor software-based self-testing. *IEEE Design Test of Computers*, 27(3):4–19, 2010.

[79] P. Bernardi, R. Cantoro, A. Floridia, D. Piumatti, C. Pogonea, A. Ruospo, E. Sanchez, S. De Luca, and A. Sansonetti. Non-intrusive self-test library for automotive critical applications: Constraints and solutions. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 920–923, 2019.

[80] M. Peña-Fernandez, A. Lindoso, L. Entrena, and M. Garcia-Valderas. The use of microprocessor trace infrastructures for radiation-induced fault diagnosis. *IEEE Transactions on Nuclear Science*, 67(1):126–134, 2020.

[81] Stefan E. Damkjar, Ian R. Mann, and Duncan G. Elliott. Proton beam testing of seu sensitivity of m430fr5989srgcrep, efm32gg11b820f2048, at32uc3c0512c, and m2s010 microcontrollers in low-earth orbit. In *2020 IEEE Radiation Effects Data Workshop (in conjunction with 2020 NSREC)*, pages 1–5, 2020.

[82] Justin Meza, Qiang Wu, Sanjeev Kumar, and Onur Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 415–426, 2015.

[83] Alberto Bosio, Paolo Bernardi, Annachiara Ruospo, and Ernesto Sanchez. A reliability analysis of a deep neural network. In *2019 IEEE Latin American Test Symposium (LATS)*, pages 1–6, 2019.

[84] Riccardo Cantoro, Andrea Firrincieli, Davide Piumatti, Marco Restifo, Ernesto Sánchez, and Matteo Sonza Reorda. About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications. *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6, 2018.

[85] 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*, pages 502–506, April 2009.

[86] [Online]. Libfixmath library. https://github.com/Petteri-Aimonen/libfixmath, 2020.

[87] A. Ruospo, A. Bosio, A. Ianne, and E. Sanchez. Evaluating convolutional neural networks reliability depending on their data representation. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 672–679, 2020.

[88] Larry R. Squire. Memory systems of the brain: A brief history and current perspective. *Neurobiology of Learning and Memory*, 82(3):171–177, 2004. Multiple Memory Systems.

[89] Annachiara Ruospo and Ernesto Sanchez. On the reliability assessment of artificial neural networks running on ai-oriented mpsocs. *Applied Sciences*, 11(14), 2021.

[90] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[91] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*, 2011.

[92] Alex Krizhevsky. Learning multiple layers of features from tiny images. *University of Toronto*, 05 2012.

[93] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[94] Pierre Sermanet, Soumith Chintala, and Yann LeCun. Convolutional neural networks applied to house numbers digit classification. In *Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012)*, pages 3288–3291, 2012.

[95] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net, 2015.

[96] Vincenzo Piuri. Analysis of fault tolerance in artificial neural networks. *Journal of Parallel and Distributed Computing*, 61(1):18 – 48, Jan. 2001.

[97] Annachiara Ruospo, Angelo Balaara, Alberto Bosio, and Ernesto Sanchez. A pipelined multi-level fault injector for deep neural networks. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2020.

[98] E. Flamand et al. Gap-8: A risc-v soc for ai at the edge of the iot. In *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 1–4, 2018.

[99] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Davide Rossi, and Luca Benini. Xpulpnn: Accelerating quantized neural networks on risc-v processors through isa extensions. In *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*, DATE '20, page 186–191, San Jose, CA, USA, 2020. EDA Consortium.

[100] Fernando Fernandes dos Santos, Pedro Foletto Pimenta, Caio Lunardi, Lucas Draghetti, Luigi Carro, David Kaeli, and Paolo Rech. Analyzing and increasing the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, 68(2):663–677, 2019.

[101] Edward Petersen. *Single Event Effects in Aerospace*. John Wiley & Sons, Hoboken, NJ, USA, 2011. doi: 10.1002/9781118084328.

[102] PG036 - Soft Error Mitigation Controller v4.1 Product Guide, 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/sem/v4_1/pg036_sem.pdf.

[103] R.C. Baumann. Soft errors in advanced semiconductor devices-part i: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, 1(1):17–22, 2001.

[104] J.L. Leray. Effects of atmospheric neutrons on devices, at sea level and in avionics embedded systems. *Microelectronics Reliability*, 47(9):1827–1835, 2007. 18th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis.

[105] MICHAEL F. L'ANNUNZIATA. 1 - nuclear radiation, its interaction with matter and radioisotope decay. In Michael F. L'Annunziata, editor, *Handbook of Radioactivity Analysis (Second Edition)*, pages 1–121. Academic Press, San Diego, second edition edition, 2003.

[106] Lucas Matana Luza et al. Effects of thermal neutron irradiation on a Self-Refresh DRAM. In *IEEE 15th International Conference on Design & Technology of Integrated Systems in Nanoscale Era*, pages 1–6, Apr. 2020. doi: 10.1109/DTIS48698.2020.9080918.

[107] Giuseppe Desoli, Nitin Chawla, Thomas Boesch, Surinder-pal Singh, Elio Guidetti, Fabio De Ambroggi, Tommaso Majo, Paolo Zambotti, Manuj Ayodhyawasi, Harvinder Singh, and Nalin Aggarwal. 14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 238–239, 2017.

[108] Giuseppe Desoli, Nitin Chawla, Thomas Boesch, Surinder-pal Singh, Elio Guidetti, Fabio De Ambroggi, Tommaso Majo, Paolo Zambotti, Manuj Ayodhyawasi, Harvinder Singh, and Nalin Aggarwal. 14.1 a 2.9tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 238–239, 2017.

[109] Jun-Ho Lee and Hoon Jang. Uniform parallel machine scheduling with dedicated machines, job splitting and setup resources. *Sustainability*, 11(24):7137, Dec 2019.

[110] D. B. Shmoys, J. Wein, and D. P. Williamson. Scheduling parallel machines on-line. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 131–140, 1991.

[111] Thomas Bosman, Dario Frascaria, Neil Olver, Renx00E9; Sitters, and Leen Stougie. Fixed-order scheduling on parallel machines. In Viswanath Nagarajan and Andrea Lodi, editors, *Integer Programming and Combinatorial Optimization*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 88–100, Germany, 2019. Springer Verlag. 20th International Conference on Integer Programming and Combinatorial Optimization, IPCO 2019 ; Conference date: 22-05-2019 Through 24-05-2019.

[112] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini. Pulp-nn: A computing library for quantized neural network inference at the edge on risc-v based parallel ultra low power clusters. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 33–36, 2019.

[113] Www Org, Andrew Mason, and Iain Dunning. Opensolver: Open source optimisation for excel. *Proceedings of the Annual Conference of the Operations Research Society of New Zealand*, 01 2010.

[114] P. Chandra and Y. Singh. Fault tolerance of feedforward artificial neural networks- a framework of study. In *Proceedings of the International Joint Conference on Neural Networks, 2003.*, volume 1, pages 489–494 vol.1, 2003.

[115] M.D. Emmerson and R.I. Damper. Determining and improving the fault tolerance of multilayer perceptrons in a pattern-recognition application. *IEEE Transactions on Neural Networks*, 4(5):788–793, 1993.

[116] C.H. Sequin and R.D. Clay. Fault tolerance in artificial neural networks. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 703–708 vol.1, 1990.

[117] C. Khunasaraphan, K. Vanapipat, and C. Lursinsap. Weight shifting techniques for self-recovery neural networks. *IEEE Transactions on Neural Networks*, 5(4):651–658, 1994.

[118] Annachiara Ruospo, Riccardo Cantoro, Ernesto Sanchez, Pasquale Davide Schiavone, Angelo Garofalo, and Luca Benini. On-line testing for autonomous systems driven by risc-v processor design verification. In *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2019.

[119] A. Ruospo, D. Piumatti, A. Floridia, and E. Sanchez. A suitability analysis of software based testing strategies for the on-line testing of artificial neural networks applications in embedded devices. In *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–6, 2021.

[120] Andreas Kurth, Alessandro Capotondi, Pirmin Vogel, Luca Benini, and Andrea Marongiu. Hero: An open-source research platform for hw/sw exploration of heterogeneous manycore systems. In *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE '18, New York, NY, USA, 2018. Association for Computing Machinery.

[121] Zhen-Ping Lo and Behnam. Bavarian. Multiple job scheduling with artificial neural networks. *Computers Electrical Engineering*, 19(2):87 – 101, 1993.

[122] A. Naithani, S. Eyerman, and L. Eeckhout. Optimizing soft error reliability through scheduling on heterogeneous multicore processors. *IEEE Transactions on Computers*, 67(6):830–846, 2018.

[123] Masoud Rostami, Farinaz Koushanfar, and Ramesh Karri. A primer on hardware security: Models, methods, and metrics. *Proceedings of the IEEE*, 102(8):1283–1295, 2014.

[124] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra. Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution. In *2018 IEEE International Test Conference (ITC)*, pages 1–10, Oct 2018.

[125] S. Yao et al. Fastrust: Feature analysis for third-party ip trust verification. In *2015 IEEE International Test Conference (ITC)*, pages 1–10, Oct 2015.

[126] Y. Wang, T. Han, X. Han, and P. Liu. Ensemble-learning-based hardware trojans detection method by detecting the trigger nets. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, May 2019.

[127] S. Narasimhan et al. Hardware trojan detection by multiple-parameter side-channel analysis. *IEEE Transactions on Computers*, 62(11):2183–2195, Nov 2013.

[128] Y. Liu, K. Huang, and Y. Makris. Hardware trojan detection through golden chip-free statistical side-channel fingerprinting. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2014.

[129] D. Ismari, J. Plusquellic, C. Lamech, S. Bhunia, and F. Saqib. On detecting delay anomalies introduced by hardware trojans. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, Nov 2016.

[130] Xiaolong Guo, R. G. Dutta, Yier Jin, F. Farahmandi, and P. Mishra. Pre-silicon security verification and validation: A formal perspective. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2015.

[131] Samuel King et al. Designing and implementing malicious hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET'08, 01 2008.

[132] Y. Jin, N. Kupp, and Y. Makris. Experiences in hardware trojan design and implementation. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 50–57, 2009.

[133] M. Hicks et al. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *2010 IEEE Symposium on Security and Privacy*, pages 159–172, 2010.

[134] Jie Zhang and Qiang Xu. On hardware trojan design and implementation at register-transfer level. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2013.

[135] Rana Elnaggar and Krishnendu Chakrabarty. Machine learning for hardware security: Opportunities and risks. *Journal of Electronic Testing*, 2018.

[136] F. Demrozi, R. Zucchelli, and G. Pravadelli. Exploiting sub-graph isomorphism and probabilistic neural networks for the detection of hardware trojans at rtl. In *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 67–73, 2017.

[137] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In Armand Prieditis and Stuart Russell, editors, *Machine Learning Proceedings 1995*, pages 194–202. Morgan Kaufmann, San Francisco (CA), 1995.

[138] A. Damljanovic, A. Ruospo, E. Sanchez, and G. Squillero. A Benchmark Suite of RT-level Hardware Trojans for Pipelined Microprocessor Cores. *24th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS) (to be published)*, 2021.

[139] Felipe Silva et al. Special session: Autosoc - a suite of open-source automotive soc benchmarks. In *2020 IEEE 38th VLSI Test Symposium (VTS)*, pages 1–9, 04 2020.

[140] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, 2015.

[141] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, aug 2019.