POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master's degree in Electronic Engineering

Master's thesis

Fault-Injection and Neural Trojan Attacks on Spiking Neural Networks

Supervisors:
Prof. Maurizio Martina
Prof. Muhammad Shafique
Project Ass. Alberto Marchisio

Candidate:
VALERIO VENCESLAI

March 20, 2020

Acknowledgements

I want to express my gratitude to everyone who helped me to succeed in these years. First of all, I would like to thank Prof. Maurizio Martina who gave me the opportunity to work on my thesis and who have been demonstrated to be patient, caring and kind. Many thanks must be devoted to Prof. Muhammad Shafique as well, who supervised my thesis work during my abroad period in Vienna. His dedication, willingness and kindness should be an example for all the people who want to succeed in life. A special remark must be expressed towards my supervisor and friend Alberto Marchisio for his endless care and support, which guided me during my thesis. His altruism and patience have been a comfort and a model to face the difficulties encountered. I would like to thank all the PhD students group in Vienna who always demonstrated kindness, goodwill and expansiveness. I want to thank my parents and grandparents which made my education possible and always helped me in the toughest times with endless devotion. Special thanks must be devoted to the people that have been with me during the thesis in Vienna and the friends I found there, who made me feel like I was home. My gratitude and affection go to the friends I have met in Turin. Words cannot describe how grateful and lucky I feel for having found so many beautiful people. My house mates, my colleagues and my friends always showed me incredible kindness and love. I must thank all my long date friends from my home town who always make me feel their support and presence as if I have never left my city. Special thanks must be reserved to my girlfriend Cristina, who always cared about me, supported my choices, comforted me for my mistakes. If I have grown up wiser, learnt many things, professionally and humanly is also due to the support of the people I mentioned. Having the possibility to share my path with such people has made my life a wonderful one.

Summary

Deep Neural Networks (DNNs) are systems resilient to numerical perturbations and architectural imprecision. This is proven through a regular performance even after aggressive pruning, quantization, and other compression techniques [16], which significantly reduce either the number of parameters in the network or their precision. However, recent works [41][40] have shown that these networks are vulnerable to surgical bit-flips in specific locations. In [34], fault-injection techniques are used to modify one bit or more than one bit stored in SRAM or DRAM to bring the system to misclassification. Moreover, system level threats called adversarial attacks [13] have shown effective ability to induce behavioural anomalies in DNNs. In fact, DNNs are vulnerable to modified inputs crafted to yield erroneous labels, while being undetectable to human observers. In safety-critical applications such as transportation systems, adversarial examples could be a non negligible threat to public safety. For this reason, attacks and defences on adversarial examples draw great attention in the scientific community. On the other hand, due to the ubiquity of machine-learning, attacks from the supply chain such as hardware trojans emerged as a threat to DNNs security.

Spiking Neural Networks (SNNs) constitute a biologically plausible alternative to DNNs, because the event-based communication model between neurons resembles more inherently the human brain. The biological affinity brings lots of advantages. SNNs can rely on the spiking times as a source of information therefore naturally lend themselves to be used in temporally-dependent contexts. Furthermore, learning methods suited for SNNs mimic more coherently the learning processes in human brain and can be employed on-line in state-of-the-art neuromorphic hardware implementations. Moreover, due to the asynchronous and spike-based propagation, the SNN neuromorphic hardware is naturally more energy-efficient than classical DNN hardware, as shown with success by neuromorphic chips like Intel Loihi [9], IBM TrueNorth [37] and ARM SpiNNaker [11]. An analysis of statistical and then targeted fault-injection of soft-errors is carried on. The amount of faults injected is minimized taking advantage of a *gradient search algorithm* which is capable of finding few vulnerable weights among the huge amount of weights of a neural network. Then, the point of view of an attacker with a

white-box knowledge of the system has been assumed: the knowledge about vulnerable weights is exploited to craft a new attack vector that threatens the integrity of both DNNs and SNNs. A cross-layer and layer-wise attack against neural networks that transforms a circuit level vulnerability to a system level security flaw is proposed. Memory bit-flips in neural networks' parameters are exploited, through a hardware trojan triggered using a surgical adversarial attack. The work is organized as follows:

- **Chapter 1**: the introduction to the work is provided highlighting motivations, scientific challenges and novel contributions of it.
- **Chapter 2**: the main concepts of *Artificial Intelligence*, *Machine Learning* and *Deep Neural Networks* (DNNs) are presented. A focus is made on deep learning architectures and learning algorithms.
- **Chapter 3**: an overview of attack strategies is provided explaining which are the sources of threaten for an *integrated circuit* (IC) in general and, specifically, for neural networks architectures. State-of-the-art attack techniques are provided.
- **Chapter 4**: the main concepts of *Spiking Neural Networks* (SNNs) are shown. Motivations and advantages are highlighted. Finally, different learning methods are discussed; weaknesses and strengths of each one are discussed. SNNtoolbox operation, which is ultimately used in the present work to simulate the behaviour of a SNN, is introduced.
- **Chapter 5**: an analysis of the resilience of DNNs and SNNs when subject to errors in the network's parameters is performed. Statistical and targeted (through the *gradient search algorithm*), cross-layer and layer-wise analysis are carried on on three different datasets and four networks and results are discussed in a critical way.
- **Chapter 6**: the build up of a methodology for triggering a hardware trojan attack remotely through an adversarial input pattern is performed. Moreover, the built-up of methodology guidelines for crafting an input trigger capable of yielding maximal *stealthiness* and *effectiveness* are taken into consideration.
- **Chapter 7**: the results obtained are shown and analysed. Considerations about the hardware overhead reduction are carried on.
- **Chapter 8**: the results obtained are summarized and possible future work is discussed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Nowadays, neural networks have been demonstrated to perform with outstanding accuracy into many fields and are being integrated in many human life tasks, pervading human lives. However, the assimilation of neural networks in safety-critical applications such as autonomous driving, healthcare, IoT, etc... is raising clear concerns. It has been increased the worry that these networks can be easily fooled to produce dangerous results, with undeniable consequences on security and final applicability. The attack of a neural network can be deployed in different stages from training, through production, to the final application, with different mechanisms.
An adversarial attack is a deliberate modification of the input of a neural network which leads the neural network itself to misbehave (reduction in accuracy or confidence). It has been demonstrated that neural networks can be easily fooled by adversarial attacks [54] [57] [13], which consist of imperceptible perturbations applied on the inputs. Attacks of this kind and related countermeasures have been extensively investigated.
The more vivid interest in adversarial attacks with respect to attacks on internal parameters of the network is justified by the fact that DNNs have shown excellent resilience when internal parameters are subject to errors or approximations (soft-errors, pruning, quantization), as demonstrated in many works [46][47][17]. However, attack vectors against internal parameters should not be underestimated. Sources of errors like soft-errors are unintended, rare and randomly directed, whereas pruning and quantization are approximations intended to reduce the complexity but must be limited to avoid the impairment of the system itself. On the contrary, an ill-intentioned could, in some way, target specific sensitive weights, insert errors and crush the reliability of a neural network. The targeted **fault-injection** of errors, often used as a test vector for general ICs, when targeted to neural network's internal parameters is highlighting the weaknesses of neural networks when

subject to targeted attacks in internal parameters. Both high-resilience of DNNs-SNNs to random errors in parameters and low-resilience of DNNs-SNNs to targeted errors in parameters will be demonstrated in Chapter 5.
Another potential worrying source of attack could be the hardware trojan attack. Due to the current silicon production chain, when a neural network is implemented in hardware, it undergoes different steps from design, through fabrication, validation/testing until the final field application. Different parties are involved to shorten the production chain and to let each part focus on their expertise. However, the fact that there are so many parties involved, opens the way to malicious attackers from different sides. It is clear that an attacker can target one or more of these steps to deploy an attack to the network, reducing, for example, the classification accuracy and compromising the reliability. If an attack is deployed during the supply chain, by insertion of hardware, an **hardware trojan attack** is performed. The hardware can be crafted to be undetectable during testing, and then be activated by external means during the application phase. If this happens, a **hardware trojan backdoor attack** is performed. The concern about hardware trojan is severe, for example, "in 2007, a Syrian radar failed to warn of an incoming airstrike; a backdoor built into the system's chips was rumoured to be responsible" [38].
On the other hand, *Spiking Neural Networks* (SNNs) have received an increasing interest in neuro-science and deep-learning for many reasons. First of all, SNNs are biologically-plausible implementations of DNNs, which try to reduce the gap between computation science and neuroscience, providing a more realistic model of operation of the human brain, as well as learning models. Different models for biological spiking neurons have been developed and studied to both mimic the actual information transport of brain cells, providing rich firing patterns, and to be computationally supportable. Moreover, due to their spiking activity, SNNs can leverage the spike timings as an information source and can be successfully applied in intrinsically time-dependent problems or being integrated with event-based sensors. Finally, for their intrinsic spiking activity, SNNs naturally lend themselves to low-energy implementations [36] and are expected to gain more and more interest in the future. Given these many advantages, SNNs are and are believed to become even more a promising alternative to classical DNNs in many AI fields. Hence, it is clear how the problem of SNN security must be carefully addressed so that potential breaches can be detected and solved. Although many studies have been devoted to the analysis of the security of classical DNNs, few works have been carried on about security issues for SNNs.

1.2 Scientific Challenges

Perturbations of the data of an IC, stored in memory elements, can degrade its performance and has to be carefully taken into account in some applications (servers, medical-equipments, etc...). In fact, since the output of an *Artificial Neural Network* depends on both inputs and internal parameters, keeping them as clean as possible is necessary to reach satisfactory accuracy and confidence, which are mandatory if the system has to be applied on safety-critical tasks. However, as previously anticipated, perturbations on internal parameters (soft-errors) are generally considered to be negligible since they are not affecting the system performances, due to the high resilience shown by neural networks when subject to these kinds of errors. Nevertheless, neural networks have been demonstrated to be far less-resilient when targeted perturbations on the internal parameters are applied.
Studying the resilience of a network when subject to errors in internal parameters is not an easy task if, for example, a fault-injection analysis is carried on. If it is wanted to highlight which are the sensitive weights, it would be extremely time-wasting and eventually not feasible to inject faults in each parameter and record accuracy variations. In the following, a systematic fault-injection attack, targeting the parameters of the network, will be deployed. It will be shown, in section 5.2, how this setup can identify which are the most vulnerable weights of the network. In Chapter 6, errors in these sensitive weights are exploited as payload circuitry for a hardware trojan; the methodology to deploy an effective hardware trojan is presented. However, the initial hardware overhead is considered to be non-negligible. Therefore, reducing the initial hardware trojan overhead as much as possible, both in terms of area and power consumption was mandatory. In this way, the hardware trojan can escape sophisticated hardware trojan detection and diagnosis techniques. A non-trivial challenge has been to keep the input trigger pattern imperceptible.

1.3 Novel Contributions

Much effort has been devoted to the study of security issues of DNNs, especially on adversarial attacks. Nevertheless, just a few works have been carried on about security issues related to Spiking Neural Networks. At the beginning, reliability of DNNs and SNNs is considered, carrying on experiments to simulate both soft-error resilience and fault-injection resilience. In particular a *Statistical Fault-Injection* is carried on in section 5.1, whereas a targeted *Fault-Injection* is carried on in section 5.2. These last experiments will show how targeted perturbations in some parameters of the network, i.e. sensitive weights, can severely reduce the reliability. In the following, the analysis is extended to many networks and three different datasets

and a critical analysis is carried on to highlight differences, weaknesses and strengths of them. Besides, the analysis is carried on both on a cross-layer basis and on a layer-wise basis, underlining how a global analysis could be not always sufficient to have a complete view of the weaknesses of a neural network. Finally, the results obtained in section 5.2 are leveraged in Chapter 6 to craft a hardware trojan triggered by an input pattern. To the best of my knowledge, the first tentative to threaten the security of a SNN through a hardware trojan hidden in the supply chain triggered by an input pattern added to the image is carried on. The main contributions of my work are summarized and connected in Figure 1.1.

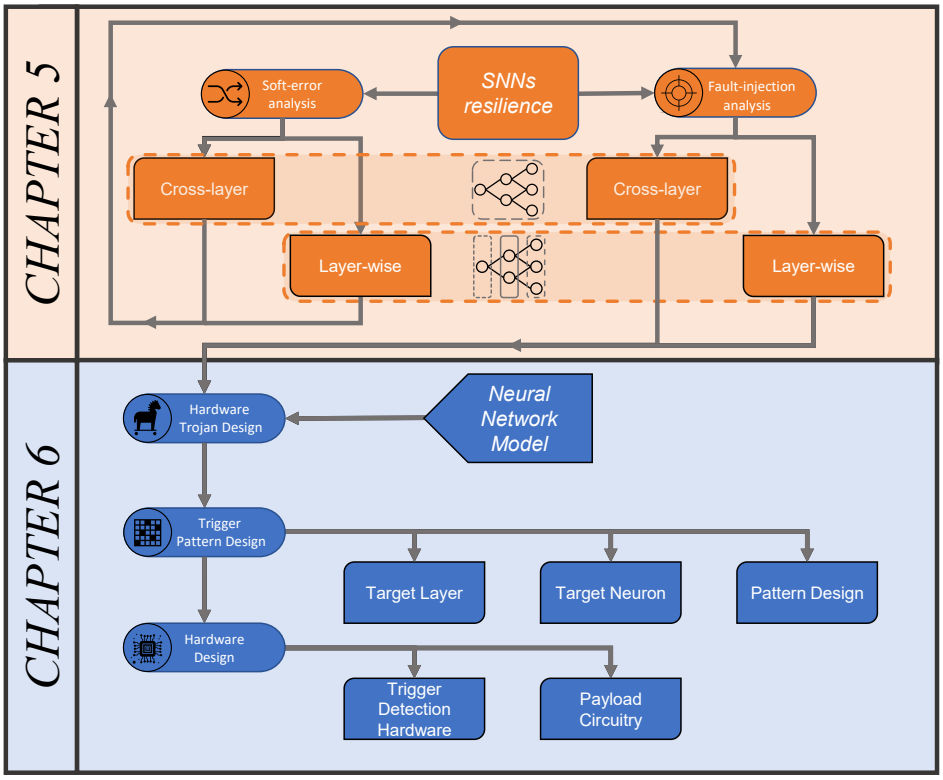

Figure 1.1: Structure organization of the thesis.

Chapter 2

From artificial intellience to deep learning

2.1 Artificial Intelligence

Oxford Dictionary defines *Artificial Intellingence* (AI) as *"The theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages"*.

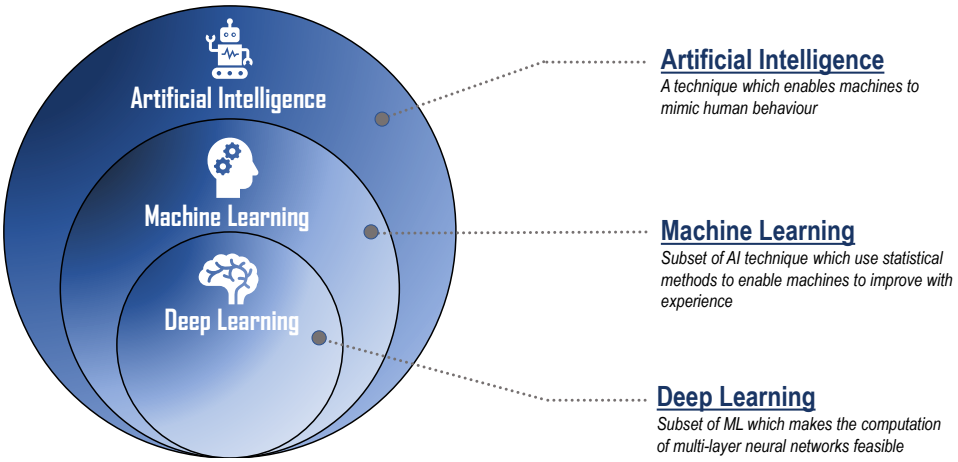

Figure 2.1: *Artificial Intelligence* taxonomy [Source: edureka.com].

AI is an extended subject embracing many different science areas from biology to electronics and targets many different tasks like autonomous driving, IoT, videogames, etc... A crucial subset of AI is machine learning.

2.2 Machine Learning

Arthur Samuel in 1959 defined machine learning as a *"Field of study that gives computers the ability to learn without being explicitly programmed"*. A more rigorous and recent definition from is the one from Tom Mitchell in 1998 which states *"A computer is said to learn from experience E with respect to some task T and some performance P, if its performance on T, as measured by P, improve with experience E"*. Generally speaking, what is wanted to do is to build up a model $f(X, \Theta)$ that has to predict some particular result Y given a set of inputs features X and having some set of parameters Θ, so that

$$Y = f(X, \Theta).$$

The model is trained on a subset of X, X_A, so that during the learning process it can update parameters Θ to match the correct prediction. If correctly trained, when the model is applied to the unseen dataset X_B, it can correctly predict the results.
A first differentiation for machine learning problems is on how the learning phase is carried on:

- In *supervised learning*, each input data is associated with a particular label, which is the final value for the prediction, which is known and is used to drive the learning phase.
- Given an *unsupervised learning* algorithm, it is capable to find, without any labels, a particular structure in the dataset. A clustering unsupervised learning algorithm, for example, is capable to define clusters of input data, given an unlabeled input dataset.
- *Reinforcement learning* procedure combined with a Neural Network architecture enables the network itself to take the best decision in a software *environment*. In practice, the *agent*, i.e. the neural network, is put in a software environment, which is in a particular state S_t at the time t, and take actions on it. The environment is calculating its next state S_{t+1} based on the action of the agent. The outputs from the environment to the agent are the updated state of the environment S_{t+1} and the *reward*, that it's a value that measures the success or failure of the agent's actions. This reward values are updating the *policy* of the agent, which is the criterion for which the agent is deciding which is the next action to take.

The problems that a machine learning algorithm addresses can be of two types. When dealing with a *regression* problem, the outcomes, the values to be predicted, are real numbers whereas in *classification* problems the outcomes are discrete values. Classification is the process of categorizing a given set of data into classes which are usually referred to as *target labels* or *categories*.

2.3 Deep Learning

The combination of high computational power and a large number of images belonging to different datasets led to the outstanding development of deep neural networks. GPUs, FPGAs or dedicated accelerators are extensively applied to provide a significant speed-up for both the learning and inference phase. The human brain is composed of tens of billions of neurons, which are the basic computational units of the human brain. The artificial structure that models the biological human brain structure is called *Artificial Neural Network* ANN. A Deep Neural Network is created by stacking more groups of neurons (layers) in a hierarchical way so that the output of one layer is the input of the following one. This enables a network to build up more levels of abstraction, which in turn gives the possibility to solve very complex features recognition tasks.

2.3.1 From neuron structure to perceptron

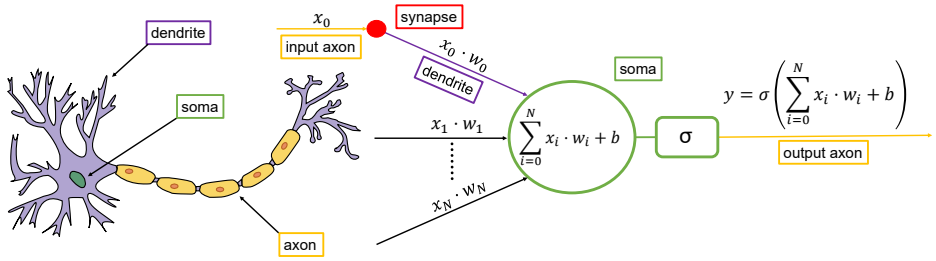

Figure 2.2: Biological neuron structure (left) [source: wikipedia.org] and artificial neural network neuron representation (right).

As previously said, the biological neuron is the basic computational element of the human brain. A biological neuron is composed by the *soma*, also called body, *dendrites* and *axon* (Figure 2.2 left). The dendrites are the inputs of the neuron and the soma is the actual computing unit, which takes the inputs and processes them to produce the output, that is sent through the axon which is the output connection. Different neurons are connected through synapses, so that, by convention, if two neurons are connected together a *pre-synaptic neuron* and a *post-synaptic neuron* can be defined. By connecting multiple neurons via synapses, very complex patterns with high computational capability can be created.
In an *Artificial Neural Network* (ANN) as well, the artificial neuron is considered the basic computational node and the same structure and taxonomy is replicated (Figure 2.2 right). When the *axons* (outputs) of an amount N of neurons are connected via *dendrites* to the soma of another post-synaptic neuron, this whole computational unit is called *perceptron*. The output neuron receives as input the signals multiplied by the *synaptic weights*; then

the results are summed together and a bias b is added. Finally, a non-linear function σ is applied to the result [12], so that the output of the neuron is:

$$y = \sigma\left(\sum_{i=1}^{N} x_i \cdot w_i + b\right).$$

In Figure 2.2, on the right, a sketch of the neuron structure and its computations are shown.

2.3.2 Multilayer Perceptron

The *Multilayer Perceptron* (MLP) is given by the stack of multiple layers of basic computational units named perceptrons, described in section 2.3.1. The neurons are connected in a dense (or fully-connected) fashion, so that each neuron in layer l receives, as inputs, the outputs of each neuron in the previous layer *l-1*. A MLP is composed by, at least, one input layer and one output layer plus one so-called *hidden layer* in between. The amount of synapses and related weights connecting one layer to the previous one is given by $n_{l-1} \cdot n_l$, where n_l is the number of neurons in a given layer l. In Figure 2.3 the structure of a MLP with two hidden layers is represented.

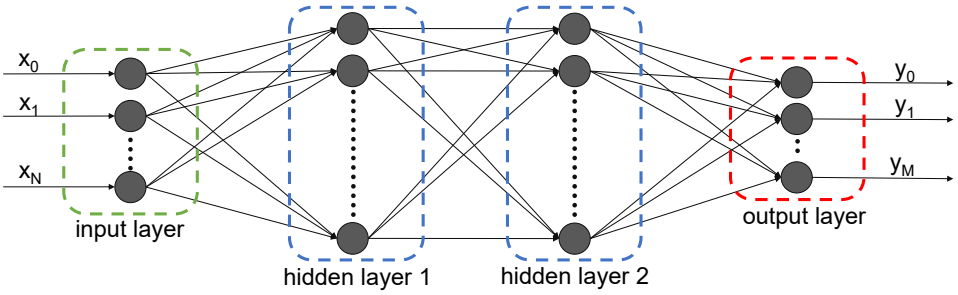

Figure 2.3: Structure of a *Multilayer Perceptron* with two hidden layers.

2.3.3 Convolutional Neural Network

With *Convolutional Neural Networks* (CNNs), additional type of layers are introduced: *convolutional layers* (Figure 2.4-a) to extract features from the input image and *pooling layers* (Figure 2.4-b) to progressively reduce the dimensionality of the data.
Convolutional layers, whose weights are updated during the learning phase, exploit the convolution of filters with the input image. When a filter of size f is applied to a region of the input image, each pixel value of the image is multiplied by the corresponding filter value and the multiplication results are summed together to produce the output pixel value. In Figure 2.4-a, the

example of the computation of a kernel with size *f=(3,3)*, applied to an 8×8 image is shown. In order to build deep neural networks one modification to the basic convolutional operation that is needed to be used is *padding*. The reasons to apply padding are mainly two:

- It is not wanted to down-size the image at each convolution layer because for networks with a lot of layers the dimensionality in the last layers would be reduced too much.
- The pixels at the borders (edges, corners) of the input image are included in the convolutions much less than pixels in other areas of the image so a lot of information about the edges of the images could be removed.

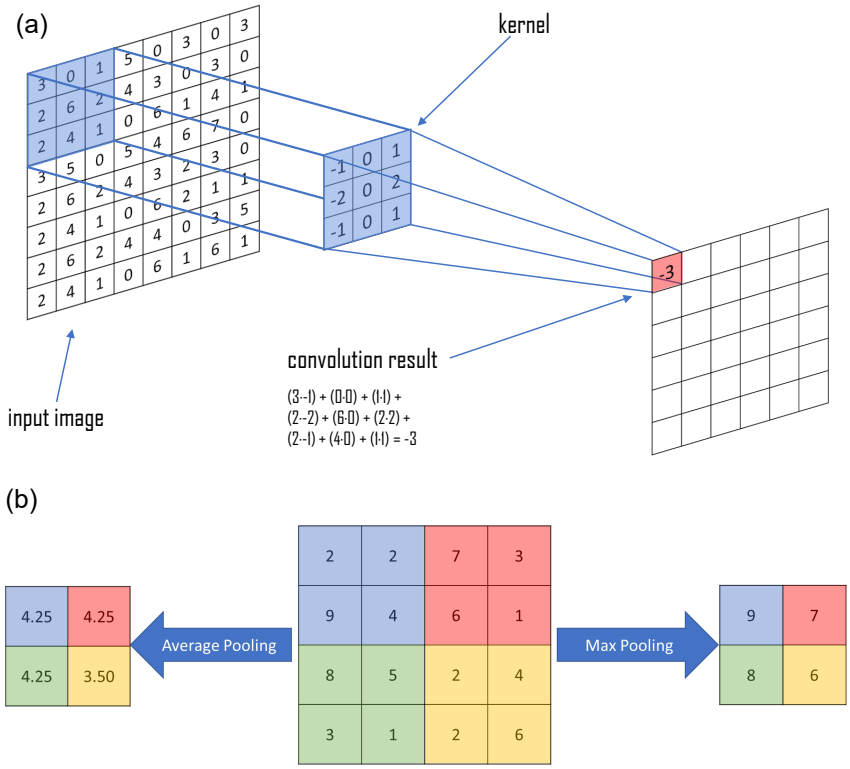

Figure 2.4: Example of the computation (a) for a 3×3 kernel of a *Convolutional Neural Network* [source: towardsdatascience.com] and (b) for average and max pooling layer applied on a 4×4 image.

Zero-padding an image by a factor p means adding a frame of thickness p around the image. The filter is then moved on the image in steps defined by the *stride* parameter to produce the so called *feature map*. In Figure 2.5, four steps of convolution process for a 8×8 input image and a filter with *f=(3,3)* and *s=(1,1)* is shown. It can be clearly seen that the output map

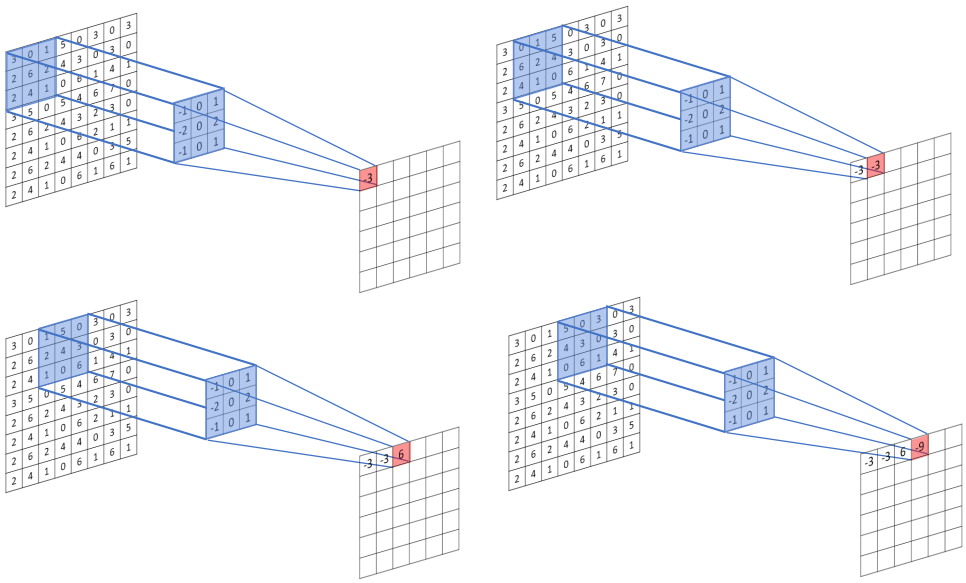

Figure 2.5: Four steps of convolution for a 3x3 kernel of a *Convolutional Neural Network* [source: towardsdatascience.com].

has a reduced dimensionality, in fact, in the example shown in Figure 2.5, the output is a 6×6 image, since the zero-padding is not applied.
Generally, after a convolutional layer, a *pooling layer* is present in order to reduce the dimensionality of the image and speed-up the computation. Hyperparameters for the pooling layer are the filter size f and the stride s, which are usually taken as f=s. There are mainly two types of pooling:

- **Max Pooling**: The image to pool is divided into regions and for each of them the maximum value of the pixel is taken as output (Figure 2.4-b on the right).
- **Average Pooling**: The image to pool is divided into regions and for each of them the average of the values is taken as outputs (Figure 2.4-b on the left).

Given an input image of $n_H \times n_W \times n_c$, the size of the output image will be given by:

$$\left(\frac{n_H - f}{s} + 1\right) \times \left(\frac{n_W - f}{s} + 1\right) \times n_c,$$

where $n_H \times n_W$ is the dimension of the image and n_c is the amount of channels.

The complete scheme of a LeNet-5 CNN [26] is shown in Figure 2.6. At the second convolutional layer, the inputs are the feature maps from the first convolution layer, after pooling. Each filter is applied on all the *input feature maps* and the results are summed so that the amount of *output feature maps* is equal to the number of filters.
Convolutional Neural Networks, leveraging the properties of the convolutional layers, shown excellent capabilities of extracting features present in the input images. This trait led to reach outstanding performances in many image-recognition and classification tasks. The LeNet-5, for example, achieved excellent performances in terms of accuracy in classifying images belonging to the MNIST dataset. One image from each class of the MNIST dataset is reported in Figure 2.7.

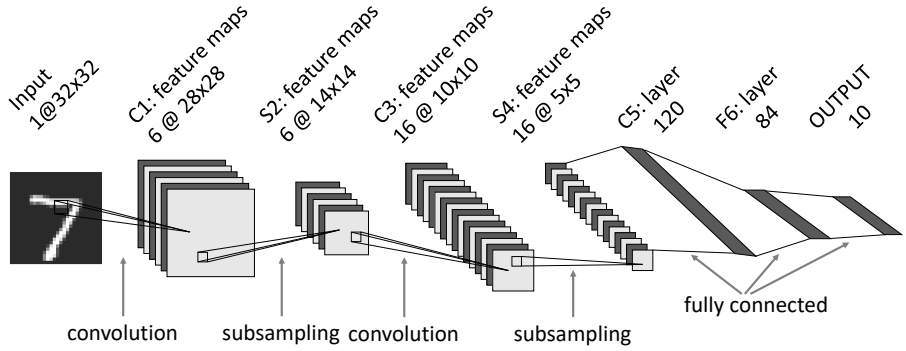

Figure 2.6: Structure of a LeNet-5 *convolutional neural network* [26].

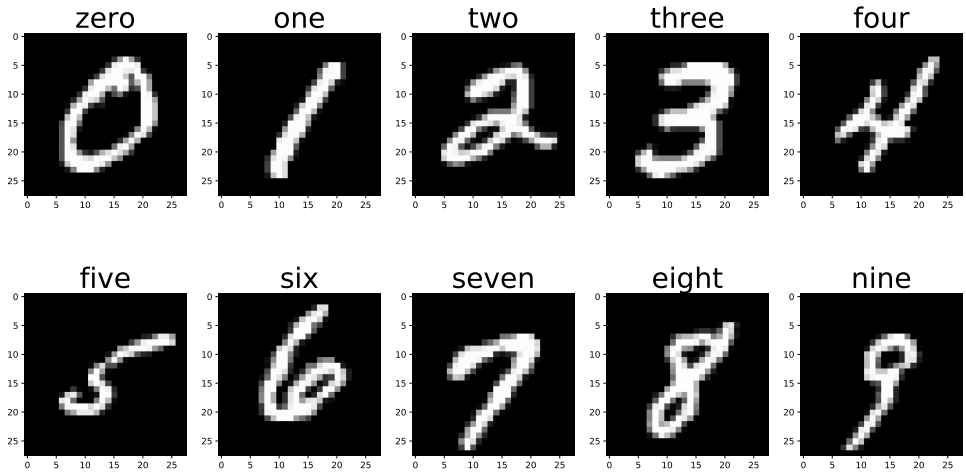

Figure 2.7: Sample images from the *MNIST* dataset with labels representing the corresponding class.

The outstanding results obtained by the CNNs in image-recognition tasks are due to the capability of CNNs to recognize features in the input image. In the first layers, simple features like lines, circles, edges, etc... can be detected. In deeper layers, these features are combined together to detect more complex features up to the end where the image is finally classified. In Figure 2.8, features derived from a face recognition net in different layers are displayed.

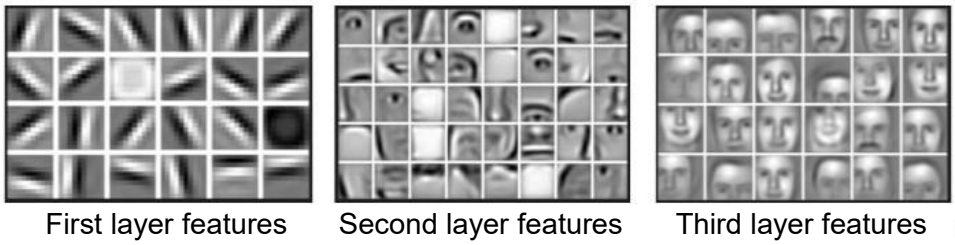

Figure 2.8: Features learnt from a face recognition net [29].

2.3.4 Recurrent Neural Network

Recurrent Neural Networks, that is, neural networks with a recursive connection, not only learn during training but also exhibit memory capabilities that make them establish relationships between input data. The output results at a particular time t are not only a function of the inputs and internal parameters but also of the vector representing the previous input-output relation. In this way, the same input can produce different output results depending on the different contexts.

2.3.5 Supervised Learning Methodology

The initial crucial step of a neural network is the learning phase. Supervised learning can be applied through the *gradient descent algorithm* together with a *backpropagation mechanism* for the error.

Gradient Descent

The *Gradient Descent* is an algorithm to find the minimum of a differentiable function. Given a single variable function $f(\theta)$, to minimize its value, a minimum must be found, so that

$$\frac{\partial}{\partial\theta} f(\theta) = 0.$$

A random initialization of the parameter θ is applied and the derivative in that point is calculated, then, as shown in Figure 2.9 left, the *gradient*

descent algorithm proceeds by modifying the value of θ in the direction of the negative value of $\frac{\partial}{\partial\theta}f(\theta)$. The value of θ is updated proportionally to the value of the derivative to prevent overshooting. After some iterations, the gradient descent algorithm ends in a minimum, which could also be local, depending on the starting value of parameter θ. For a bivariate function $f(\theta_0, \theta_1)$, the gradient descent algorithm is graphically presented on the right in Figure 2.9. What is wanted to know in this case, given a starting point for θ_0 and θ_1, is the direction, i.e. the value of the two parameters, to take for which the value of the function is minimized. The direction and the value of steepness is provided by $\nabla f(\theta_0, \theta_1)$.

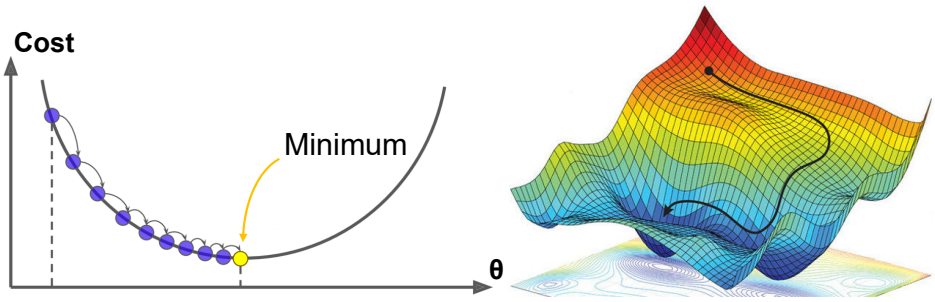

Figure 2.9: Gradient descent for monovariate function (left) and bivariate function (right) [source: www.datasciencecentral.com].

The concept can be extended to a multivariate function with an amount P of parameters $f(\theta_0, \theta_1, ..., \theta_P)$. The gradient descent algorithm is used during the learning phase of a neural network for the minimization of the cost function C by iteratively updating the set of weights and biases which constitute the parameters of the network. For each image, given as input during the training phase, a cost function can be calculated. The value of the cost function, averaged over a number M of training images, is given by the sum of the squared errors divided by the number of images

$$C = \frac{1}{M}\sum_{k=0}^{M}(e_k)^2,$$

where C is the cost function, M is the number of images and e_k is the error related to the k-th image. In *supervised learning*, the class of the image is known and is defined by a *label* associated to the image. Given N classes, hence N output neurons, a N-length zero vector $\overline{y}$ with 1 in position corresponding to the class of the image is associated to each image. The value e_j is the error for each training image and can be therefore expressed by the following formula:

$$e_k = \sum_{j=0}^{N}(x_j - y_j)^2,$$

where x_j is the activation of the j-th neuron and y_j is the label of the j-th neuron.

Learning through backpropagation

After the forward propagation of an image, that is the calculation of the outputs of the neural network when an input image is provided, the backward propagation has to be applied. *Backpropagation* is a method to calculate the gradient of the cost function C with respect to all the weights and biases of the neural network, so that it would be possible to finally apply the gradient descent algorithm. Generally, the training set is divided in *batches*, each one containing an amount of images defined by the *batch size*, which are processed in parallel. The parameters of the network are updated after the forward propagation and backward propagation of each batch of the training set. For each image in the batch it is possible to derive the cost function C_k where h is the index of the image in the batch. Suppose $k=0$, it can be derived:

$$C_0 = \sum_{j=0}^{N_L-1} (a_j^{(L)} - y_j)^2,$$

where $a_j^{(L)}$ is the activation at layer L, that is the output layer, for neuron with index j, and where y_j is the label for the j-th neuron. The total number of neurons at the last layer is N_L and it equals the number of classes for the classification. For each neuron of layer L the output would be

$$a_j^{(L)} = \sigma\Big(z_j^{(L)}\Big),$$

where

$$z_j^{(L)} = \sum_{i=0}^{N_L-1} w_{ji}^L \cdot a_i^{L-1} + b_j,$$

so that $a_j^{(L)}$ is the output $z_j^{(L)}$ after the application of the activation function σ. The derivative of the cost function with respect to the weights in layers L is given by:

$$\frac{\partial C_0}{\partial w_{ji}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{ji}^{(L)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}} \tag{2.1}$$

The derivative of the cost function with respect to the activations of layer *L-1* is

$$\frac{\partial C_0}{\partial a_{ji}^{(L-1)}} = \sum_{j=0}^{N_L-1} \frac{\partial z_j^{(L)}}{\partial a_i^{(L-1)}} \cdot \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \cdot \frac{\partial C_0}{\partial a_j^{(L)}}, \tag{2.2}$$

since the output of a neuron influences the cost function through different multiple paths. These are the functions of the *chain rule* of the backpropagation algorithm, which are applied one after the other to find the derivative of the cost function with respect to the weights of that layer (equation 2.1) and the derivative of the cost function with respect to the activation function of the previous layer (equation 2.2).

Chapter 3

Background and Related Work

In this chapter, the description of background studies on threathens for ICs is carried on. Reliability issues due to unintended soft-errors, intentional fault-injections used as test vector for ICs and trojan attacks are considered. This issue is then related to the case of neural networks' hardware implementation. Regardless the actual hardware implementation of a neural network (FPGAs, GPUs, dedicated accelerators), large memory elements, storing the networks' internal parameters, are a must, and the related vulnerability will be analysed at system level in Chapter 5. Despite the generally high resilience shown by neural networks concerning soft-errors on network's parameters, a well-targeted fault-injection may be very effective, compromising the reliability of a Deep Neural Network as well as a Spiking Neural Network. A reduction of classification accuracy in safety-critical applications, such as automotive or healthcare, is not acceptable and could yield to catastrophic situations.

3.1 Soft-errors

Bit-errors in ICs can occur for different reasons. If a bit-error is occurring and is corrupting the data but not damaging the device in which the data is present, then a *soft-error* is occurring. A soft-error can be temporary or permanent. Soft-errors are caused by mainly two phenomena:

- *Alpha particles* emitted by the integrated circuit or by its package
- *Cosmic neutrons* generated when atoms in the Earth's atmosphere get hit by cosmic rays [20]

The ionization of the hit material by a highly-energetic alpha particle, can generate charges, i.e. electron-hole pairs, along the path of the particle itself.

Neutrons are devoid of charge but can deposit charges in their traces as well, by nuclear interaction with the target material, leading to the same effect of an alpha particle direct ionization. The two phenomena are shown in Figure 3.1-a.

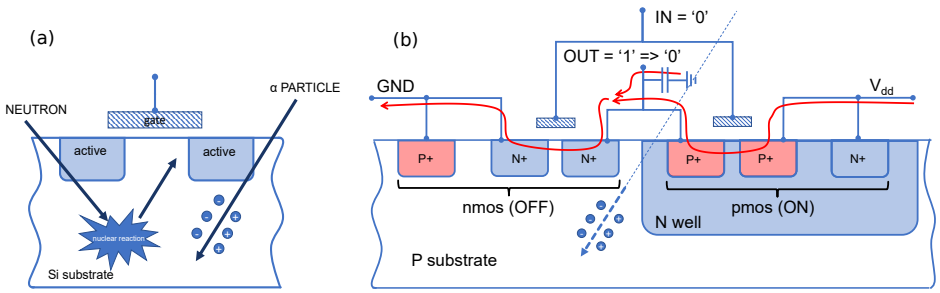

Figure 3.1: Soft-error phenomenon for both alpha particles and neutrons colliding with a silicon substrate of an IC (a), effect of soft-error on an inverter (b).

Radiations can affect electronic devices in several ways, however, the one here considered is called *Single Event Effect* (SEE) and is caused by a single event of a particle collision with the device, resulting in a permanent or temporal failure [5]. SEE can, in turn, be divided into different categories

- *Single Event Transient* (SET): is a voltage spike occurring in an area of an IC due to the collision of a particle with it.
- *Single Event Latch-Up* (SEL): is the result of a collision of a particle with a CMOS structure, resulting in a current injected which sustained by the positive feedback of CMOS parasitic structures.
- *Single Event Burnout* (SEB): caused when a self-sustained current is generated by a particle strike and is entirely localized in a minimal area and eventually melts or cracks the device itself.
- *Single Event Gate Rupture* (SEGR): is caused by a single high-energy gitting particle generating the rupture of the gate oxide of a MOSFET, with consequently increased gate leakage and the degradation or the complete failure of the device itself [5].

Specifically, the SET is when the event is interacting with a combinational circuit, whereas an event that is interacting with a memory element, ultimately flipping the content of a memory location, is called *Single Event Upset* (SEU) [5]. If the memory location is not overwritten, this phenomenon leads to a permanent error in memory.
When a cosmic neutron or an alpha particle is colliding with an IC, it is generating charges due to photoelectric effect. These charges are then transported and collected by the structures of the IC, mainly p-n junctions, by

drift and diffusion phenomena, so that a spurious current is generated, which in turn generates a current glitch in a sensitive area of the chip. An injection of charges in the substrate of the OFF transistor in an inverter, for example, caused by a high-energy hitting particle, can temporarily flip the correct output value. For example, as can be seen in Figure 3.1-b, if an inverter with IN='0' and OUT='1' is considered, during correct operation, the nmos is OFF and the pmos is ON. If the high-energy particle collides with the p-substrate, the generated current can temporarily cause a switch-on of the transistor. The inverter's output value is consequently pulled-down; if this happens in one of the inverters of a SRAM memory cell, it can eventually flip the stored content of it [1].
DRAMs have been proven to be resistant to SEUs, differently from SRAMs. A metric for SRAMs' soft-error measurement is the *Soft-Error Rate* (SER) measured as the number of failures per area per a certain amount of time. SER is expressed in *Failures In Time* (FIT) units, where 1 FIT corresponds to 1 failure in 10^9 device hours. In well-designed systems, the soft-error rate is kept very low. For some applications which are making use of large memory elements (high-end servers) or which need a high reliability level (automotive, healthcare) the soft-error rate and its implications must be carefully addressed. In fact, in the first case, also a minimal amount of soft-error rate could lead to frequent system crashes, which, of course, must be avoided while in the second case could lead to dangerous consequences for human beings [19].
While the cell topology and layout significantly affect SER [5], it is quite clear that with the *Very Large Scale Integration* (VLSI) driven by Moore's law, the concern about soft-errors in the hardware implementation is growing. This is primarily due to two reasons:

- smaller feature size and higher densities [20]
- reduction of the supply voltage, since the *critical charge* for which the content of a memory cell is flipped is directly proportional to the reduction of the supply voltage [10]

Soft-errors are typically measured through experimental setups (*System SER*) which include the *Device Under Test* (DUT), for example a SRAM memory, and a source of irradiation whose irradiation flux is known. The sensitivity of a memory element to radiation is derived by exposing the device under test to a source of radiation and analysing the differences in data stored in it afterward; of course, to calculate the sensitivity, the original pattern written in the memory must be known, as well as the radiation flux [5]. The neutron-induced SER is proportional to the neutron-flux which increases as a function of the altitude (10× each 3 km and saturates at about 15 km) so that at high altitudes the neutron-induced effect dominates the

alpha effect [20]. It is possible to measure the *Accelerated SER* (ASER) with a high irradiation condition to drastically reduce the testing time. In this case the neutron and alpha-particle component must be measured in a separate way [5]. There is also the possibility to simulate the effect of soft-errors via software simulations at device level [1].
When more than one adjacent cell is flipped, an event called *Multiple-Bit Upset* (MBU) is taking place. This event is related to the incidence of some particles which have such high energy to provoke the flip of two adjacent cells or can be related to an increasing angle of incidence of the hitting particle [20]. SEUs can be corrected with *Error Correcting Codes* (ECCs) which are powerful since they can be applied at high level, with a compromise in area overhead, without changing the production masks. However, MBUs are nowadays the main problem for memory elements since these types of errors cannot be corrected with ECCs like the Hamming code, which is supposing that a single error per word can be corrected. Another technique that is capable of inducing hardware errors is the fault-injection, described in the next section.

3.2 Fault-Injection

Fault-injection is a *testing technique* applied to study how ICs behave when subject to faults, how errors propagate in them and generally which is the response of an IC when stressed to operate in unusual conditions. Fault-injection attacks are also a popular *attacks* against cryptographic circuits and can be applied to bypass security [21][2][15]. Many different fault-injection techniques have been exploited that can be categorized as: software-based, hardware-based, simulation-based, emulation-based or hybrids [58]. Even though many techniques have been developed, the underlying fact is that it is not possible to apply all possible errors in all possible locations to have complete knowledge of the response of the circuit. Consequently most of the fault-injection techniques adopted in literature are *statistical fault-inections* (SFI) [31]. Basic examples of hardware fault-injections are shorting connections in boards (bridging faults), clock period variations or power supply glitches. Electromagnetic fault-injections can be locally focused and allows precise targeting of faults in memory, inducing spurious currents to induce the soft-error phenomena.

In Chapter 5, a simulation-based approach which is simulating fault-injections exploited as bit-flips in SRAM's memory cells, storing the parameters of the networks, is carried on. As explained, the outputs of a DNN depend on both the input images and its internal parameters. By inserting errors in the internal parameters of an image classification network (fault-injecting), it is possible to lead the system to misclassify input images. Due to the development of reliable and accurate state-of-the-art fault-injection techniques

like *laser beam fault injection* [48] and *row hammer attack* [23], it is possible to launch precise attacks against memory elements. Since the parameters of a DNN are stored in these memory elements, precise fault-injection attacks can be directed against them as well. The *row hammer attack* is capable of flipping the bits stored in a DRAM targeting a whole word-line by rapidly and repeatedly accessing (reading) that word-line and thus can be considered a hybrid attack, whereas the *laser beam fault injection* can provoke soft-errors in SRAM cells with the high precision given by the laser beam. *Laser Beam Fault Injection* is the state-of-the-art fault-injection technique, capable to induce very precise bit-flips in SRAM locations, leveraging the photoelectric effect in a silicon-based transistor. The main drawbacks are the high cost and expertise and time needed to set all the parameters of the laser beam (x, y position on the plane, laser spot diameter, laser beam energy) so that the number of faults that can be realistically applied must be kept low. Also, this is a semi-invasive attack since the package must be removed by mechanical or chemical interaction, to gain access to the silicon. By carefully setting the diameter of the laser beam is also possible to simultaneously flip different adjacent locations in the memory (MBUs, see section 3.1).
Due to the time required to tune the parameters of the laser beam to apply faults in different locations, the lowest amount of faults to test the reliability of a particular hardware, must be applied. Therefore, shattering the accuracy of a DNN in a significant way, with a low amount of faults, is a challenging task. This is due to the high resilience of neural networks which will be analysed in Chapter 5. To this aim, an efficient fault-injection technique will be used in section 5.2 and it will be shown that few tens of faults (bit-flips), associated to network's internal parameters, are sufficient to provoke a considerable reduction of performances. This is a key point since, to the best of my knowledge, just few works have been carried on about the analysis of fault-injection on neural networks [21][45].
The results of this analysis will be used to build up an efficient attack methodology through the hypothesis of a hardware trojan insertion in the supply chain plus a well-crafted input trojan trigger pattern, which could, jointly, heavily threaten the security properties of a DNN or SNN. The hardware trojan threat is presented in the following section.

3.3 Hardware Trojan Attacks

"*Hardware Trojans (HT) appeared to be a powerful hardware attack vector against ICs in recent years, capable of altering or disabling the capabilities of an IC, or leaking some sensitive information*" [32]. The general HT scheme can be the one depicted in Figure 3.2, in which some signals described as trigger inputs activate the trojan trigger, which is detected and affects the

behaviour of the IC through the payload circuitry.

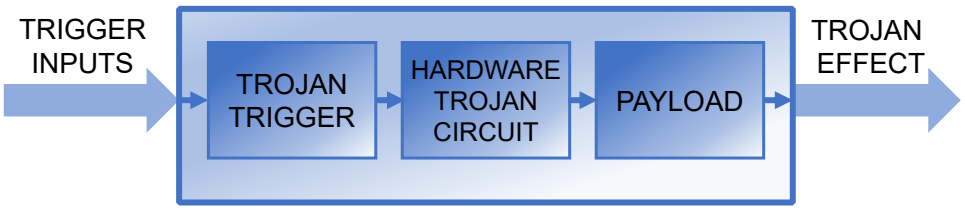

Figure 3.2: Scheme of the general hardware trojan circuitry.

HTs are designed to be silent (stealthy) for most of the time, being active just in presence of rare signals or events. These events should be chosen to be rare and unlikely to rise during normal function or during the testing phase so that the hardware appears to be completely functional. Hardware trojans can be implemented as combinational or sequential, analog or digital circuits. The hardware trojan threat is justified by the IC industry evolution which allows many different parties to get involved in the development of an IC from design through manufacture to the application phase. EDA vendors, IP vendors, SoC designers, foundries and end-users cooperate in the production of ICs and in each stage a hardware trojan can be inserted, by the company itself or by an ill-intentioned employed, without the knowledge of the other parties.

To detect or avoid the threaten of hardware trojans, different countermeasures have been developed. *HT detection* gives the possibility to understand whether a hardware trojan is present or not and *HT diagnosis* can identify where the hardware trojan is and identify the payload circuitry. Also *HT prevention* mechanisms can be applied by IP or SoC vendors to prevent the deployment of an HT from malicious instead of detecting it just after [32]. Advanced power measurement analysis, such as differential power analysis, are available to detect potential hardware trojan. Inspection of signals' delays in the integrated circuit can be carried on as an useful method for HT detection as well. Partition or segmentation based approaches have been proposed for hardware trojan diagnosis: this is due to the reduced hardware trojan overhead with respect to the total hardware of the circuit. The basic idea is to increase the impact of the HT on the circuit so that it can be easily detectable. Hardware minimization to reduce the hardware overhead is fundamental to craft a stealth hardware trojan attack.

3.4 Adversarial Attacks

An adversary, using information learnt about the structure of the classifier, tries to craft the perturbations added to the input to cause its misclassification, i.e. its incorrect classification. For explanation purposes, we consider a generic DNN for image classification. Given an original input image x and a target classification model $C(.)$, the problem of generating an adversarial example x^{adv} can be formulated as a constrained optimization [57]:

$$x^{adv} = \arg\min_{x^{adv}} \mathcal{D}(x, x^{adv}), s.t.$$
$$C(x) = l, C(x^{adv}) = l^{adv}, l \neq l^{adv}$$

Where $\mathcal{D}$ is a distance metric used to quantify the similarity between two images, and the goal of the optimization is to minimize the added noise, typically to avoid the detection of the adversarial perturbations. l and l^{adv} are the two labels of x and x^{adv}, respectively: x^{adv} is considered as an adversarial example if and only if the label of the two images are different ($C(x) \neq C(x^{adv})$) and the added noise is bounded ($\mathcal{D}(x, x^{adv}) < \epsilon$ where $\epsilon \geqslant 0$). Given an input image, the neural network can be fooled by adding a carefully crafted attack, called adversarial example, which is barely perceptible, to the image. Different types of adversarials have been explored (FGSM, JSMA [54], etc...).

3.5 Neural Networks attack taxonomy

A special taxonomy must be provided when considering attacks to neural networks. In this context, attacks can be of many types as can it is shown in Figure 3.3, depending on the combination of the phase in which the attack is supposed to take place, the object of the attack, the scope or target of the attack and the degree of knowledge that the attacker has about the attacked network.
Attacks can be directed to the training phase, by modifying the training algorithm or training set. For example, the training image set can be extended with tampered images and tampered labels so that a clean image is correctly recognized, whereas a tampered image is misclassified. This kind of attack is exploited in [14] having full knowledge of the training set. Attacks can be directed to the production, acting on parameters, operators or the architecture itself. In [56] an attack is carried on with a trigger trojan in the computational chain of the convolutional stage of a CNN implemented on an FPGA. Finally, the inference phase can be also taken into account to deploy an attack by modifying the input images with adversarials. In Figure 3.3, the attack carried on in the following is circled in shaded green;

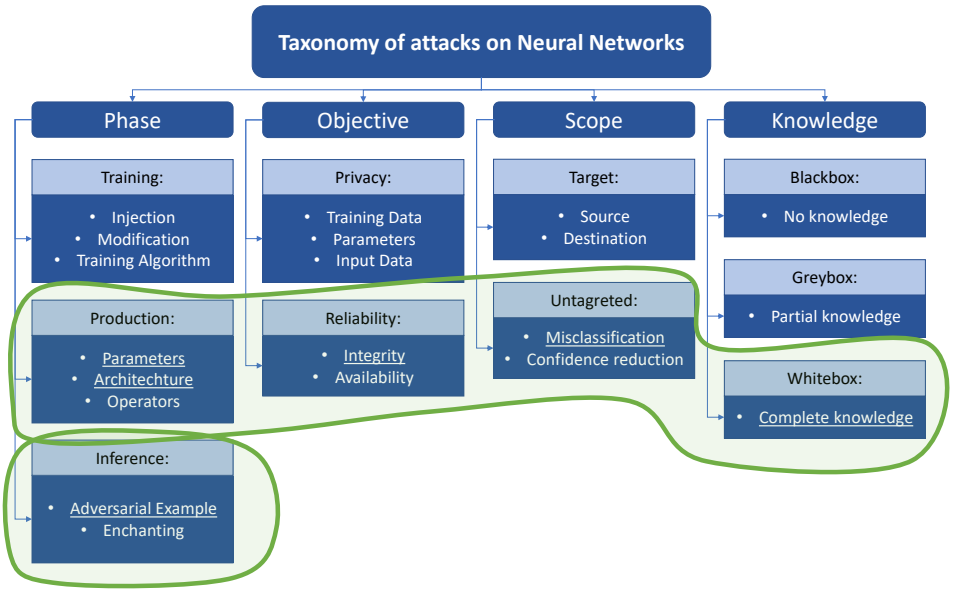

Figure 3.3: Neural networks attack taxonomy scheme [8] in which the taxonomy adopted in my work thesis is circled by a green line.

it can be seen how both production and inference phase are targeted. In fact, the hardware trojan is placed during the production phase but externally triggered during the inference phase by an input pattern applied to the image.

Chapter 4

Spiking Neural Networks

4.1 Introduction

Spiking Neural Networks (SNNs) are considered as the third generation of neural networks. SNNs, inspired by biological communication between neurons in the human brain, encode the information into spike trains, whereas the previous generations employed continuous values for the output signals of the neurons. Due to the temporal coding of spikes, SNNs are particularly prone to recognize time-dependent patterns. Moreover, SNNs, for their binary (spiking or no-spiking) operation, are suited for fast and energy efficient simulation on hardware devices [18]. In the following discussion, the continuous representation of the activations, the ones of previous neural network generations, will be also named with the term "analog" to distinguish it from the spiking representation of values in the time domain.

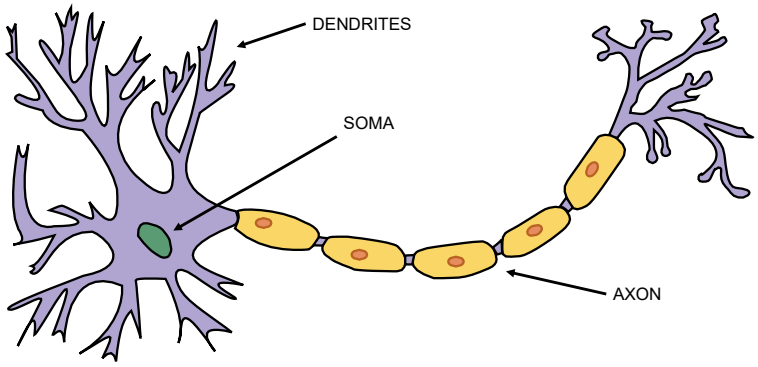

Figure 4.1: Biological model of a neuron [source: wikipedia.org].

4.2 Spiking Neuron Models

As explained in section 2.3.1, in case of ANNs the neuron is considered as a simple computational node, regardless of the effective biological electrical signals. In SNNs, the *soma*, sums the contributions of the input dendrites and, if the result exceeds a particular threshold, a spike is emitted and transmitted to the axon.
In a biological neuron, the soma is an enclosure delimited by a membrane: charges are inserted into this membrane making the *membrane potential* increase. When a spike coming from a pre-synaptic neuron arrives at the input of the post-synaptic neuron, the associated synaptic weight w_i will be integrated on the membrane. Biologically speaking, a spiking current is injected into the soma of the neuron and this contributes to raise its *membrane potential* V_m. When the membrane potential overcomes a threshold V_t, the neuron *fires*, emitting a spike on the output axon, and resets its membrane potential to a value V_R, which is the reset value for the membrane potential. In addition, the charge inside the soma is not maintained for an infinite amount of time: due to *leakage*, the membrane potential decreases continuously at the leak rate τ_m between two input spikes [6]. However, if a certain time, namely the *refractory time* T_R, is not reached after a neuron spike, the neuron cannot spike again. Many different models for the spiking neurons have been studied. These models must be at the same time (1) biologically accurate and (2) computationally simple. For example, the Hodgkin-Huxley biologically-accurate model [3] is computationally prohibitive but capable of providing rich mammal brain-like firing patterns, whereas, on the other hand the simple *Leaky Integrate and Fire* (LIF) model [53] gives the opportunity, despite being unrealistically simple, to simulate lots of neurons in real-time [22]. Other models have been developed to reach a compromise between the two requirements. One above all is the Izhikevich model [22]. However, the simple LIF model is in the following considered to explain in the details the working principles of a SNN. Due to its simplicity, the LIF model is the most popular spiking neuron model and is supported by all state-of-the-art SNNs' hardware implementation architectures.
Looking at the electrical representation of the LIF model in Figure 4.2, it can be seen that the presynaptic spike $\delta(t - t_j^{(f)})$ arrives as input at time t_j and is low-pass filtered, generating an input current $\alpha(t - t_j^{(f)})$. The soma of the neuron is modeled as a capacitance in parallel with a resistance, in fact, when a current is injected into the soma of the neuron, the membrane potential rises; this effect is modeled as a capacitance charged by a current. Some charges can leak through the soma and this effect is modeled as a leakage through a resistance. The sub-threshold dynamics of the LIF spiking neuron can be formulated as

$$\tau_m \frac{dV_m}{dt} = -V_m + R \cdot I(t),$$

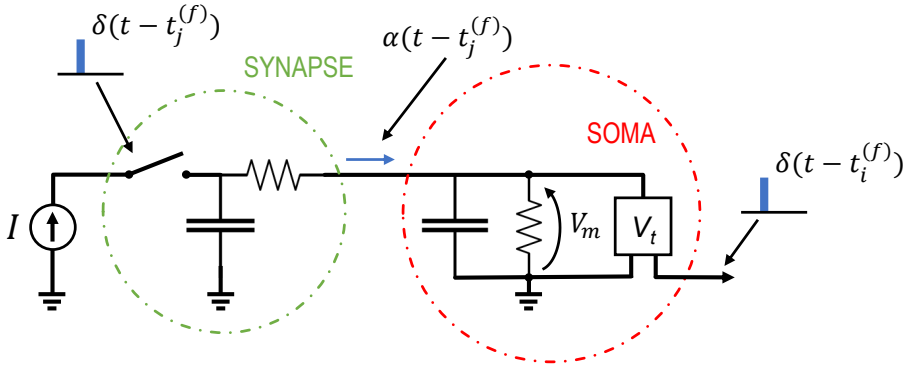

Figure 4.2: Leaky Integrate and Fire equivalent electrical circuit of a neuron.

where V_m is the membrane potential and $\tau_m = RC$ is the time constant for the membrane potential leakage [28]. When the membrane potential across the RC parallel exceeds the threshold V_t, a spike $\delta(t - t_i^{(f)})$ is generated at the output at time t_i. There are different ways in which continuous values can be coded as spikes in time domain. The most commonly used are *rate coding* and *time coding*. In the first case, an higher spiking rate (number of spikes per second) refers to an higher value in the continuous representation of the activation. In other words, the value for the activation is provided by the spiking rate, which is the amount of spikes per second. Since the spiking times are highly irregular but the mean rate, related to the mean time between events, is fixed, the spiking rates, can be described as the mean rate of a Poisson process [52]. The *time coding* can be implemented in different ways; the most popular is the *latency coding*, in which the higher activation is provided by the neuron which is spiking first, i.e. the activation value is inversely proportional to the delay of the spike. Latency coding is for sure less energy consuming since the spikes are sparser with respect to rate coding.

In Figure 4.3, it can be seen how each incoming signal from an input neuron, which is encoded in the SNN technology as a spike train, is multiplied by the weight of the synapses. The results are added together (integration) to produce the membrane potential expressed as

$$V_m = \sum_{i=1}^{N} w_i \cdot s_i,$$

where N is the number of input synapses. When the membrane potential reaches a the threshold value V_t the output neuron "spikes", or "fires".

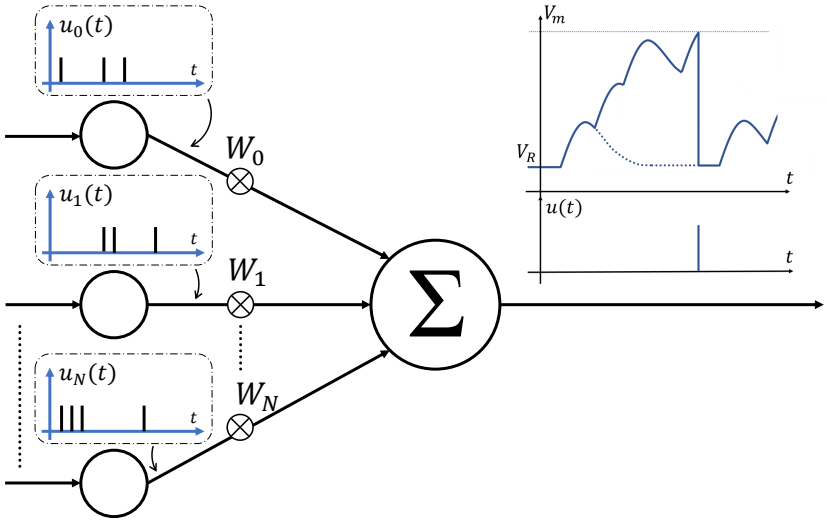

Figure 4.3: Perceptron in SNN implementation: pre-synaptic neurons and related spiking patterns (left) post-synaptic neuron (centre) and related membrane potential variation and output spiking pattern (right).

4.3 SNN learning

SNNs try to reduce the gap between classical DNN operations and a more biologically-plausible computing, trying to mimic how informations are actually represented and transferred in the human brain. Although the great advantages that Spiking Neural Networks can offer, learning in SNNs is still an open issue [51]. In this context, the conventional backpropagation mechanism used to train DNNs for supervised learning cannot be applied due to the non-differentiabile nature of the spiking function [28]. In the following, different strategies to solve or bypass the problem are presented.

4.3.1 Local Learning Rules for Unsupervised Learning

Local learning rules are derived from the study on animal and human brain and can be applied to train a Spiking Neural Network. Moreover, local learning rules:

- due to the time-dependent nature of spikes are capable to catch temporal-related features which can be present in the input of the network.
- are particularly efficient to be deployed on dedicated hardware architectures so that an on-line training can be applied.

Spiking Time Dependent Plasticity (STDP) local learning rule is popularly used to train SNNs. The goal of such a rule is to strengthen the synaptic

weight of two neurons whose spiking activity happens in a highly-correlated causal dependency order, and to weaken it otherwise [7]. The formal equation that states this unsupervised learning rule can be expressed as:

$$\Delta w_{STDP} = \eta \cdot \left(e^{-\frac{t_{post} - t_{pre}}{\tau}} - STDP_{offset} \right) \cdot (w_{max} - w)^{\mu},$$

where Δw_{STDP} represents the modification of the synaptic weight value, η is the learning rate, t_{pre} is the time instant of a pre-synaptic spike and t_{post} is the time instant of a post-synaptic spike, τ is the time constant for the membrane potential decay, w_{max} is the maximum value that the weight can assume, and w is the current weight [7]. If the post-synaptic neuron is spiking within a time defined by $STDP_{offset}$, then the synaptic weight associated is increased or weakened otherwise [7].
Learning through unsupervised learning rules is found to be effective just for shallow networks. That is mostly due to the close locality of such learning rules, where each layer is changed with coordination to the output of the previous layer [6]. How these close locality can be extended to other layers in the network, so that the whole network can be effectively influenced by the learning procedure is still an open issue. Local on-chip training can also take advantage of the hardware accelerators *Intel's Loihi* [9] neuromorphic processor or *SpiNNaker* [11], which are both supporting STDP learning rule.

4.3.2 Spikes' Approximate Derivative Method for Supervised Learning

If supervised learning wants to be applied in Spiking Neural Networks, the solution is to take advantage of an approximate derivative method for the spike trains to apply backpropagation. This solution has been extensively studied in many works [39][28][4][30], which provide different approximations for the derivative of the spike trains. The advantage of this type of solution is that the temporality of spikes is maintained and the network can learn features also accordingly to this information. On the other hand, this approach is not aiming to be biologically plausible but, instead, to reproduce and extend the consolidated results of the state-of-the-art of DNNs.

4.3.3 Off-line trained DNNs to SNNs conversion

Converting off-line trained DNNs to SNNs is a good choice to leverage the state-of-the-art goals reached in the field of machine learning and image recognition tasks. In fact, conventional DNNs shown excellent results accuracy in classification tasks on various datasets (MNIST, CIFAR-10, CIFAR-100, ImageNet etc...). This approach is aimed at reproducing the mapping of input/output signals of a DNN into a SNN.

Most of the conversion approaches apply rate coding of continuous activation functions and weights rescaling according to SNNs neurons parameters such as leak rates and refractory times, which must be specified as hyper-parameters [42]. However, some problems arises if a conversion mechanism is adopted.
The continuous values for the activations in DNNs can be either positive or negative, nevertheless only positive values can be translated into spiking rates. A proposed approach is to use employ separated spiking neurons for positive and negative activations, starting from the fact that in biology spikes associated to positive or negative values travel on different paths [43]. With the introduction of the $ReLU$ function the importance of the problem is reduced since the activations can assume just positive or zero values.
Another limitation is that the *max pooling* operation is non-linear and cannot be readily implemented; in fact, just at the end of the simulation time, the spiking rate would be known, i.e. the max pooling cannot be implemented spike by spike [42]. Max pooling operations are usually substituted by *average pooling* operations, which can be easily implemented. If *latency coding* is implemented, the max pooling operation can be easily implemented: the max between the input neuron's values is computed as the first spike to arrive to the post-synaptic neuron. However, this approach is not suited for the conventionally used *rate coding* conversion.

SNNtoolbox In the analysis carried on, the method described in this section is exploited: The networks are off-line trained as DNNs and then converted into SNNs, taking advantage of the useful SNNtoolbox [49]. SNNtoolbox's workflow is shown in Figure 4.4: a neural network architecture is described with some neural network library (e.g. *Caffe*, *Lasagne*, *Keras*), SNNtoolbox parses the network, extracting useful information and creating an equivalent Keras model. The parsed model serves as a common abstractions stage from the input and is internally used by the toolbox to perform the actual conversion into a SNN [49].

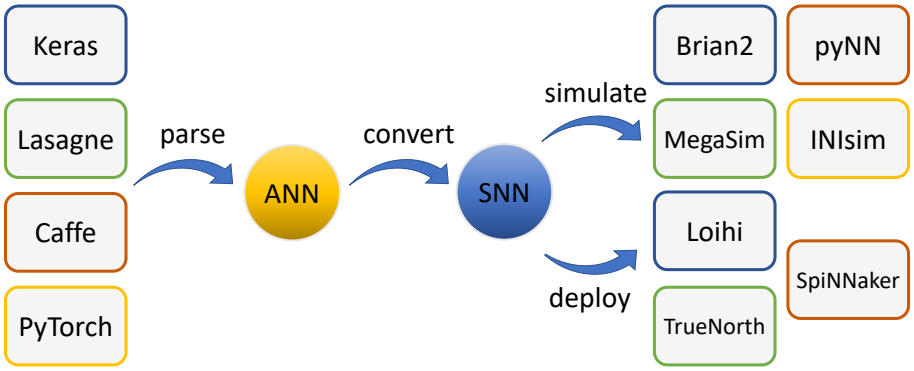

Figure 4.4: Sketch of the work flow of SNNtoolbox [49].

After the model is converted, the resulting SNN can be **exported** for simulation in a spiking simulator or **deployed** on dedicated spiking neuron chips (e.g *TrueNorth*, *SpinNNaker*, *Intel Loihi* etc...). Provided simulators are *INI*, *MegaSim*, *Brian2*, *Nest*. SNNtoolbox documentation shows the different capabilities of the simulators with respect to the features (layers) that can be implemented on them. SNNtoolbox gives the possibility to impose different specifications for the various steps from conversion to simulation.
In this work, networks are described taking advantage of the *Keras* library and simulated exploiting the built-in *INIsim* simulator which implements highly parallel computation reducing the simulation time; *rate coding* is chosen as coding strategy, jointly with *Leaky Integrate and Fire* neuron model. *INIsim* is, as well, the only simulator which implements constant input currents to convert the input image pixels into spikes, so that the Poisson input generator can be turned off. Moreover, it is the only simulator which implements *softmax* activation function (in the other simulators, softmax is replaced by *ReLU*) and *max pooling*.

Chapter 5

Bit-flip Resilience Analysis of SNNs

In this chapter, the effect of errors in the parameters of some networks operating on MNIST, CIFAR10 and GTSRB datasets, is analysed. As anticipated in section 3.2, the experiments carried on are simulation-based fault-injections in the memory elements (supposed SRAMs) storing the parameters of some neural networks. These experiments can be thought as the simulation of an actual physical laser beam fault-injection testing. In the following experiments, the unique interest metric to quantify the resilience, i.e. the response to the errors, is the high-level classification accuracy reached on the overall test set. Therefore, the advantage with respect to a physical hardware Laser Beam Fault Injection is clear as the testing time and cost are drastically reduced.
The faults injected are modeled as permanent SEUs (see section 3.1), exploited as bit-flips in the memory units storing the parameters of the network. Different networks have been chosen for the MNIST, CIFAR10 and GTSRB datasets. After training, weights and biases are quantized to 8 bits, with almost null reduction of accuracy.
Specifically, in section 5.1, a layer-wise analysis and then a cross-layer analysis of a statistical bit-flip insertion are performed. On the contrary, in section 5.2, a gradient search algorithm is applied, both on a layer-wise and cross-layer basis. This algorithm is capable to find out which are the most critical weights, i.e. weights that, subject to a variation, contribute to a drastic reduction of the accuracy with respect to the others.

Datasets The datasets analysed in my thesis work are:

- **MNIST** (Mixed National Institute of Standards and Technology): dataset for image recognition of 28×28 pixels grayscale images organized in 50 000 training images and 10 000 test images of handwritten digits from 0 to 9 belonging to 10 different classes [27].

- **CIFAR10** (Canadian Institute for Advanced Research): dataset for image recognition of 32×32 pixels RGB images organized in 50 000 training images and 10 000 test images from natural world belonging to 10 different classes [24]. A sample image for each class is reported in Figure 5.2.
- **GTRSB** (German Traffic Sign Recognition Benchmark): Security issues in automotive field are nowadays of great interest, that is why it has been decided to analyse on the *German Traffic Sign Recognition Benchmark* (GTSRB). The GTSRB is composed of more than 50 000 images of different size belonging to 43 different classes. A sample image for each class is reported in Figure 5.3. The training set is composed by one directory per class. Training images are grouped by tracks, each one containing 30 images. There is not an even distribution of tracks per class, as it is shown in Figure 5.1. First of all, since each image is of a different size, they should be resized to match the input layer of the CNN. Since the trigger generation and imperceptibility depends also on the input image size, different resizing of the images have been tried to understand this aspect.

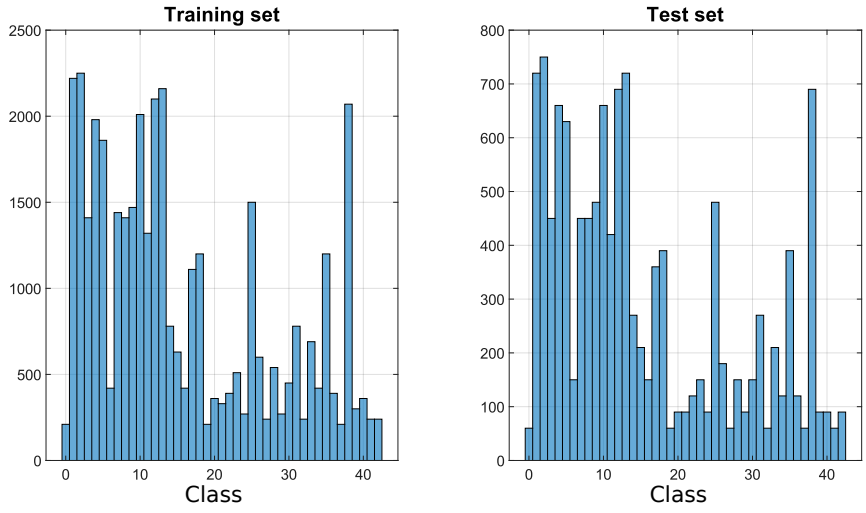

Figure 5.1: GTSRB histogram for both training set (left) and test set (right).

Networks All networks have been trained for 150 epochs to reach the maximum accuracy. The analysed networks which are classifying images belonging to the MNIST dataset are

- a simple 4-layers Multilayer Perceptron (see section 2.3.2), whose structure is reported in Table 5.1, whose baseline accuracy is 95.54%,

Figure 5.2: Sample images from the *CIFAR10* dataset with labels representing the corresponding class.

Figure 5.3: Sample images from the *German Traffic Sign Recognition Benchmark*.

- a CNN with a LeNet-like structure, shown in Table 5.2 (the number of feature maps is increased both for first and second convolutional layer), whose baseline accuracy is 99.05%.

Convolutional Neural Networks are used to classify both the CIFAR10 (structure reported in 5.3) and the GTSRB (structure reported in 5.4) datasets reaching a baseline accuracy of 79.65% and 98.39% respectively.

Table 5.1: Structure of the MLP classifying images belonging to MNIST dataset with baseline accuracy of 95.54%.

Layer	Output shape	Activation
Input	784	-
Dense	1200	ReLU
Dense	1200	ReLU
Dense	10	ReLU

Table 5.2: Structure of the CNN classifying images belonging to MNIST dataset with baseline accuracy of 99.05%.

Layer	Output shape	Features	Kernel	Stride	Activation
Input	(28, 28, 1)	-	-	-	-
Conv2D	(28, 28, 32)	32	(5,5)	(1,1)	ReLU
MaxPool	(14, 14, 32)	-	(2,2)	(2,2)	-
Conv2D	(10, 10, 48)	48	(5,5)	(1,1)	ReLU
MaxPool	(5, 5, 48)	-	(2,2)	(2,2)	-
Dense	256	-	-	-	ReLU
Dense	84	-	-	-	ReLU
Dense	10	-	-	-	Softmax

Table 5.3: Structure of the CNN classifying images belonging to CIFAR10 dataset with baseline accuracy of 79.65%.

Layer	Output shape	Features	Kernel	Stride	Activation
Input	(32, 32, 3)	-	-	-	-
Conv2D	(32, 32, 32)	32	(3,3)	(1,1)	ReLU
Conv2D	(30, 30, 32)	32	(3,3)	(1,1)	ReLU
MaxPool	(15, 15, 32)	-	-	(2,2)	-
Dropout 0.25	(15, 15, 32)	-	-	-	-
Conv2D	(15, 15, 64)	64	(3,3)	(1,1)	ReLU
Conv2D	(13, 13, 64)	64	(3,3)	(1,1)	ReLU
MaxPool	(6, 6, 64)	-	-	(2,2)	-
Dropout 0.25	(6, 6, 64)	-	-	-	-
Dense	512	-	-	-	ReLU
Dropout 0.25	512	-	-	-	-
Dense	10	-	-	-	Softmax

Table 5.4: Structure of the CNN classifying images belonging to GTSRB dataset with baseline accuracy of 98.39%.

Layer	Output shape	Features	Kernel	Stride	Activation
Input	(32,32,3)	-	-	-	-
Conv2D	(30,30,32)	32	(3,3)	(1,1)	ReLU
Conv2D	(28,28,128)	128	(3,3)	(1,1)	ReLU
MaxPool	(14,14,128)	-	-	(2,2)	-
Conv2D	(12,12,128)	128	(3,3)	(1,1)	ReLU
MaxPool	(6,6,128)	-	-	(2,2)	-
Dropout 0.25	(6,6,128)	-	-	-	-
Conv2D	(4,4,128)	128	(3,3)	(1,1)	ReLU
MaxPool	(2,2,128)	-	-	(2,2)	-
Dropout 0.5	(2,2,128)	-	-	-	-
Dense	128	-	-	-	ReLU
Dropout 0.5	128	-	-	-	-
Dense	43	-	-	-	Softmax

5.1 Statistical Analysis of Random Bit-Flips

In this section the resilience of SNNs to random bit-flips in internal parameters is analysed. The bit-flips are simulated as a natural soft-errors source is present, thus supposing there is a source of alpha particles or neutrons as described in section 3.1. The amount of bit-flips is therefore taken as a percentage of the complete amount of bits in the network. Then, also a *Statistical Fault Injection* simulation is carried on; in this case the interest is in the amount of bit-flips, taken as an absolute number. The simulations are carried on both on a cross-layer manner and a layer-wise manner.

5.1.1 Cross-layer Analysis

The cross-layer (global) analysis of the resilience of the neural networks is studied on a system level, by monitoring the variation of classification accuracy, while the amount of soft-errors is increasing. Considering that the area of the memory elements storing the parameters of the network is constant, then an increasing soft-error rate is related to an increasing number of faults in time, which can be caused, for example, by the increasing flux of a radiating source. Both SEU and MBU are considered in a ratio 1/10. At each step, the clean parameters are set, then, at the first step, an amount x_0 of bits is flipped. At the next step, weights are reset and a higher amount x_1 of bits is flipped, and so on until an amount x_N of bits is flipped. This sequence is repeated 15 times per network and then the results are averaged. What is important is to catch the point on the x axis where the curves start to bend, so when the accuracy is starting to decrease.

Multi Layer Perceptron The cross-layer resilience of the MLP to natural soft-errors is found to be affected also for smaller values of bit-flip probability, with respect to CNNs. That is because the amount of parameters is very high, and therefore also a small percentage of soft-error is sufficient to cause a larger amount of bit-flips, hence a bigger classification accuracy drop. Moreover, this is perfectly physically plausible because a network with more parameters has memories with a larger area and the soft-error rate (hence bit-flips) is proportional to the memory area (see section 3.1).

Convolutional Neural Networks CNNs seems to be less affected by natural soft-errors. That is because, as explained, the amount of parameters, for each CNN is less with respect to the MLP. Plotting the result for each different simulation for the same line, it can be highlighted an increasing variance for intermediate values of bit-flip probability meaning that the effect of the bit-flip is not only depending on the amount of errors but also on which specific weight is subject to the error. With low amount of bit-flips, the variance appears to be low since it is unlikely that the network accuracy is affected in a severe way, wherever the error is applied. The same happens when the amount of soft-errors is very high because the accuracy is probably compromised for the effect of so many errors, regardless their positions. For intermediate amount of errors, the position of them seems to be crucial. This effect is clearly shown in Figure 5.5 just for the MLP case, for clarity. In the simulation pointed by green arrow, the random bit-flips result to be placed in more sensitive positions with respect to the other experiments, yielding a lower accuracy.

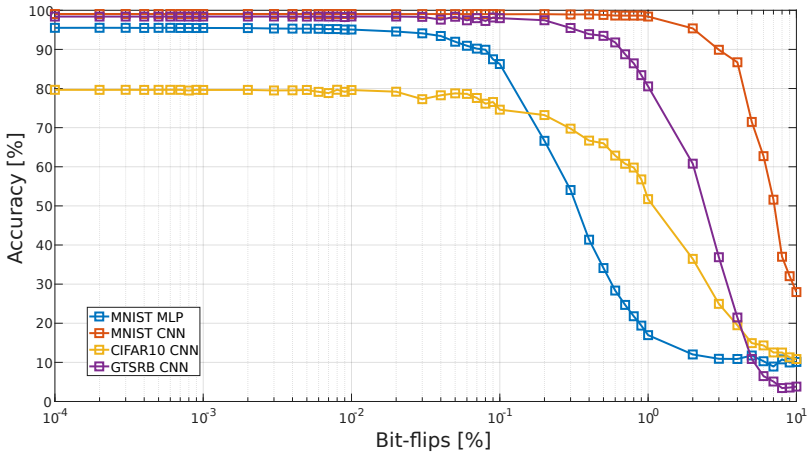

Figure 5.4: Cross-layer averaged accuracy with respect to the bit-flip probability for MNIST MLP (blue), MNIST CNN (red), CIFAR10 CNN (yellow) and GTSRB CNN (purple).

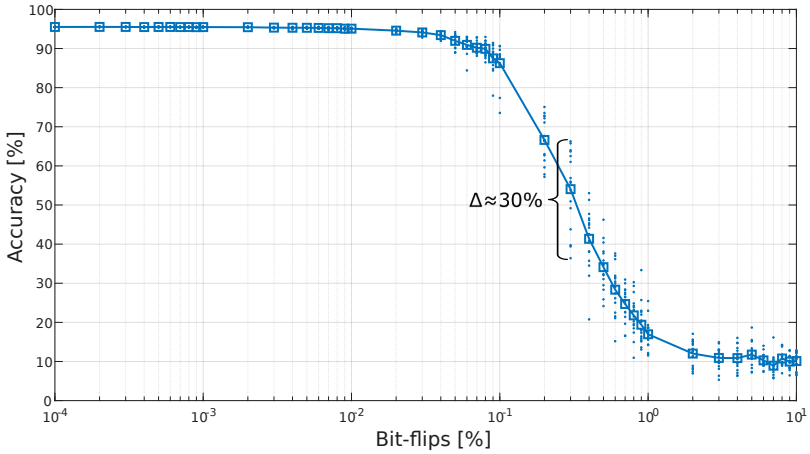

Figure 5.5: Cross-layer accuracy results for the different experiments (dots) and averaged accuracy (line and triangles) with respect to the bit-flip probability for the MLP.

Despite, realistic values for natual soft-error cannot be directly derived, since the SER is measured over time, it is understandable how realistic values of natural soft-errors are very low and can be considered to be in the far left part of Figure 5.4, so that the accuracy reduction, in average is not significant.

Table 5.5: Total amount of parameters and bits for each network.

	Parameters	**Bits**
MNIST MLP	2 953 424	19 161 680
CIFAR10 CNN	1 250 858	10 006 864
GTSRB CNN	494 267	3 966 424
MNIST CNN	369 174	2 953 392

These results show that there is no difference in resilience between different architectures of neural networks or datasets. The only possible conclusion is that networks with a larger amount of parameters are subject to a higher amount of errors and the related classification accuracy is highly affected, compared to smaller or shallower networks. The number of parameters and bits for the nets are shown in Table 5.5.
The soft-error range for the simulation is stressed for most of the plot to highlight variations of the response. Another analysis is conducted by considering the amount of soft-errors injected as an absolute number, not related to the overall amount of parameters of the network. In fact, in the previous analysis the lines started bending in dependence of the amount of the parameters of the network: first the MLP, then CIFAR10 CNN, then GTSRB CNN and last the MNIST CNN, following the sequence in Table 5.5. Here

the bending points are closer one to each other, as shown in Figure 5.6; it can also be seen that for just a high number of bit-flips i.e. between $\approx 10^4$ and $\approx 10^5$. The high resilience of neural networks is therefore proven.

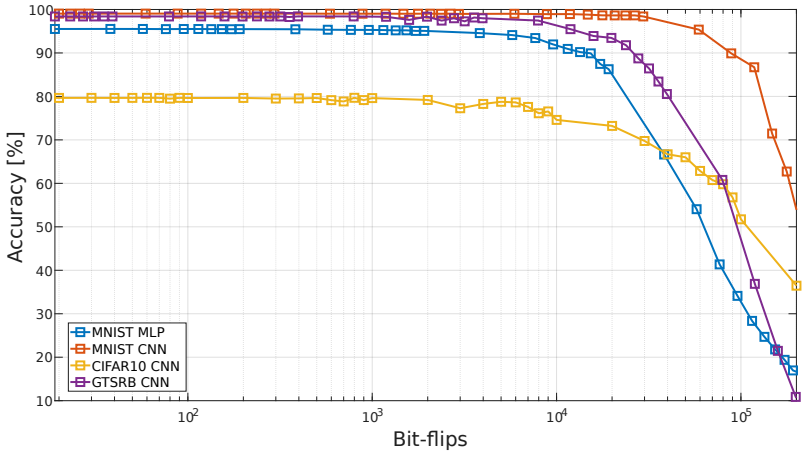

Figure 5.6: Cross-layer accuracy results for the different experiments (dots) and averaged accuracy (line and triangles) with respect to the bit-flip probability for the MLP.

5.1.2 Layer-wise Analysis

A layer-wise analysis of natural soft-error has been carried on. However, such low values of bit-flip probability (soft-error rate) carry on two main drawbacks:

1. For classification tasks of the datasets considered in this work, the neural networks used, despite having from hundreds of thousands up to two million parameters, are considered, anyway, small compared to much bigger nets like AlexNet [25] or VGG-16 [50]. If realistic values for soft-error are applied, placing bit-flips on the network's weights is possible just on a global (cross-layer) basis, therefore a layer-wise analysis could not be possible due to the small number of weights of each layer.
2. It could not be clearly pointed out the different resilience of each layer. In fact, characterizing the difference in resilience when random bit-flips are applied can steer the (targeted) fault-injection analysis.

In order to overcome these difficulties, the bit-flip probability range is extended to larger values. However, it can be clearly seen from this research that differences arise when bit-flips are focused on the weights of a particular layer. The results are averaged over 15 different experiments.

Multilayer Perceptron Looking at Figure 5.7 on top, it can be seen that the MLP is almost insensitive with respect to a random bit-flip in the biases, regardless of the layer. On the other hand, the sensitivity is prominent when bit-flips are applied to the weights. The accuracy curves for each layer show almost the same trend, however, the network appears to be slightly more robust with respect to bit-flips applied in the weights belonging to the last layer.

Convolutional Neural Network In CNNs, the effect of random bit-flips on the overall classification accuracy is depending also on the layer type. What is clear from Figure 5.7 is that first convolutional layer is critical for each network. In fact the first convolutional layer is, generally, the one which has the lower amount of parameters. Moreover, its position is critical because the associated parameters are related to the first feature maps. By applying errors on them, there is a sort of chain reaction. For CIFAR10 CNN and GTSRB CNN, also the second convolutional layer is critical. This is due to the different sequence of layers of these last tow networks, where two convolutional layers are placed one after the others in the first two layers; on the other hand on MNIST CNN, the first convolutional layer is followed by a pooling layer which is somehow mitigating the effect of the errors on the second layer. Dense layers are generally more resilient, except for the last one, due to its position, strongly related to the final classification task.

In summary, the previous analysis points out the high resilience of neural networks to random errors injected in internal parameters. The performances, considered in terms of accuracy, are degraded just for huge amounts of faults injected. However, these networks, as demonstrated in the following section, are resilient only for probabilistic attacks, while showing very different behaviour in case of well-targeted errors that can be applied by an adversary.

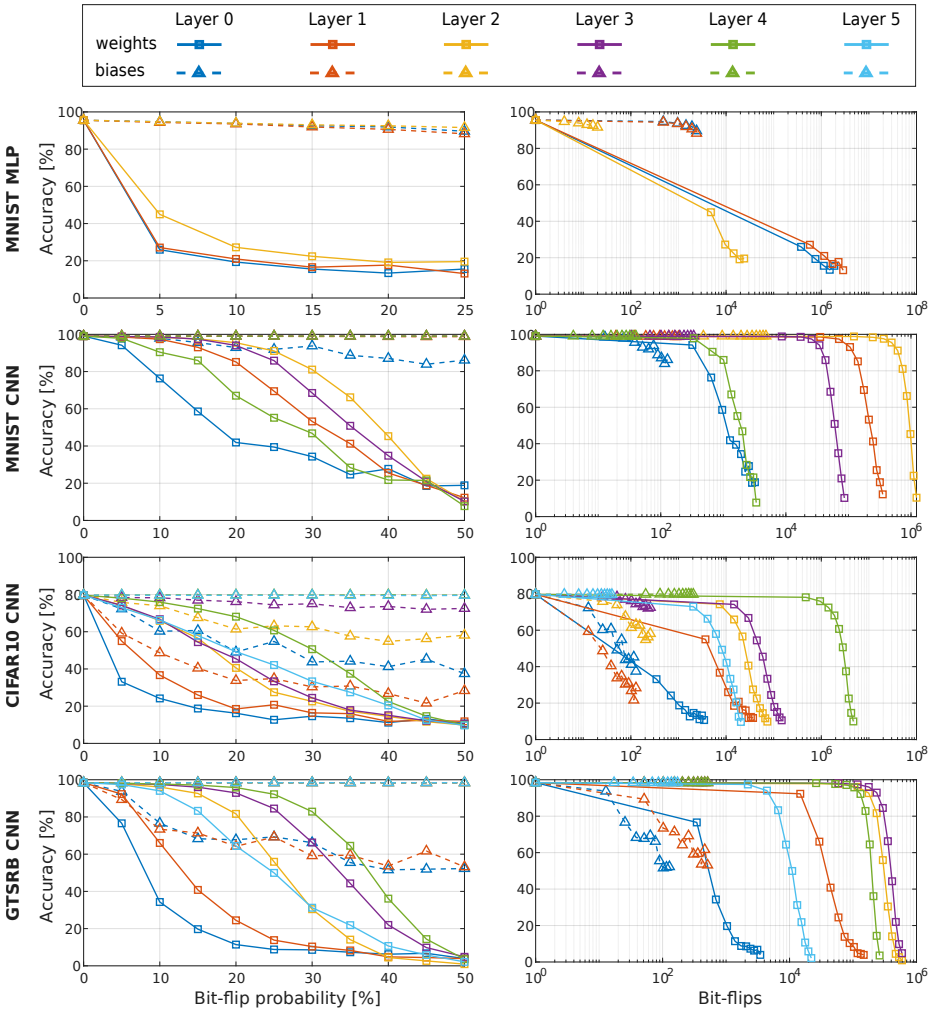

Figure 5.7: Layer-wise accuracy with respect to the bit-flip probability for (first row) MNIST MLP, (second row) MNIST CNN, (third row) CIFAR10 CNN and (fourth row) GTSRB CNN.

5.2 Bit-Flip with Gradient Search Algorithm

As explained in section 3.2, the fault-injection technique is a powerful experimental testing technique for ICs. If the *efficiency* of a fault-injection attack, for neural networks, is computed as the reduction of accuracy with respect to the amount of faults injected, it is clear that a random placing of faults is not the best solution. Starting from the assumption, gained by the discussion of the results from the previous section, that in a NN there are some connections which are more vulnerable than others, it could be possible, in principle, to identify them by iteratively applying the fault on each parameter of the network while recording the accuracy variation. However, due to the high amount of parameters in a NN, this approach would be highly time-consuming and definitely not feasible.
In this section, a method to identify the most vulnerable weights, without the need for an overwhelming amount of simulations, is described. Taking inspiration from [44] and [34], the gradients of the loss function with respect to the parameters of the network are computed in a similar way to what is done during the learning phase (see section 2.3.5). The highest gradient in absolute value is considered and the related parameter is taken as the target parameter to apply the fault. Differently from the learning phase, the value of the weights is modified in the direction of the positive value of the gradient, since the aim is to identify the vulnerabilities of the net, therefore to reduce the accuracy as much as possible. To reach the minimum accuracy by varying the target parameter, each bit is flipped iteratively starting from the original value and the accuracy is registered. The bit-flip leading to the highest reduction is selected as the target and the value of the target parameter is updated for the next iteration. The target parameter is then masked, so that it is not considered at the next iteration.

5.2.1 Cross-layer Analysis

The results show that the accuracy is highly reduced for very low amount of bit-flips for the MNIST MLP, blue line in Figure 5.8, and for the MNIST CNN, red line in Figure 5.8, considering a global analysis of the parameters. Note, only $\simeq 30$ bit-flips are sufficient to completely crush the accuracy of the two considered networks.
Similar experiments have been obtained also for the CIFAR10 dataset. As shown by the yellow line in Figure 5.8, the accuracy drop is far more emphatic. In fact, the accuracy reaches a plateau around 10% for just $\simeq 4$ bit-flips, which is a more critical result than the one obtained with the MNIST MLP and MNIST CNN. The GTSRB CNN accuracy is also highly reduced and reaches the minimum plateau for $\simeq 15$ bit-flips (purple line in Figure 5.8).
The plateau is given by the fact that the algorithm applied, after some

perturbations, is leading the network to classify each image as belonging to a given class. Since this attack is not targeted, this class is not directly chosen to be a specific one. For example the GTSRB CNN accuracy plateau is $(1/43) \cdot 100 \simeq 2.3\%$, since there are 43 classes. With the proposed fault-injection attack, when reaching the plateau, the neural network is classifying each traffic sign as a general "attention" sign; a targeted attack which aims to direct the misclassification to a specific choice (for example by aiming a high speed limit, leading the car to accelerate) can increase the degree of danger of the attack. However, such targeted attack scenario is left as future work.

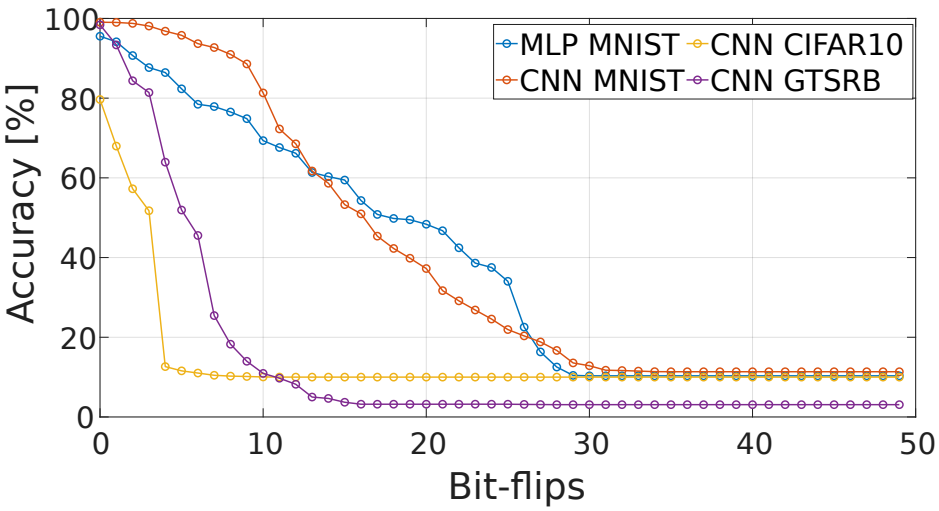

Figure 5.8: Accuracy vs number of bit-flips for MNIST MLP (blue), MNIST CNN (red), CIFAR10 CNN (yellow) and GTSRB CNN (purple).

5.2.2 Layer-wise Analysis

In Figure 5.9 on top left and Figure 5.9 on top right, the layer-wise classification accuracy values for the MNIST MLP and MNIST CNN network are respectively reported. If there are some lines which are not composed by 50 points it is due to the fact that they are related to errors in biases; since it has been chosen to put one bit-flip per parameter, it has been not always possible to insert the wanted amount of errors. For example, biases in the last layer of MNIST net are only 10; same thing can happen for a low amount of feature maps (Figure 5.9, top left, blue triangle line). The layer-wise analysis of the bit-flip attack, however, is again showing, that there are some layers that are more vulnerable to bit-flip attacks with respect to the others.

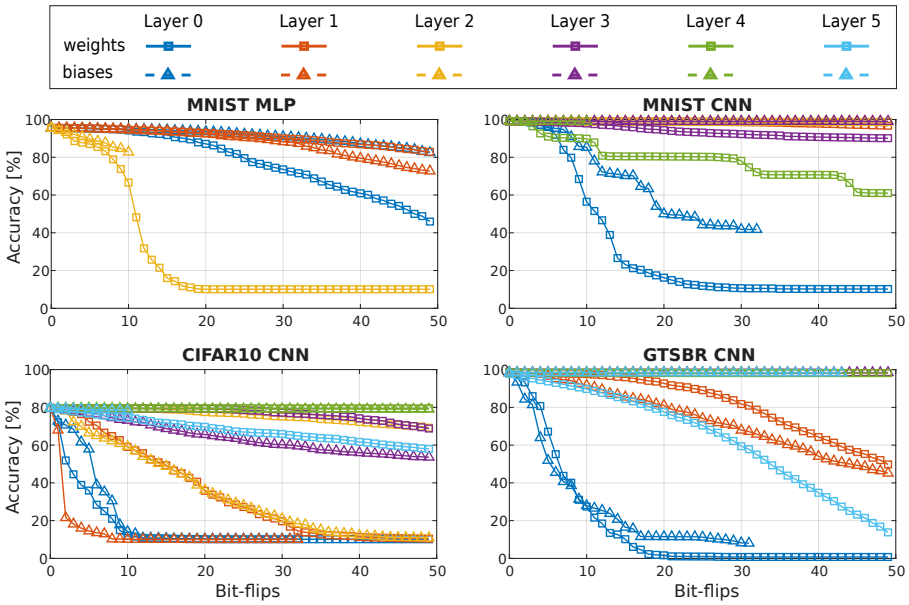

Figure 5.9: Layer-wise accuracy with respect to the amount of bitflips for: from top left to bottom right MNIST MLP, MNIST CNN, CIFAR10 CNN and GTSRB CNN.

Multilayer Perceptron The last layer in the MLP seems to be the most vulnerable, maybe because its strong connection with the classification layer. Again, a high resilience is proven by the hidden layer. Fault-injections in the first layer lead to results which are halfway between the two. The lower resilience of the first and the last layer could be due to:

- the position of these layers, providing an higher correlation with input image and with the classification stage respectively.
- the lower amount of parameters with respect to the hidden layer.

The analysis of the curve for the output layer is very interesting because is showing that, focusing the bit-flip injection in the weights belonging to the last layer, the amount of bit-flips to reach the plateau is $\simeq 20$. On the contrary, a cross-layer attack on the same network, was reaching the same plateau for $\simeq 30$ bit-flips (Figure 5.8). This result can be counter-intuitive: however, it must be considered that the gradient descent is not always capable of finding the local minimum.
On the other hand, the last layer was found to be the most robust one when subject to random bit-flips (green line in Figure 5.7).

Convolutional Neural Networks The analysis of CNNs show that weights and biases from the first convolutional layer are again the most vulnerable to the attack. This is due to the fact that weights in the first layer are the input image feature maps. When subject to surgical bit-flips, the first

layer is not capable of recognizing features in the input image, causing an avalanche effect degrading the final accuracy. Even the last (dense) layer shows lower resilience, which is probably due, as in the case of the MLP, to its deep correlation with the final classification stage.

From this chapter, it is clear that the resilience of the networks is depending on the different layers' type (dense or convolutional), the amount of parameters associated and on the position in the network as well. What is in general found is:

1. The resilience of a layer of MLP when subject to random bit-flips, generally depends on the amount of parameters in that layer with direct proportionality; this is valid for surgical bit-flips as well.
2. The critical layer for CNN is the first convolutional layer since (1) it is always the layer with the lowest amount of parameters (2) it is directly related to the first features in the input image. The last (dense) layer of a CNN is critical as well due to the generally low amount of parameters related to it and due to its relationship with the final classification.

An important result is that the layer-wise fault-injection can be, in some cases, more more effective than the cross-layer fault-injection. For example in the case of the MLP is found that the maximum drop of accuracy is reached for $\simeq 20$ bit-flips when targeting just the last layer, whereas is reached for $\simeq 30$ bit-flips with layer-wise injection. This is due to the nature of the algorithm applied that, analyzing just the gradients of the loss function, is not always capable of reaching the minimum.

Chapter 6

Trojan Attack Methodology

In this chapter, a hardware backdoor trojan attack methodology is provided, based on the results of the previous chapter. The idea is to craft and apply a particular input pattern to the image that must be classified so that a particular target neuron is overstimulated. This irregular behaviour is detected and leveraged to be used as a trojan trigger. The payload circuitry is modeled as few bit-flipping circuits in the memory elements storing the parameters of the attacked NN. The attack can be directed against a DNN or a SNN, just modifying some hardware.

6.1 Threat Model

The attack phase is supposed to lay within the *supply chain*, where a malicious can insert hardware trojans. In fact, due to the modern ICs fabrication chain, lots of different parties, each one focused on its own expertise, get involved in the project of a hardware product to shorten the production chain. This makes extremely difficult to control the goodness of the product itself during the whole supply chain and opens door for hardware trojan attacks to malicious companies or to ill-intentioned staff [8]. Moreover, the attack is a *white-box* setting: the attacker has a complete knowledge of the system architecture, internal parameters and low level implementation informations; however it is not aware of the training set and training hyperparameters.

6.2 Hardware Trojan Design

The hardware trojan is designed to perform bit-flips in the network parameters to undermine its integrity and degrade its accuracy. The malicious behaviour is triggered by the input through a specifically crafted input pattern. Taking advantage of the analysis carried on in section 5.2, stealthy hardware trojans are placed. Ideally, each trojan consists of a 2-way multiplexer with one input which is the original bit, whereas the other input is the

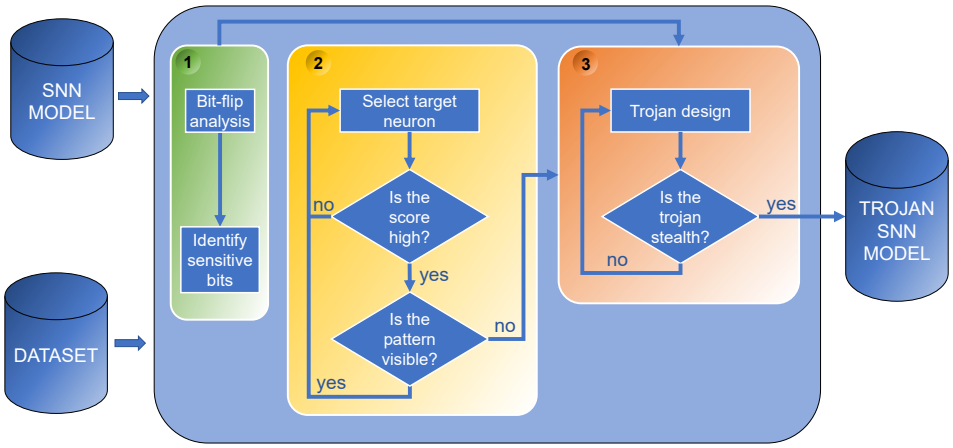

Figure 6.1: Scheme of the trojan attack generation flow.

complemented bit obtained through an inverter. The multiplexer's selection signal is a signal which is at a high logic value only when a trigger is added to the input image. In this way, the network will behave correctly, providing the baseline accuracy for the original dataset, for example during test phase, but, when a trigger is inserted in the image, the hardware trojans will be activated and therefore the reliability will be highly degraded. The setting is explained in Figure 6.2, in which the thickest orange arrows represent the synapses with bit-flip applied and the grey neuron corresponds to the target neuron.

To produce the selection signal of the multiplexers, the output of a selected neuron is compared against a threshold, chosen accordingly to the results of the simulation. Ideally, it is wanted that the output of the neuron is exceeding the threshold just when the trigger is added to the image.

To transfer the methodology from the DNN to the SNN rate-based implementation, a counter, which accumulates the number of spikes, is needed at the input of the comparator. Moreover, the threshold must be transferred from its digital representation to spike representation, considered in this work as spike rate. The counter is cleared at the end of the simulation time.

Some steps are followed to craft the attack. This is, however, just an idea of how a trojan trigger can be implemented. In section 7.3, a deeper analysis of how the trigger can be physically deployed is taken into account.

As a first step, the selection of a particular neuron to satisfy the wanted behaviour has to be performed. Even though a methodology is provided in the following discussion, results strongly depend on the datasets and on the specific networks analysed for the classification task; some intuitive deductions can shorten the process and lead to good results.

6.3 Trigger Pattern Design

Since there can be found a direct relationship between the continuous output value of a neuron and the corresponding spike rate, the knowledge obtained through the analysis of a DNN can be transferred to the SNN implementation. Moreover, a good correlation between analog output value and spike rate is a necessary condition for a coherent simulation in SNNtoolbox. The goal is to embed the trigger inside one neuron of the network, namely the *target neuron*. In other words, what is wanted is that such target neuron is activated by a carefully designed pattern in the input image.
The goals that the pattern must satisfy are the following

- *Effectiveness*: the target neuron must be effectively overstimulated by the trigger pattern when this is added to the input image.
- *Stealthiness*: the target neuron must not be unintentionally overexcited by the original dataset so that the trojan is not accidentally activated.
- *Imperceptibility*: the trigger pattern must be as much imperceptible as possible to the human eye.

To simultaneously satisfy all of these conditions has been found to be generally non-trivial; however, some promising results are obtained. The choice of the target neuron and a methodology to generate the input pattern are explained in the following.

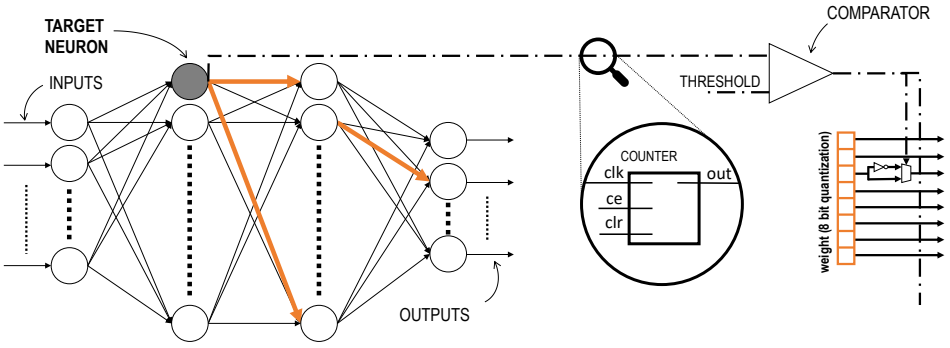

Figure 6.2: Scheme of the trojan attack for the MLP network with the counter added present only in SNN implementation.

6.3.1 Choosing the target layer

Each neuron in a neural network is related in some way to the input image's pixels; the relationship between the activation of a particular neuron and the input image can be expressed in terms of *gradients*. In the following, the term *gradient* will be referred to as the *relationship between activation*

of a neuron and the input image's pixels. The pattern must be placed in areas of the image which are covered by the gradients of the selected neuron, otherwise, the activation of the neuron is not affected by the pattern itself. Hence, the selection of the *target pattern*, strongly depends on the *target layer*. In the case of MLPs, the gradients of each neuron cover the entire image: if a smaller pattern is wanted, a mask, can be crafted to select a particular area of the image.
In the case of a CNN, the choice of the layer is directly connected to the choice of the size of the pattern. This is due to the fact that neurons belonging to deeper convolutional layers are related to a larger area of the input image. For example, by looking at Figure 6.3, the gradients of a neuron belonging to the first and second convolutional layers are reported. The deeper the layer, the larger the area of the image that will account for the pattern. At the first convolutional level, the shape, position and value of the gradients are quite clear and corresponds to the *feature map* of the neurons. That is why in the following, the first layer will be often considered as the target layer.

6.3.2 Choosing the target neuron

The target neuron is chosen as the one with the maximum between the sum of absolute values of weights, connecting that neuron to the previous layer. This is modeled by the following equation:

$$\underset{t}{argmax}\left(\sum_{i=1}^{N} ABS(W_{layer_{i,t}})\right).$$

This is a wise choice, inspired by [35] since in this way, a powerful variation of the output when the input is changed, is granted. However some considertions about the dataset can be carried on. For example, considering the MNIST dataset, it can be found that, generally the value of the pixels at the border of the images is low; placing the pattern in this area will provide a good separation of activation between original and modified dataset.

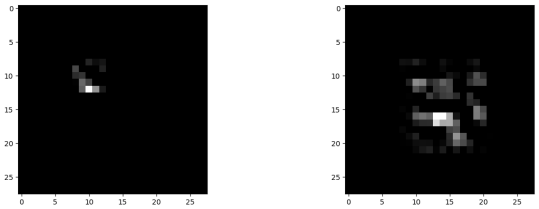

Figure 6.3: Gradient representation of a random neuron from (left) the first and (right) the second convolutional layer of a CNN.

6.3.3 Choosing the triggering mask

A *random initial image*, of the same size as the input layer of the network, is generated and the network is inferred with that image, leading to a value $initial_{OUTPUT_k}$ at the output of the target neuron. The value $target_{OUTPUT_k}$ is chosen to be way much higher than $initial_{OUTPUT_k}$. A cost function is then defined as follows:

$$cost = \frac{\sum_{i=1}^{N} \delta_i^2}{N},$$

Where $\delta_i = target_{OUTPUT_i} - initial_{OUTPUT_i}$ and i is the index of the i-th neuron of the N neurons in the target layer. Being k the index of the target neuron, we rewrite the expression as:

$$cost = \frac{\delta_1^2 + \delta_2^2 + ... + \delta_k^2 + ... + \delta_N^2}{N}$$

For each δ_i it is imposed that $target_{OUTPUT_i} = initial_{OUTPUT_i}$ except for δ_k, where $target_{OUTPUT_k} \neq initial_{OUTPUT_k}$. The derivative of the cost function is computed with respect to the pixels of the random input image, to find out which part of the input image influences the target neuron, that is computing the *gradients*. Based on this, a mask M is created and a *random initial pattern* is generated by the dot product between the mask and the *random initial image*. The mask can also be chosen in a different way, but it must have some overlap with the gradient matrix, otherwise the following algorithm, is not effective.

6.3.4 Generating the trigger

The trigger generation algorithm (see Algorithm 1) is inspired by the work of Liu and al. on trojan attacks on neural networks [35]. However, in the work of Liu and al. the trojan is exploited in a different way, by retraining the network with additional images created by the attacker. Here the loop is exploited to trigger the over-excitation of the target neuron. In the first rows of the code, some initialization parameters are set: val_{min} and val_{max} are useful parameters to manage the imperceptibility characteristics of the pattern; val_{min} and val_{max} boundaries are (0,1). The imperceptibility is not measured in mathematical terms but just considered through a visual inspection and checked to be quite low otherwise the result is discarded. Crafting an imperceptible backdoor trojan attack is marginal with respect to the scope of my work thesis and can be future work.
The gradients Δ are first calculated and then limited by a mask that can be suited for the gradients (in that case, line 4 of Algorithm 1 can be skipped) or can be decided in another way; line 6 is added to limit the maximum and minimum values for the pixels in the trigger. The loop proceeds until the

cost is reduced under a particular threshold *th* or until a maximum number of iterations *epochs* is reached.

Algorithm 1 Trigger generation loop

1: **INIT**$(val_{min}, val_{max}, lr, epc, epochs, th, cost)$
2: **while** $cost < th$ and $epc < epochs$ **do**
3: $\quad \Delta = \frac{\partial cost}{\partial x}$
4: $\quad \Delta = \Delta \cdot M$
5: $\quad x = x - lr \cdot \Delta$
6: $\quad x = clip(x, val_{min}, val_{max})$
7: $\quad epc = epc + 1$
8: **return** x;

At the end of the loop, a new pattern is generated with pixels' values optimized to provoke the saturation of the target neuron. In general, the target neuron will not reach $target_{value_k}$ but a lower value, namely $final_{value_k}$. A *threshold* is chosen, such that if the neuron's output value exceeds it, the output of the comparator is set high and the multiplexers are switched so that for each targeted weight, the selected bit is complemented.
The *threshold* is derived through the following formula:

$$threshold = final_{value_k} - \xi,$$

where ξ is a parameter, that can be chosen accordingly to the method of pattern application and the parallelism of the network (in case of a DNN) or the rate encoding difference in normal and triggered execution (in case of a SNN).

6.4 Trigger application

The trigger can be applied on the image in mainly two ways: (1) as a **stamp** on the image or (2) as a **noise** on the image. In the first case, the values of the pixel in the pattern area are exactly the optimal ones as generated by the loop described in the previous section so that the wanted value for the output of the target neuron is for sure reached. However, this solution could be less imperceptible and in that case a careful choice of the layer and/or a careful choice of the trigger mask parameters (position, dimension, max_{val}) should be taken into consideration. This type of trigger patch can be easily deployed on traffic signs.
The second mode of application produced good results, even if not as much as the other solution, due to an increased imperceptibility, as will be shown in the following Section 7. The variation in the image is, in fact, appearing as an added noise. This noise, added to the image, can be applied to the

camera vision system, as reported for example in [33], in which coloured stickers are applied on the camera to act like an adversarial and to fool the image recognition system.
Moreover, supposing to have some knowledge about the pixel intensity distribution on the image dataset targeted by the network, the choice of the trigger parameters can rely also on these information.

Chapter 7

Results

7.1 Experimental Setup

Both the original and the modified datasets are inferred and the amount of times for which both datasets make the target neuron exceed the threshold is recorded. There is the possibility that some images from the original dataset produce the saturation of the target neuron as well, hence an unwanted activation of the hardware trojans for an $exceed_{ORIGINAL}$ amount of times. For a stealthy attack purpose, a carefully crafted trigger should lead to a situation in which this value, if not nullified, is kept very low.
Therefore, the accuracy is not excessively reduced when the input trigger pattern is not present, i.e., the presence of hardware trojans is stealthy. The number of images in the dataset is $dim_{DATASET}$, the number of images from the original dataset for which the threshold for the target neuron is exceeded is $exceed_{ORIGINAL}$ and the number of images from the modified dataset for which the threshold for the target neuron is exceeded is $exceed_{MODIFIED}$. The attack aims at being both effective and stealthy, and thereby to simultaneously satisfy the following conditions:

1. $exceed_{ORIGINAL} << exceed_{MODIFIED}$
2. $exceed_{ORIGINAL} << dim_{DATASET}$
3. $exceed_{MODIFIED} \simeq dim_{DATASET}$.

Although all these conditions should be jointly satisfied is more important to have a stealthy attack which is less effective than a very effective attack which is not stealthy, namely is considered mandatory to satisfy the second condition. The conditions to be satified can be rewritten as:

1. $exceed_{ORIGINAL} = 0$
2. $exceed_{MODIFIED} \simeq dim_{DATASET}$

In the following, the results obtained on the MNIST, CIFAR10 and GTSRB datasets are described.

7.1.1 Results on the MNIST dataset

Convolutional Neural Network

Targeting the **first convolutional layer** of a CNN with parameters listed in the first row of Table 7.1 the trigger shown in Figures 7.1-d, is produced.

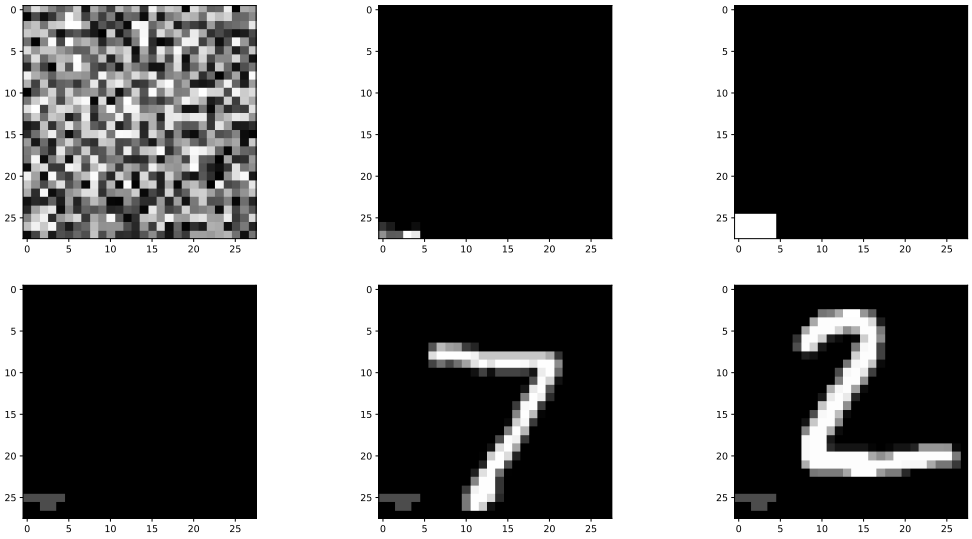

Figure 7.1: From top left to bottom right: (a) initial input trigger, (b) gradients of the selected neuron, (c) mask created through gradients, (d) final trigger after loop, (e) and (f) two images with applied trigger.

In Figure 7.1-a,b,c the *random initial image*, the initial gradients and the mask M are respectively shown. In this case, the mask is crafted to follow the shape of the gradients. The images from both the original and the modified test set (two examples from this last image set are shown in Figure 7.1-e,f) are inferred and the results, as reported in Table 7.1, are: $exceed_{ORIGINAL} = 0$, $exceed_{MODIFIED} = 10\,000$. This is the better result obtained.
Targeting the **second convolutional layer**, the produced results are significantly different. In fact, the trigger is far more perceptible and highly superimposed with the significant part of the images, as can be seen in Figure 7.2. In this case, with the same simulation parameters, the obtained statistics about the threshold exceeding are: $exceed_{ORIGINAL} = 5$ and $exceed_{MODIFIED} = 7\,585$, as also reported in Table 7.1. This points out that targeting a neuron belonging to the second convolution layer leads to a worse result. In fact, it can be pointed out that the gradients are, on average, higher than the gradients referred to a target neuron belonging to

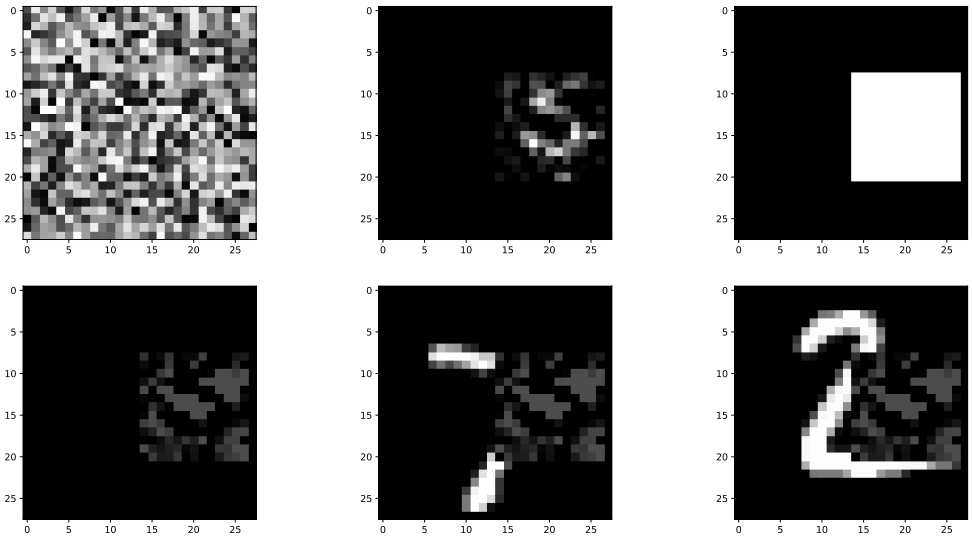

Figure 7.2: From top left to bottom right: (a) initial input trigger, (b) gradients of the selected neuron, (c) mask created through gradients, (d) final trigger after loop, (e) and (f) two images with applied trigger.

the first convolution layer. We define the correlation between the target neuron and the masked part of the image S as follows:

$$S = \frac{\sum_{i,j}^{N} \gamma_{i,j}}{M^2},$$

where $\gamma_{i,j}$ is the gradient referred to the pixel with indexes i,j in the trigger mask and M is the size of the side trigger, in case of a square trigger. It can be seen that in the first convolution layer $S = 2.21 \cdot 10^{-5}$, whereas in the second convolution layer $S = 1.4 \cdot 10^{-6}$. This clearly shows that, for a neuron in the 2^{nd} layer, the variation with the input pixel is much lower.
If we call ρ the value

$$\rho = \frac{(exceed_{MODIFIED} - exceed_{ORIGINAL}) \cdot 100}{dim_{DATASET}},$$

we can see that it is getting lower when choosing target neurons belonging to deeper layers.

Multilayer Perceptron

Taking into consideration the MLP, each neuron, also at the first layer, is connected to each pixel of the input image. Therefore, the gradients of the neuron with respect to the input image, spread all over the image itself. Hence, a square mask is created to limit the area of the trigger. These results can be applied to the dense layers of a CNN as well. The trigger is added to the image in a way for which the right bottom corner of the square matches the right bottom corner of the 28×28 image. The side size of the square trigger is varied between 5 and 17 pixels, with steps of 2 pixels. As it can be seen in Figure 7.3, with a trigger size of just 5×5 pixels, the value of ρ is far from the optimal one. This is due to the fact that if the area of the trigger is too small, the amount of pixels whose value has to be optimized by the algorithm to reach the *target value* for the *target neuron*, is too low. It should be underlined that, in the case of the MLP, the *target neuron*'s output value is depending on all the pixels of the input image. The difference between $initial_{value_k}$ and $final_{value_k}$ results to be small. Moreover, a huge number of images from the original dataset make the target neuron exceed the threshold, leading to a small value of ρ. As previously said, this condition is the critical one. A larger area of the trigger increases ρ as can be seen in Figure 7.3 but, on the other hand, leads to a less imperceptible trigger pattern.

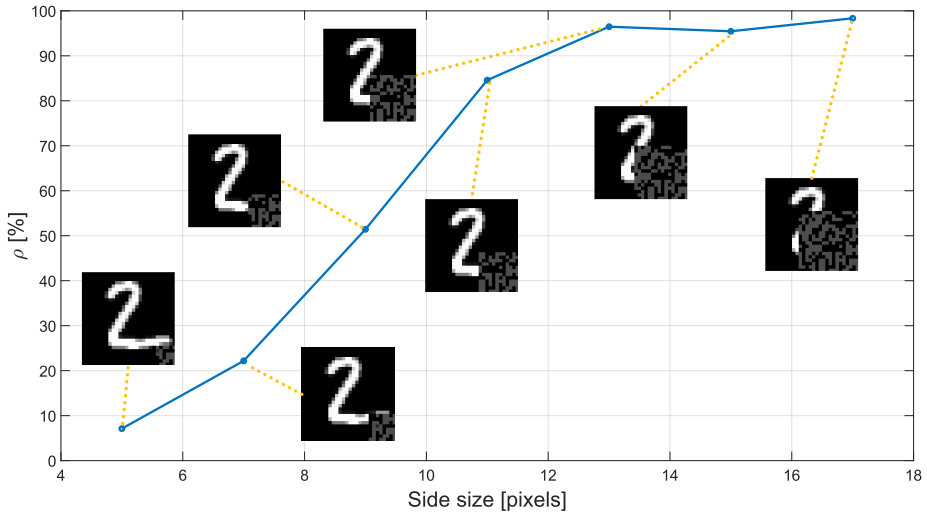

Figure 7.3: Plot of ρ with respect to the trigger size.

In the case of the MLP network, an interesting result is obtained imposing a lower value of $max_{val} = 0.1$. Even though we are targeting the first layer, the gradients are covering all the image (Figure 7.5-b), since it is a fully

connected layer. Hence, we create a mask suited for the gradients, which spans across the whole image, as shown in Figure 7.5-c. In this case, the second method described in section 6.4 is used to apply the pattern, hence the pattern is applied as a noise on the image. Due to the low value of val_{max}, the trigger results to be imperceptible, as shown in Figures 7.5-e,f. We obtained a very high ρ, shown in Table 7.1, and higher imperceptibility.

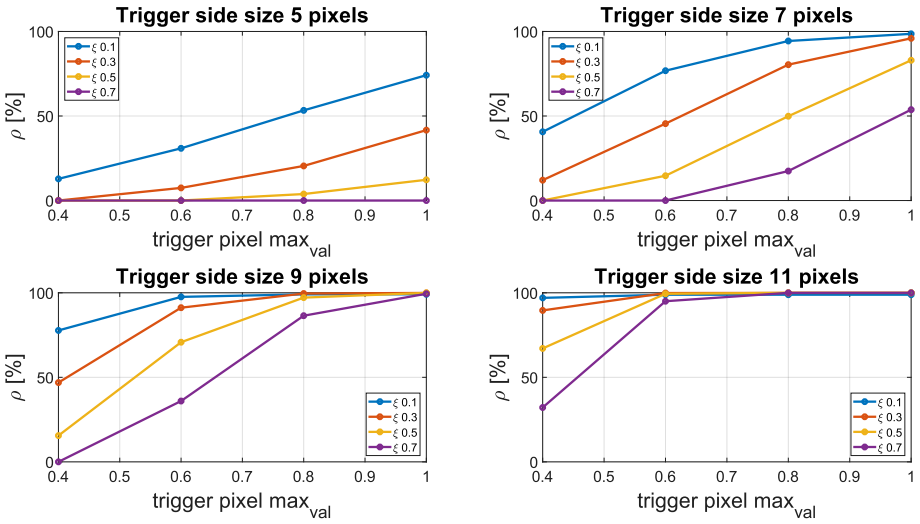

Figure 7.4: Values of ρ for different values of trigger side dimension, max_{val} of pixels and ξ.

It can be seen in Figures 7.4 that the lines are monotonically increasing with increasing value of max_{val}. Having a smaller value of ξ is always better. It can be also pointed out that having an higher area for the trigger is good but, of course, a smaller trigger could be less perceptible even if we are always talking about a stamp on the image.
These considerations on the MLP structure can be also extended to the dense layers in CNNs. In fact, the structure of a CNN is always including one or more dense layers after the convolutional and pooling layers for the final classification stage.

7.1.2 Results on the CIFAR10 and on the GTSRB datasets

In this case, targeting the **first convolutional layer**, with parameters set as shown in Table 7.1, the trigger shown in Figure 7.6-d is produced. The superposition of the trigger on the original images (two examples) is shown in Figures 7.6-f,h. For the GTSRB dataset, results are very good as well. In fact, a deep separation of the activations (spiking activity) during clean images and patterned images during inference is reached. Two examples of clean and patterned images for the GTSRB are reported in Figure 7.7. Results obtained for CIFAR10 and GTSRB are reported in Table 7.1.

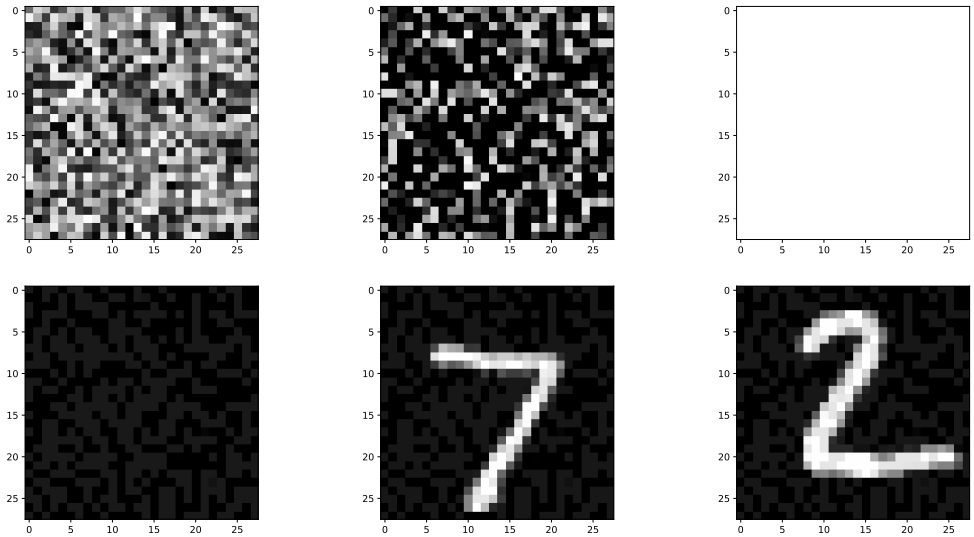

Figure 7.5: From top left to bottom right (a) initial input trigger, (b) gradients of the selected neuron, (c) mask created through gradients, (d) final trigger after loop, (e) and (f) two images with applied trigger.

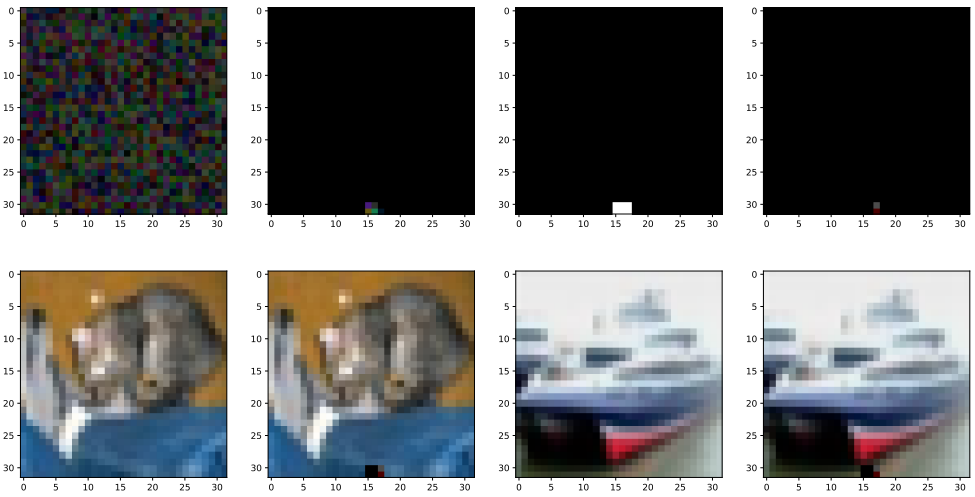

Figure 7.6: From top-left to bottom-right: (a) initial input trigger, (b) gradients of the selected neuron, (c) mask created through gradients, (d) final trigger after loop, (e) first image from the dataset (f) first image with trigger applied (g) second image from the dataset (h) second image with trigger applied.

Table 7.1: Dataset, type of the networks, parameters' values and results.

Dataset	Net	Layer	Parameters		ρ
MNIST	CNN	1st Conv2D	val_{max}	0.3	100%
			ξ	0.1	
			$target_{OUTPUT_k}$	100	
			$initial_{VAL_k}$	0.04	
			$final_{VAL_k}$	0.21	
	CNN	2nd Conv2D	val_{max}	0.3	75.8%
			ξ	0.1	
			$target_{OUTPUT_k}$	100	
			$initial_{VAL_k}$	0.08	
			$final_{VAL_k}$	1.56	
	MLP	1st Dense	val_{max}	0.1	98.99%
			ξ	0.1	
			$target_{OUTPUT_k}$	100	
			$initial_{VAL_k}$	0.05	
			$final_{VAL_k}$	1.21	
CIFAR10	CNN	1st Conv2D	val_{max}	0.3	99.98%
			ξ	0.1	
			$target_{OUTPUT_k}$	100	
			$initial_{VAL_k}$	0.02	
			$final_{VAL_k}$	0.23	
GTSRB	CNN	1st Conv2D	val_{max}	0.5	99.97%
			ξ	0.1	
			$target_{OUTPUT_k}$	100	
			$initial_{VAL_k}$	0.10	
			$final_{VAL_k}$	1.01	

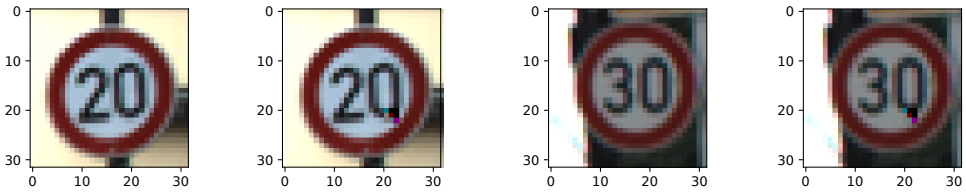

Figure 7.7: From left to right: (a) 20 km/h speed limit (b) 20 km/h speed limit with pattern applied (c) 30 km/h speed limit (d) 30 km/h speed limit with pattern applied.

7.2 Hardware Overhead

Given the amount M of bit-flips applied, the hardware overhead should be constituted by:

1. M inverters, made of 2 transistors each;
2. M 2-way multiplexer, constituted by 16 transistors each in a 4 NANDs implementation;
3. *in the case of a DNN*, a digital comparator, whose complexity depends on the parallelism of the neuron's output result, which is connected to the target neuron's output, or
4. *in the case of an SNN*, a counter, to count the spikes, plus a comparator which is set when the counter reaches a particular value.

The overhead of multiplexers and inverters can be estimated as $(2+16) \times M$. From the simulations reported in section 5.2, it is clear that an amount of about just 30 bit-flips is enough to completely crash the performances of the DNN for the two networks operating on MNIST dataset, or 4 bit-flips in the case of the CNN operating on the CIFAR10 dataset. The hardware overhead of inverters and multiplexers, calculated in terms of transistors, is about $(2+16) \times 30 = 540$ in the first case, whereas it is just $(2+16) \times 4 = 72$ in the second case. In the case of a SNN, a counter is added, whose module should be at least as much as the maximum spiking rate a neuron can have. The amount of transistors needed for a module N counter are given by $\#_{transistors} = (N-2) \times 6 + (N \times 4) \times 4$, where the first addend gives the contribution of the AND gates, whereas the second gives the contribution of the T-type flip-flops. The overhead in terms of transistors can be considered low with respect to the large amount of transistors in a DNN or SNN accelerator. However, as explained in section 3.3, sophisticated detection techniques are available and therefore this hardware overhead cannot be considered as negligible and must be reduced.

7.3 Hardware Overhead Reduction

The considerations made on hardware must be revisited. In fact, different problems raised when trying to figure out how the attack can be effectively deployed on physical hardware. It must be updated due to two principal motivations:

- the hardware overhead can still be too high so that the prescence of the trojan can be highlighted, for example, by a simple differential power detection technique;
- the hardware trojan can be optimized for the specific hardware.

A complete knowledge of the low-level hardware implementation and related mapping algorithms, by hypothesis, must be known to perform the attack. An attack against a reconfigurable neuromorphic hardware (Intel Loihi, IBM TrueNorth, ARM SpiNNaker) is difficult to be realized. The target must be a known neuromorphic hardware suited for a specific application:

- the location of the target neuron axon (output) must be known to introduce the *detection* circuitry;
- the locations and the content of the memory cells to flip must be known.

Trigger Detection The ideal condition to deploy the detection circuit would be the one for which each connection between neurons is hard-wired: each synapses is mapped to a different conductive line in the hardware. However,in general, this is not the real situation in the current hardware implementations of SNNs, where the routing is multiplexed. This is not only due to allow the reconfigurability of the hardware but is also mandatory due to the huge amount of virtual synapses in a NN. In fact, synapses area occupation is the major problem to be addressed in current neuomorphic hardware implementation. Hence, to detect the spiking activity of the target neuron, besides knowing which is the physical shared routing path in which the target synapses is laying, it has to be known also the signal which is transferred.

Spike Counter The main issue is that in SNNs no digital activation value for the activation of the target neuron is present. Therefore, there is not the possibility to directly compare the activation value with a particular digital threshold to trigger the bit-flip. This could be also an advantage, which allows for an additional overhead reduction. In fact, the use of a comparator can be avoided and the amount of spikes at the output of the target neuron could be counted instead. In addition, not only the counter, but also the hardware overhead for a digital counter wants to be avoided. Besides, the complete count would be ready just at the end of the time duration of simulation for that image. This implies that the spike rate cannot be directly considered as a trigger source. Anyway, an higher spiking rate, caused by the input pattern applied to the image, is for sure providing an higher amount of spikes.
All these considerations lead to take into consideration an **analog solution**. In Figure 7.8 a simple detection circuit schematic for the trigger signals is shown. When a spike travels on the synapses, the nmos transistor is turned on for a t_{spike} amount of time. Here the spike is considered as a square pulse from GND to V_{DD}; this choice (1) is due to a simplistic assumption (2) details about spike shape are not clarified in literature on neuromorphic

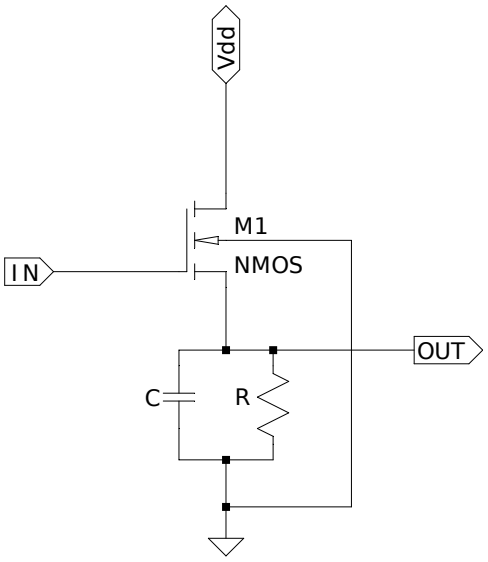

Figure 7.8: Schematic of the detection circuit.

hardware (3) is not a limitation, in fact, if spike communication is carried on by actual spike signals, there are more sophisticated charge-sharing techniques to measure them (one example is provided by Yang et al. [55]).
The RC parallel is charged to the supply voltage V_{dd}; the charge-discharge curve depends on the parameters R, C and the turn-on time of the transistor, which in turn depends on the pulse width t_{spike}. To have a good trigger detection circuit, the parameters R and C must be optimized taking into consideration also t_{spike}. What is wanted is that the output, namely the trigger signal T is at high level after a certain amount of spikes, i.e. the spiking activity threshold found through the methodology in Chapter 6, is reached. The discharge curve is critical: what is wanted is that after a short time after the triggered image, the trigger signal T returns to be low again and consequently the correct logical value in the memory cell is recovered, hence the trigger is stealthy again. For example, if a triggered image and then a clean image are inferred, the trigger signal is wanted to be at high logical level just for the first one. Therefore the discharge time of the RC parallel must be adjusted to satisfy this requirement. In the case reported in Figure 7.9, the specifications of a trigger circuit can be relaxed since the spiking rate of the original image is null, i.e. the target neuron is spiking zero times when the clean image is inferred. The output voltage can finally be used to activate the bit-flip circuit, for example, by being directly connected to one of the inverters in a 6T SRAM cell. What is also wanted is that the RC parallel is quickly discharged after the triggering sequence is passed.
In Figure 7.9 the spiking activity of the *target neuron* of GTSRB CNN, for a 50 ms long simulation, is shown. In particular, it is shown the spiking

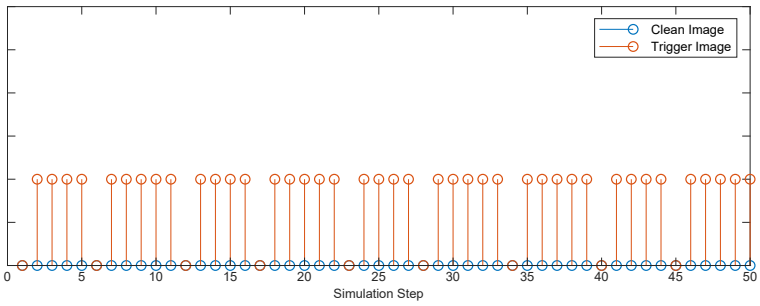

Figure 7.9: Spiking activity of the target neuron for one random image taken from the MNIST dataset without (blue) and with the pattern applied (red).

activity of the *target neuron* of first convolutional layer, for a random image of GTSRB dataset without the trigger pattern applied (clean image, blue lines) and with the trigger pattern applied (trigger image, red lines). The clean image is reported in Figure 7.7-a and the triggered one is in Figure 7.7-b.
In Figure 7.9, the deep difference in spiking rates can be clearly seen: the triggered image makes the target neuron spike 41 times out of 50 possible, whereas the original clean image was leading to 0 spikes so that the spike rate is raised from 0 Hz to 820 Hz. This great difference in firing rates can be used to activate the hardware trojan, as previously said.

Bit-flip in SRAM Many solutions have been considered to trigger the bit-flip, however some of them have been discarded for different reasons. It has been chosen to adopt solutions to, again, minimize the hardware overhead and to avoid bit-flips triggered only during read operation of the target bit due to the tight timing constrains of read operation in SRAMs. Therefore, it has been decided to flip the content of the cell directly inside the cell.
These solution is effective and highly stealthy, because no hardware is added so that the area and power consumption can be kept almost unchanged. Considering the original SRAM cell (Figure 7.10), from hypothesis it is known *where*, so, in which particular cell, is stored the target bit and *which* is the target bit stored in that cell. Considering Q as the correct stored value

- if Q = '0', $\overline{Q}$ = '1', break the connection $\overline{Q} \rightarrow Q_{IN}$ and make the connection $T \rightarrow Q_{IN}$ (Figure 7.11, top)
- if Q = '1', $\overline{Q}$ = '0', break the connection $Q \rightarrow \overline{Q}_{IN}$ and make the connection $T \rightarrow \overline{Q}_{IN}$ (Figure 7.11, bottom)

It can be clearly seen how, when the trigger is active, therefore T = '1', the correct content of the cell is flipped.

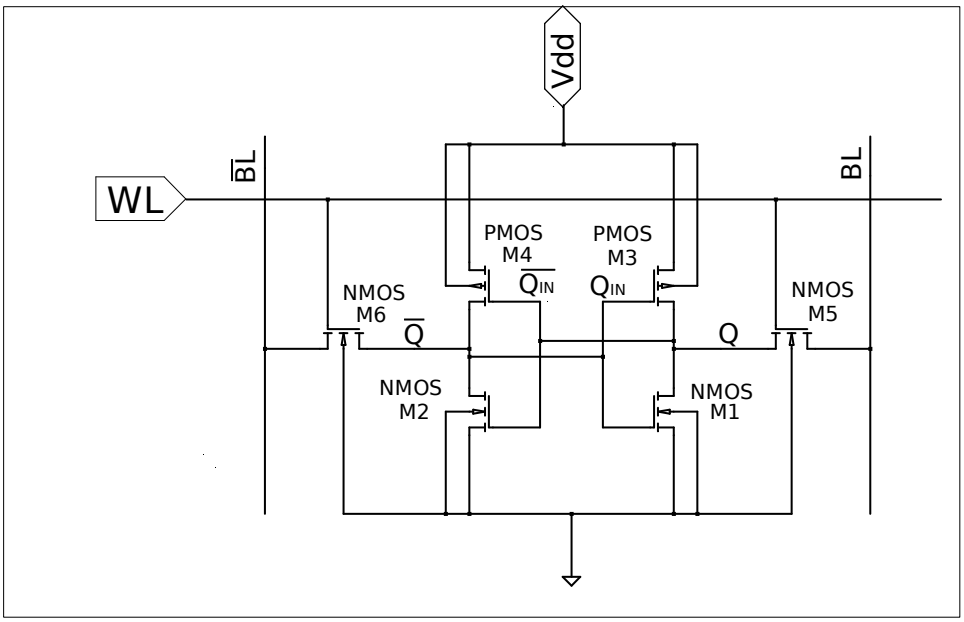

Figure 7.10: Schematic of the original 6T SRAM cell.

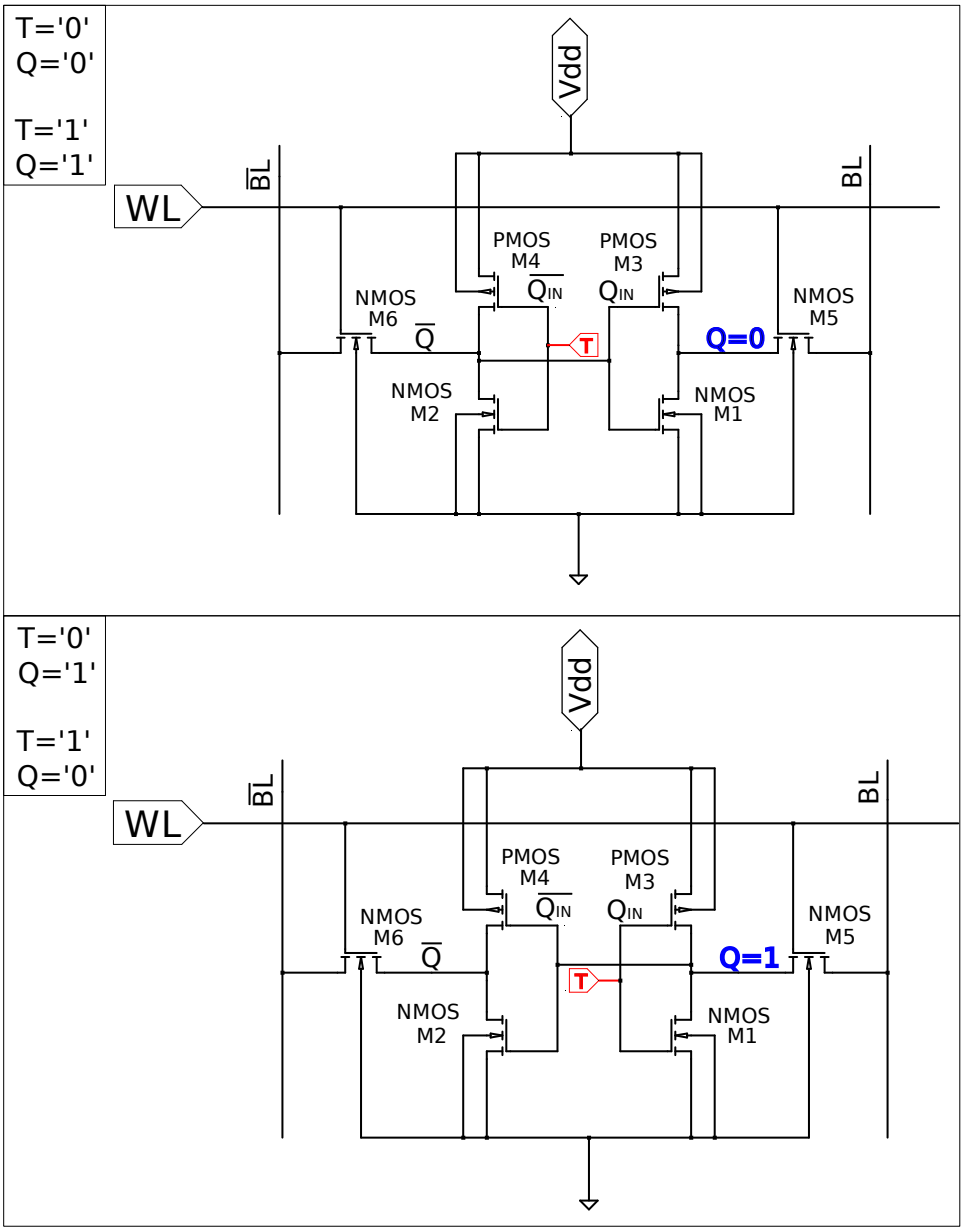

Figure 7.11: Schematic of the 6T SRAM cell (top) with trigger connection and original condition $Q = 0$ (bottom) and with trigger connection and original condition $Q = 1$.

Chapter 8

Conclusion

In the thesis work, threatens on the reliability of DNNs and SNNs are discussed. As predicted, Statistical Fault-Injection is found to be not the best solution when the response of a neural network, subject to parameters errors, is wanted to be inspected. The coss-layer analysis confirmed that neural networks show high resilience in presence of random errors. A layer-wise Statistical Fault Injection, however, shown that some layers are way less robust than others. This difference in robustness between layers is found to be due to a combination of sources like the type of the layer, the relative position in the network, the amount of parameters.
On the contrary it has been demonstrated that a targeted Fault-Injection can dramatically undermine the accuracy of a DNN or an SNN by applying bit-flips on a few vulnerable parameters. These parameters are found through a gradient search algorithm; despite being able to obtain excellent results, it is not always capable of finding the best solution, which is, from an attacker point of view, the most consistent reduction in accuracy with the lowest amount of injections. This has been underlined by the fact that, following a layer-wise approach, that is constraining the gradient search algorithm to work on a selected layer, the attack is more effective. This is highlighting how possible sources of threat can escape detection and how a layer-wise analysis should be always applied for testing.
A hardware trojan, supposed to be placed in the supply chain, and triggered externally by an adversarial input pattern, is deployed. It is supposed to be a white-box setting since the attacker has complete knowledge, from hypothesis, of the architecture, internal parameters and low-level implementation information. It has been shown how is easy for an attacker to leverage its knowledge about the system and to craft a targeted stealthy attack. It has been demonstrated how an attacker can easily target a few vulnerable parameters by means of payload circuits of a hardware trojan, exploited as bit-flipping circuits hidden in the hardware of a NN. Due to the linear relationship between DNN activations and SNN spike rates, the attacker

can indiscriminately target an SNN or a DNN with just a few adaptations of the hardware. This proves that the attack represents a threat to both SNNs and DNNs. The security issue is made more severe especially by the stealthiness of the trojan attack because it is only effective when triggered by the external adversarial noise, and practically imperceptible elsewhere. Then, the hardware trojan overhead minimization is carried on, focusing the attack on an SNN hardware accelerator. The added hardware is crafted to minimize the overhead both in terms of area and power consumption to escape trojan detection. The trigger pattern applied to the input image is considered to be imperceptible just through visual inspection. Increasing the imperceptibility of the trigger pattern and applying a metric to measure its imperceptibility could constitute future work.

Bibliography

[1] J.-L. Autran, S. Semikh, D. Munteanu, S. Serre, G. Gasiot, and P. Roche. *Soft-Error Rate of Advanced SRAM Memories: Modeling and Monte Carlo Simulation*. 09 2012. ISBN 978-953-51-0749-1. doi: 10.5772/50111.

[2] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni. Countermeasures against fault attacks on software implemented aes: Effectiveness and cost. In *Proceedings of the 5th Workshop on Embedded Systems Security*, WESS '10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300780.

[3] D. Beeman. Hodgkin-huxley model. In *Encyclopedia of Computational Neuroscience*, 2013.

[4] S. M. Bohte, J. N. Kok, and H. L. Poutré. Error-backpropagation in temporally encoded networks of spiking neurons. *Neurocomputing*, 48 (1):17 – 37, 2002.

[5] S. A. Bota, G. Torrens, I. de Paúl, B. Alorda, and L. A. Segura. Accurate alpha soft error rate evaluation in sram memories. In *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*, pages 205–209, July 2013. doi: 10.1109/IOLTS.2013.6604080.

[6] M. Bouvier et al. Spiking neural networks hardware implementations and challenges: A survey. *ACM Journal on Emerging Technologies in Computing Systems*, 15:1–35, 04 2019.

[7] P. Cardaliaguet and G. Euvrard. Approximation of a function and its derivative with a neural network. *Neural Networks*, 5, 1992.

[8] J. Clements and Y. Lao. Hardware trojan attacks on neural networks. *CoRR*, abs/1806.05768, 2018.

[9] M. Davies et al. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, 2018.

[10] H. Fuketa, M. Hashimoto, Y. Mitsuyama, and T. Onoye. Neutron-induced soft errors and multiple cell upsets in 65-nm 10t subthreshold sram. *IEEE Transactions on Nuclear Science*, 58(4):2097–2102, Aug 2011. ISSN 1558-1578. doi: 10.1109/TNS.2011.2159993.

[11] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana. The spinnaker project. *Proceedings of the IEEE*, 102(5):652–665, May 2014. ISSN 1558-2256. doi: 10.1109/JPROC.2014.2304638.

[12] G. J. Gibson, S. Siu, and C. F. N. Cowen. Multilayer perceptron structures applied to adaptive equalisers for data communications. In *International Conference on Acoustics, Speech, and Signal Processing,*, pages 1183–1186 vol.2, May 1989.

[13] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples, 2014.

[14] T. Gu, B. Dolan-Gavitt, and S. Garg. Badnets: Identifying vulnerabilities in the machine learning model supply chain. *CoRR*, abs/1708.06733, 2017. URL `http://arxiv.org/abs/1708.06733`.

[15] O. Guillen, M. Gruber, and F. De Santis. Low-cost setup for localized semi-invasive optical fault injection attacks. pages 207–222, 07 2017. ISBN 978-3-319-64646-6.

[16] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

[17] M. A. Hanif, R. Hafiz, and M. Shafique. Error resilience analysis for systematically employing approximate computing in convolutional neural networks. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 913–916, March 2018.

[18] H. Hazan et al. Bindsnet: A machine learning-oriented spiking neural networks library in python. *Frontiers in Neuroinformatics*, 2018.

[19] T. Heijmen. Soft error rates in deep-submicron cmos technologies. In *12th IEEE International On-Line Testing Symposium (IOLTS'06)*, pages 1 pp.–, July 2006. doi: 10.1109/IOLTS.2006.57.

[20] T. Heijmen and A. Nieuwland. Soft-error rate testing of deep-submicron integrated circuits. volume 2006, pages 247 – 252, 06 2006. ISBN 0-7695-2566-0. doi: 10.1109/ETS.2006.42.

[21] X. Hou, J. Breier, D. Jap, L. Ma, S. Bhasin, and Y. Liu. Experimental evaluation of deep neural network resistance against fault injection attacks. Cryptology ePrint Archive, Report 2019/461, 2019. `https://eprint.iacr.org/2019/461`.

[22] E. M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, Nov 2003.

[23] Y. Kim et al. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 361–372, June 2014.

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

[25] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Neural Information Processing Systems*, 25, 01 2012. doi: 10.1145/3065386.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998.

[27] Y. Lecun et al. Gradient-based learning applied to document recognition. 1998.

[28] C. Lee, S. S. Sarwar, and K. Roy. Enabling spike-based backpropagation in state-of-the-art deep neural network architectures. *CoRR*, abs/1903.06379, 2019.

[29] H. Lee, R. Grosse, R. Ranganath, and A. Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. page 77, 01 2009. doi: 10.1145/1553374.1553453.

[30] J. Lee, T. Delbrück, and M. Pfeiffer. Training deep spiking neural networks using backpropagation. *CoRR*, abs/1608.08782, 2016.

[31] 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. doi: 10.1109/DATE.2009.5090716.

[32] H. Li, Q. Liu, and J. Zhang. A survey of hardware trojan threat and defense. *Integration, the VLSI Journal*, 55, 02 2016. doi: 10.1016/j.vlsi.2016.01.004.

[33] J. Li, F. R. Schmidt, and J. Z. Kolter. Adversarial camera stickers: A physical camera-based attack on deep learning systems, 2019.

[34] Y. Liu et al. Fault injection attack on deep neural network. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 131–138, Nov 2017.

[35] Y. Liu et al. Trojaning attack on neural networks. 01 2018.

[36] F. Martinelli, G. Dellaferrera, P. Mainar, and M. Cernak. Spiking neural networks trained with backpropagation for low power neuromorphic implementation of voice activity detection, 2019.

[37] P. A. Merolla et al. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 2014.

[38] S. Mitra, H. . P. Wong, and S. Wong. The trojan-proof chip. *IEEE Spectrum*, 52(2):46–51, February 2015. ISSN 1939-9340. doi: 10.1109/MSPEC.2015.7024511.

[39] E. O. Neftci, H. Mostafa, and F. Zenke. Surrogate gradient learning in spiking neural networks: Bringing the power of gradient-based optimization to spiking neural networks. *Signal Processing Magazine*, 2019.

[40] M. A. Neggaz, I. Alouani, P. R. Lorenzo, and S. Niar. A reliability study on cnns for critical embedded systems. In *ICCD*, Oct 2018.

[41] M. A. Neggaz, I. Alouani, S. Niar, and F. Kurdahi. Are cnns reliable enough for critical applications? an exploratory study. *IEEE Design Test*, pages 1–1, 2019.

[42] M. Pfeiffer and T. Pfeil. Deep learning with spiking neurons: Opportunities and challenges. *Frontiers in Neuroscience*, 12: 774, 2018. ISSN 1662-453X. doi: 10.3389/fnins.2018.00774. URL `https://www.frontiersin.org/article/10.3389/fnins.2018.00774`.

[43] J. A. Pérez-Carrasco, B. Zhao, C. Serrano, B. Acha, T. Serrano-Gotarredona, S. Chen, and B. Linares-Barranco. Mapping from frame-driven to frame-free event-driven vision systems by low-rate rate coding and coincidence processing–application to feedforward convnets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(11): 2706–2719, Nov 2013. ISSN 1939-3539.

[44] A. S. Rakin, Z. He, and D. Fan. Bit-flip attack: Crushing neural network withprogressive bit search. *CoRR*, abs/1903.12269, 2019.

[45] A. S. Rakin, Z. He, and D. Fan. Bit-flip attack: Crushing neural network withprogressive bit search. *CoRR*, abs/1903.12269, 2019. URL `http://arxiv.org/abs/1903.12269`.

[46] B. Reagen et al. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016.

[47] B. Reagen et al. Ares: A framework for quantifying the resilience of deep neural networks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2018.

[48] J. Rodriguez, A. Baldomero, V. Montilla, and J. Mujal. Llfi: Lateral laser fault injection attack. In *2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 41–47, Aug 2019.

[49] B. Rueckauer, I.-A. Lungu, Y. Hu, M. Pfeiffer, and S.-C. Liu. Conversion of continuous-valued deep networks to efficient event-driven networks for image classification. *Frontiers in Neuroscience*, 11:682, 2017.

[50] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[51] A. Tavanaei, M. Ghodrati, S. R. Kheradpisheh, T. Masquelier, and A. Maida. Deep learning in spiking neural networks. *Neural Networks*, 111:47–63, Mar 2019. ISSN 0893-6080. doi: 10.1016/j.neunet.2018.12.002. URL `http://dx.doi.org/10.1016/j.neunet.2018.12.002`.

[52] R. Vaila, J. Chiasson, and V. Saxena. Deep convolutional spiking neural networks for image classification. *CoRR*, abs/1903.12272, 2019.

[53] Z. Wang, L. Guo, and M. Adjouadi. A generalized leaky integrate-and-fire neuron model with fast implementation method. *International journal of neural systems*, 24:1440004, 08 2014.

[54] R. Wiyatno and A. Xu. Maximal jacobian-based saliency map attack. *CoRR*, abs/1808.07945, 2018. URL `http://arxiv.org/abs/1808.07945`.

[55] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester. Exploiting the analog properties of digital circuits for malicious hardware. *Communications of the ACM*, 60:83–91, 08 2017. doi: 10.1145/3068776.

[56] J. Ye, Y. Hu, and X. Li. Hardware trojan in fpga cnn accelerator. In *2018 IEEE 27th Asian Test Symposium (ATS)*, pages 68–73, Oct 2018.

[57] X. Yuan, P. He, Q. Zhu, R. R. Bhat, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *CoRR*, abs/1712.07107, 2017.

[58] H. Ziade, R. Ayoubi, and R. Velazco. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1:171–186, 01 2004.