POLITECNICO DI TORINO

Master of science in Computer Engineering

Master Degree Thesis

Analysis of neural network reliability in safety-critical applications

Supervisors

Prof. Ernesto Sanchez

Prof. Paolo Bernardi

Candidate

Alessandro Ianne

Academic Year 2018 – 2019

This page has been intentionally left blank

Abstract

Convolutional Neural Networks(CNNs) are becoming a widely adopted computational model in many fields. These networks achieve great performances in object detection, data analysis and visual recognition tasks. Besides, one interesting application is pedestrian detection in real-time for self-driving cars domain. Since they are deployed in such safety-critical environments, it starts to be mandatory to perform a careful evaluation of their reliability. The main intent of this master thesis is to analyze the reliability of CNNs during inference phase to characterize their behaviour in a faulty scenario, i.e., when the classification results deviate from the correct one. In this work we propose a methodology that evaluates reliability through a fault-injection campaign, by means of a deliberate insertion of faults into the target model, aiming to evaluate the effective convolutional neural network behaviour under faulty-conditions. In a preliminary phase, faults have been injected at bit granularity into the weight floating-point representation. In a second step, since the key operation during convolution is the multiplication, the same operation has been repeated by a fault injection at the output of multiplications. Experimental results show that the network works properly only for stuck restricted to certain locations and, ensuring the same accuracy level of a fault-free inference. For stuck-at 1 fault injected into the MSB of the floating-point exponential part we achieved a 12.3% of faults that corrupts output results.

Dedicato alla mia famiglia e ai miei cari nonni

Acknowledgements

I would like to thank all the people who contributed to this master thesis, starting from professor Ernesto Sanchez. My sincere gratitude goes also to Professor Paolo Bernardi and Professor Alberto Bosio for their advice and suggestions. Thanks to Annachiara Ruospo and all the guys of the Lab3 at Dauin, with you I shared good moments during this journey.

I would also like to thank my friends Davide and Manuel, with you I shared good and bad times and I'll never know how to repay you. Thanks to my room mates, with you i shared these fantastic three years. I would also like to thank all the people I met during this university years, professors and friends, I always learned something from you!

Contents

Chapter 1

Introduction

Nowadays Deep Neural Networks (DNNs) are widely used in different fields of the applied sciences, thanks to the promising results and their impressive performance compared with other solutions. They belong to a family of artificial intelligence algorithms called brain-inspired, able to perform complex operations previously bounded only to the human supervision.
Convolutional neural networks (CNNs) are one of the most important subsets of these computational models, mainly applied to analyzing visual imagery, where they revealed good classification accuracy for input 2D images. For this reason, they are employed in several computer science fields such as image and video recognition, recommender system and image classification. More than one decade ago, those tasks mainly relied on hand-engineered solution and hand-coded algorithms, since artificial intelligence and machine learning algorithm were not mature enough to cope with the high computational demand required. Today's, the perspective of such applications has totally changed and those old-fashion solutions has been replaced with more performing neural networks based models.

Up to now, the research community put a lot of effort into their studies keeping into consideration the optimization of metrics like model complexity, classification

accuracy, and power consumption. The results obtained from these studies, conducted by both academic and industrial players, go toward the deployment of these models into more constrained hardware platforms, like Smartphones and embedded CPUs. At the same time, very little importance has been attributed to the evaluation of host system failures and in particular how several errors could affect the software stack used during neural network processing. This is a key point for those DNN models, like convolutional neural network, that aims to be deployed into safety-critical scenarios, where system failure leading to loss of life and significant property damage.

During the last years, CNNs have gained a lot of interest in those application domains where a correct real-time classification of the surrounding environment should be performed. With their help, autonomous driving as well robotics applications are capable to recognize an object by analyzing images that come from input cameras and applying actions in response to events. The object detected by the system determines the actions to be applied in response to events. Another example is medical image analysis, in this scenario CNN algorithms process images computed by special medical equipment, making hypothesis about the presence or absence of medical diseases.

In past years there were clear examples of car crashes during tests for autonomous driving evaluation, which caused several victims. These fatalities raised the question about what is the degree of safeness of these types of applications and if there are some methodologies that try to make autonomous car safer. However, up to now, there are no clear ideas about the possible effects of both hardware and software faults that could be raised while performing image classification during autonomous driving or medical image analysis. Therefore what we propose is to perform a reliability analysis to asses CNNs behavior under different faulty scenarios.

1.1 Contribution

The goal of this master degree thesis is to evaluate and characterize the reliability of a neural networks behavior by means of a simulation-based fault injection, aiming to place data corruptions into the key points of the neural network topology. The first attempt has been made by placing permanent faults into the weights value, simulating hardware glitches that affect data storage component such as cache, RAM and register. Afterwards, another part of the experiment has been performed by injecting stuck-at faults in a subset of all the multiplication operations.
Initially, our purpose was to perform such fault injection campaign using specif tool, for an evaluation of these platform at Register Transfer or Architectural of an accelerator. After a brief exploration of the different hardware accelerator proposed by the vendors, we decided to follow an approach that target only the software model of CNNs. For this reason we tried to perform a fault-simulation, starting by one of the most spread DNN library. Obviously, before the actual fault injection simulation, we have developed the fault simulator taking in mind our objectives and. The work done can be summarized in:

- *Fault-injector simulator development*: a new fault-injection simulator has been created, taking as starting point Darknet framework.
- *Fault-injection simulation*: a fault injection has been performed using as target model Lenet5 CNN topology.
- *Result analysis*: the results produced by the simulator are processed and analyzed in order to extract the main information, for this aim different scripts has been developed to extract data.

In the rest of this thesis we try to explain how we developed our simulator and how we performed our simulation. The experimental results are presented at the end of this work, giving to reader an explanation about the value computed.

Chapter 2

Background

This chapter gives a brief introduction of topics that were covered in this master thesis dissertation, aiming to explain the main results achieved up to now by previous studies.

We begin by summarizing the main theory behind neural networks, focusing on convolutional neural networks, the models widely used during this work. They are also the main architectures adopted for many computer science tasks like image recognition, natural language processing. Then we move to a brief discussion about manufacturing testing for digital systems. Among many other techniques used in this area, we will talk about the fault injection method explaining why it has been chosen for our experiments with neural networks.

2.1 Neural Networks

Neural networks are computational models that belong to a set of algorithms that are able to cope with unforeseen circumstances, also called Artificial Intelligence algorithms. This class of procedures is much closer to human behavior like "reasoning" or "learning" concerning other types of programs, allowing to solve problems of rising complexity that previously were only restricted to human capabilities.

The recent breakthrough, due to the exceptional performance in various application domains, is a consequence of the large investigation done in the previous years by computer scientists, biologists and neuro-scientists, who created a solid ground for neural network theory. The results achieved by the research lead scientists towards a new computing paradigm, called brain-inspired computing, borrowing concept and mechanism from the human neural system.
Starting from 1943, the research community, pushed by the pioneering work done by Warren McCulloch and Walter Pitts, has suggested different algorithmic models taking inspiration from their results. With the introduction of their *Linear Threshold Unit (LTU)* [1], a primitive form of network that could compute various logical functions using neurons, they have defined key aspects that are the backbone of artificial neural networks theory.

2.1.1 Multi-Layer Perceptron

The first attempt to reproduce the human brain was done with the introduction of Multi-Layer Perceptron, a theoretical model composed by a set of artificial neurons and a set of input and output connections. The last ones allows the neurons to share information by receiving and forwarding data with their neighbors. It is a weak representation of biological neural network, which is composed by approximately 86 billion biological neurons, the "computing elements", and $10^{14} - 10^{15}$ synapses that interconnect them. The information flows through the system by means of electrical stimulus that are catched and combined by neurons, then the response is transmitted along the axon, the nerve fiber responsible to conduct electrical potential, only if the value computed is above a certain threshold.
Below is depicted a simple architecture composed by input layer, output layer and one single hidden layer.

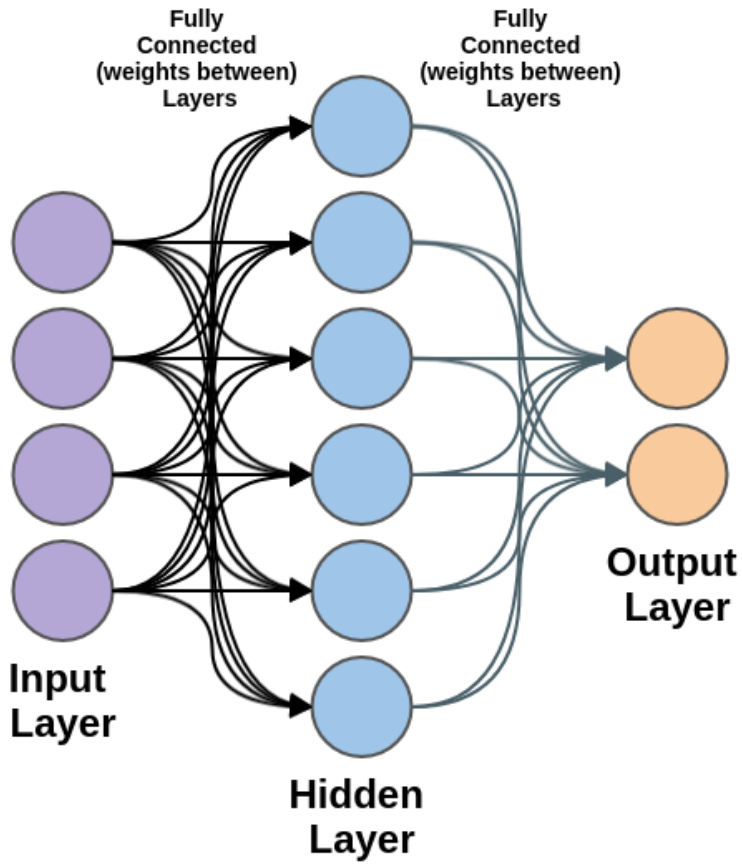

Figure 2.1. Neural network architecture

Artificial neurons are structured in layers, each one is stacked one on the top of the others, creating a simple pipeline. The information enter into the model through the input layer, is processed by the hidden layers and finally the computation results exit from the output layer, creating a so called Directed Acyclic Graph between nodes. Neurons that belong to a given layer can only receive information from previous layer and forward the results to successive layer, therefore the information can only flows from the first to the last layer of the model without cycles. This is a very important property, called in literature Feed Forward, that allows the reduction of the computational load involved during training and inference jobs.

This particular structure can be saw as an hierarchical implementation. enable also networks to extract informative feature that gradually became more abstract when the input is processed by the successive stages. For instance if we need to classify an image, the first hidden layer after the input is in charge to extract information about the pixel color. Successive layers receives and combine information from previous neurons, making it possible to extract edges, lines, figures and so on.

2.1.2 Artificial neurons

The artificial neuron is the basic computational unit behind Multi-Layer Perceptron. This entity, like the biological neuron, receives stimulus and computes only one output at a time without storing a state. Input values are first multiplicated and accumulated with weights, then summed with bias in order to generate the neuron's activation. This value is fed into a non-linear activation function that propagate the output only if a given threshold is reached.

Equation 1 describe how the neuron output is computed, f function represent a generic activation function that should be applied to the weighted sum. In literature we can found different types of activation functions, their main purpose is to convert a input signal of a node into a output signal.

As we can see, every network is composed by a number of parameters that should be defined at design time, like number of weights and the activation function; and learnable parameters, like the weights and bias values.
The exceptional strength of these models allows to arrange the architecture in different ways, allocating the hyper-parameters in such a way that the network can be adapted to every workload.

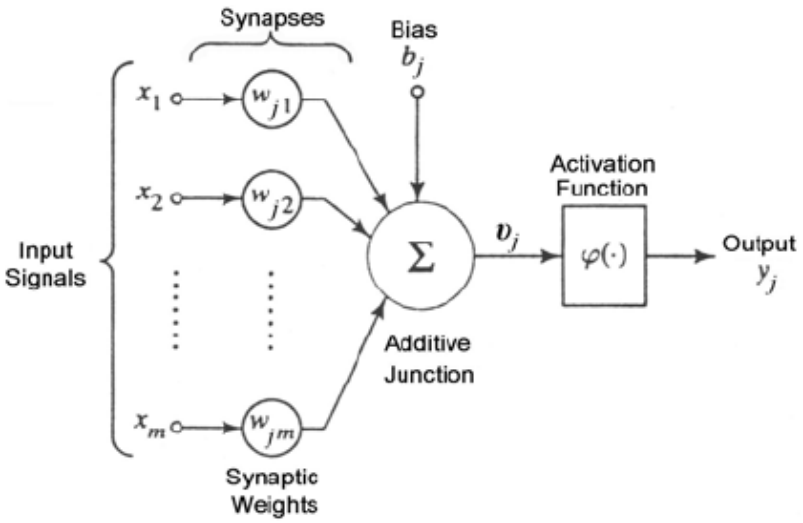

Figure 2.2. Artificial neuron

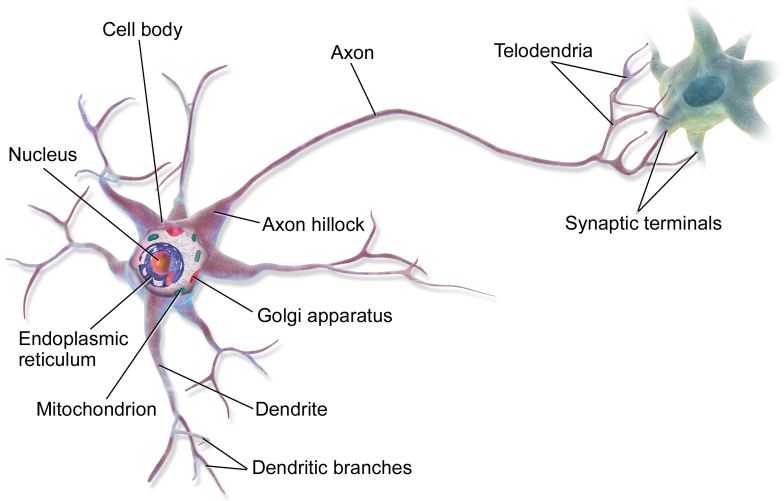

Figure 2.3. Biological neuron

2.1.3 Neural network training

As said previously, weight and bias are not hand-defined, but are learned during training phase. It is a computationally intensive process that delegate to algorithms the parameters choice, but require a careful evaluation since a bad performed training can result into low quality accuracy during inference.

For sake of clarity it is important to briefly describe the Training concept since it is not obvious. When we talk about training we mean a process that aims is to minimize the classification error produced by the network. Actually is an optimization problem where parameters to be optimized are weights and biases, while the objective function to be minimized is expressed as the difference between the value computed by the network and the ground truth label. There exists different approach and procedure that try to solve this problem, in last years Stochastic gradient descend gained a lot of popularity thanks to its good performance compared to other methods. It works using a set of input labelled that are fed during the computation and represent the source of data that the alogorithm uses in order to "teach" the network. This approach it is also called Supervised Learning and differ from Unsupervised Learning since it is strictly necessary to fed the model with human processed values.

Stochastic Gradient Descent is an iterative process, this means that we cannot compute the best solution in one step, but we have to iterate until a satisfactory solution is reached. It simply computes a gradient vector that describe each weight's influence on the error. The weights are than updated by an amount that goes into the gradient opposite direction. The magnitude of the update is governed by the learning rate and it is one of the hyper-parameter that should be carefully choose by the designer. A small amount of the learning rate can increase the time needed to converge to a solution, otherwise a too large learning rate can jump the minimum, resulting in a weak exploration of the weights space.

For a CPU could be very difficult to compute the gradient vector of a multivariable function. Differentiation is an abstract formula described in calculus that doesn't map to any primitive functionality provided by processors. Gradient computation for cost function in neural network can be a very hard task since the

differentiation is made with respect to a number of variable that are in the order of magnitude of $10^6 - 10^8$.
Backpropagation algorithm give us an efficient method for weights update computation, overcoming the previous described limitations. It simply works by back-propagating, into every hidden layer, the error produced in output by the network.

2.1.4 Overfitting

Neural networks are models that try to approximate non-linear function that otherwise could be difficult to represent, for this reason they suffer several problems. One on the most relevant problem in machine learning field is *Data Overfitting*, in particular important in Supervised Learning.
Overfitting can appear when when we lack of good amount of trained data, it is a generalization problem since the model perform well on input that are close to the object contained into the training set, but returning poor results for object that are different from those in training set.

2.1.5 Neural Network optimization

Since the introduction of MLPs, a great number of optimization has been developed in order to maximize the performance while maintaining the same basic ideas. There exists a lot of possible strategies, but they can be classified as strategies that focus on the training time reduction and solutions that try to maximize the output accuracy. Below are listed some of the most used:

1. *Batch Gradient Descent*
2. *Mini-Batch Gradient Descent*
3. *Local Response Normalization*

4. *Momentum*

5. *Weights quantization*

6. *Binarization*

2.1.6 Other implementations

Although it is a poor model and bypass the vast majority of the physical phenomena that occur into the brain, it has revealed extremely high performance compared to more complex solutions, becoming a de-facto standard in many domains.
In recent years has been proposed different new procedures that are closer to human brain; one example are Spiking Neural Networks, a class of algorithm that mimic the signal propagation incorporating the concept of time. For this reason sometimes they are referred as the 3rd Generation of Artificial Neural Networks for machine learning. The most important problem behind this models is the difficulty to find a good learning algorithm that take into account all this feature.

2.2 Convolutional Neural Networks

This is the state-of-the-art solution, in terms of accuracy, in large set of Computer Vision tasks where the aim is to classify one or more object contained into a single 2D image. Nowadays, Image classification and object detection application widely implement CNNs, their success is due to the exceptional results, with performance that can be compared, or in same case outperforming the human accuracy.
One of the most important limitation behind the previously adopted approach was the difficulty for classificator to ensure shift, scale and distortion invariance of the input image. This challenge was solved with image pre-processing with the drawback that this solution requires great effort for each image to be classified. CNNs use a totally different approach, in fact they learn features from sets of labeled

images, called *training set*, in order to classify all the possible inputs.

The structure behind a Convolutional neural networks is substantially similar to ordinary neural networks. All the neurons are grouped into layers that are responsible to extract informative feature from input, but unlike the last ones they exploit different ideas that allow to constraint layer shape in a smarter way. Although a simple neural network can be used to perform classification on 2D image, this is an impractical choice resulting into a very high number of parameters to be learned, with negative impact on training time and memory requirement, even for small images. A picture of 256×256 pixels RGB need a input layer of about 300000 neuron ,that can become millions in hidden layers.

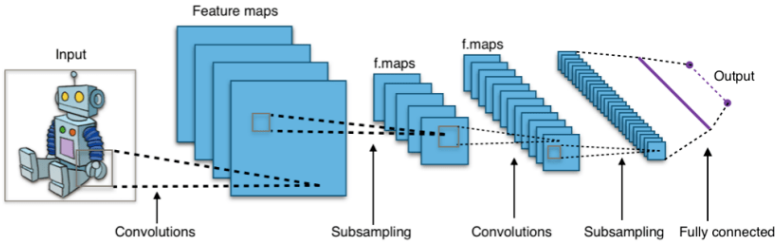

Figure 2.4. CNN architecture

Once again, as done previously, it is possible to borrow concept from biology in order to solve this issue. In particular studies done on cat's visual cortex [2] demonstrate that only a small region of neurons are connected. The introduction of these properties into neural network area is addressed by Kunihiko Fukushima [3], with the first CNN model, and then improved by Yann Lecun [4] in 1998 with the first trainable implementation of a CNN, LeNet5. The key properties exposed in work are: local receptive fields, shared weights and sub-sampling. This three properties led to a great decrease of the number of parameters to be learned since every neuron is no more connected with all previous unit, but only to a small region.

2.2.1 Layers

Unlike the basic form of neural network, there exist a family of layers that can be chosen to compose a Convolutional Neural Network architecture. Every type of layer receive an input image, called *Input Feature Map*, while in output produce *Output Feature Map*. They are represented in Tensor form, with dimension $(c \times h \times w)$, respectively the number of channels, height and width.

The most important type of layer is the convolutional one, it is the building block of the entire model. It does most of the computational heavy lifting, has been reported that 90% of the operation that are performed into one cycle of inference happen this kind of layers. CONV layer consists in an array of learnable weights, grouped into filters. Every filter, or kernel, is small spatially, but extends through the full depth of the input volume. Typically a filter can have size 5×5 or 3×3. During convolution, filters are applied to input receptive fields by means of a sliding window across the width and the length. Dot operations are performed between the entries of the filter and the input at any position. This mechanism produce a 2D activation map, each unit represent the value result from the multiplication between the sliding window at a fixed position and the filter.

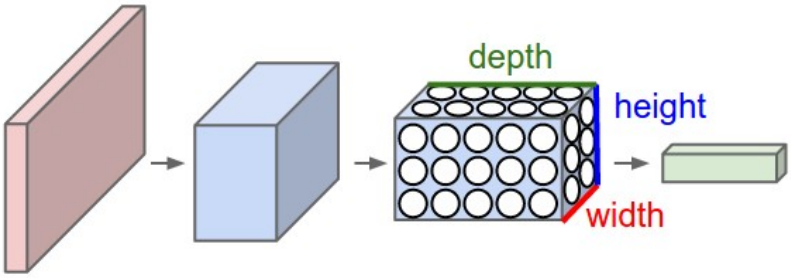

Figure 2.5. CNN convolutional layer shape

The hyper-parameters that control the convolutional layer shape are **depth**,

stride and **padding**. Depth corresponds to the number of filters that should be applied, each one learn to respond only to a particular feature; stride indicate the number of pixel that are used for the window sliding: slide 1 correspond to a move of 1 pixel at the time and so on. Finally we have padding, can be used to specify the number of pixel set to 0 added around the border.

Fully connected layer, unlike CONV layer, has connections with all the previous neurons. They are used before the output layer, connecting the convolutional pipeline that compute visual feature extraction, in order to assign an output label. Usually the most of the weights used in a CNN belong to this type of layer.

Pooling layer allows to down-sample the input representation by reducing the dimensionality. It is used either to the reduce the computational cost and to produce a more abstract form by summarizing the presence of patch contained into a feature map. There are two main implementation for this kind of layer: Max Pooling and Average Pooling Layers. Both the solutions take an input feature map applying a down-sample window of dimensions $k \times k$. The first one takes the max value, the second one make an average of the values contained into the window.

2.2.2 CNN architectures

Up to now researchers have produced several network topology in order to reach good results in terms of accuracy produced during the inference phase. We present below a list of the most important architecture proposed:

1. **Lenet5**: It was the first solution proposed for image classification, in particular has been developed to recognize hand-written digit, it is composed by 7 layer.

2. **AlexNet**: Introduced in 2012, it was the first solution that implement CNN in order to perform Image Classification for the most important challenge in Computer Vision: ILSVRC.

3. **VGGNet**: It consists of 16 convolutional layers and is very appealing because of its very uniform architecture.

4. **GoogleNet**: developed by Google it achieved a top-5 error rate of 6.67%. This was very close to human level performance which the organisers of the challenge were now forced to evaluate.

5. **ResNet**: At last, the so-called Residual Neural Network (ResNet) by Kaiming He et al introduced a novel architecture with "skip connections" and features heavy batch normalization.

2.3 Fault Injection

In literature we can find different techniques and methodologies that can be used to test the quality of a system. Those technique are aimed to test the reliability of a system, are widely used in both hardware and software testing and ensure the correct behavior to the final user. Performing an exploration about the fault tolerance is a good idea, but for a subset of applicative scenarios where hardware platform and application can be deployed, this process is mandatory. Safety-critical application are regulated by many international standards that defines a set of rules and metrics that the developers must take into account during their job. For instance, ISO 26262 is one of those. It has been created in order to define the test to be performed and a set of rules for the automotive scenarios, in order to avoid fatalities due to both hardware and software failures.
One of the most used technique used to assess the dependability of a system is the fault injection technique. At first it was used only in hardware, but has found wide

application in software testing.
In the rest of this section we will introduce the main technique used in both hardware and software testing, aiming to explain the techniques upon we relied for the development of this work.

2.3.1 Software testing

Software testing is one of the phases of the project development. It is an investigation done to ensure the quality of the software, aiming to find bugs that could be insert during the design or the development phase. The aim of every test is to maximize the coverage, i.e. the measure that describe the portion of code executed during test. Obviously, it doesn't exist technique that allow to achieve 100% of coverage, in particular for complex software, but there are some techniques that can be exploited to cover a subset of all the software components.
One of the most used technique used in software testing is the *fault injection*. It is very easy to use because ti require the a simple insertion of faults into the source code. The idea behind this technique is to explicitly follow error handling path that otherwise might not be followed.
The main idea behind this kind of techniques is to inject fault into the system in order to force a wrong behaviour that otherwise cannot be executed. Actually there are two different approach that can be used:

- *Compile-time injection*
- *Run-time injection*

Compile time fault injection is the simplest type of fault injection because allows the injection of fault at compile time, by changing the code of the software module under test. One of the most widespread is called Mutation test, it changes code statements to insert data or operation perturbations.
Run time fault injection is more complex since requires a software trigger that

explicitly inject fault into the source code. However it is more powerful, compared with compile time injection, since allows to simulate either hardware and software faults. For instance can be simulated corruption of RAM, CPU registers and I/O map.

2.3.2 Hardware testing

In VLSI, during the last years has been developed different methodologies and tools that go towards the system testing. Unlike software, testing process involves more steps to be performed since we must face physical limitation.
One of the more widespread technique used by integrated circuits designer is *Design for Testability* (DFT), that add testability feature to the design. It allows to manage in a easier way all the test performed during the several steps of the hardware manufacturing flows.
However, there exists other techniques that can be used and doesn't afflict the original design. One of those is the fault injection. It works by assessing the fault tolerance of a system by the deliberate insertion of faults, they can be classified into [5]:

- *Hardware fault injection*
- *Simulation-based fault injection*
- *Emulation-based fault injection*

The first approach relies to special hardware to introduce faults. More in details, it uses external physical sources to introduce perturbation into the system state. For instance, in *Radiation Testing* hardware components are hit intentionally exposed to ionic radiation in order to trigger transient faults.
In simulation based fault injection, the target system and the fault injector are simulated through a software process. With respect to the hardware simulation is

easier and not requires special hardware to be used. However it require additional time overhead since the entire system should be simulated in software.
With the Emulated fault injection, the fault injection is performed into re-programmable platforms, like FPGA. This approach is a trade-off between the first and the second approach since the system is tested in hardware without the necessity to use special hardware.

Chapter 3

Proposed approach

In this chapter we will explain in details how we dealt with the problem of reliability analysis of a neural network, discussing about pros and cons of the approach that we selected. Our primary goal is to perform a reliability analysis, bounded only to convolutional neural networks, that verifies the fault tolerance of these models during the inference phase, highlighting when the result deviates from the correct one.
To achieve this objective we propose a simulation-based fault injection, assessing the behavior of CNN topology by the deliberate insertion of faults into the system to determine its response. The correctness of the network is assessed with a comparison between the results computed by the model subject of the fault injection and a reference model that computes correct results.

In literature there exists different methodologies that can be used to place faults into a system, they can be briefly classified as: hardware fault injection, simulation-based fault injection and emulation-based fault injection. Obviously, the methodology selection process should be made carefully, because a correct decision made in this phase determines the achievement of objectives. We have chosen to follow the second approach since it's the easiest and more convenient way. The other

approaches widely rely on special hardware that try to access the subject system, requiring a non-negligible economic cost for the equipment purchasing. With a fault injection made using as target system a software model, we can also easily track its internal state, making successive analysis more easier to be understood. The biggest shortcoming is the impossibility to simulate every single system mechanism, indeed a complete simulation may employ too many hours before the actual results.

Previous works conducted on this topic focus the characterization of output results in term of *Silent Data Corruption*, taking into account only fixed modules that compose the network and executing the simulation only under soft-error constraint. In [**?**]uthors faced a different analysis using a permanent fault model, by injecting bit flips on a subset of weights associated to a defined layer.

Since the very earlier phases of this work, our aim was to use commercially available tools that can be exploited to perform this kind of simulation. During the time spent for this work we didn't find a reference implementation of CNN hardware accelerator to perform a fault-injection simulation at *Register Transfer Level*. For this reason we focused our effort upon a self-made software-level fault injector, aiming to imitate the same effect of a data corruption insert at *Register Transfer Level* or *Architectural level*. In next section of this chapter will be explained the fault injector expected work and how to simulation should be performed for the experiment success.

3.1 DNN fault injector: adopted strategy

As stated in the previous paragraph we decided to adopt a full software level fault injection strategy. This means that we performed our analysis at the highest level of abstraction of the neural network application stack, bypassing all the possible

hardware structure where the application runs [6]. Our aim, using this approach, is to perform a reliability analysis that is independent from the error source, either software or hardware. This idea is consequence of the key role that the software layer plays into the propagation of faults raised at hardware layer. When physical defects or soft errors afflict hardware components like RAM, cache and registers, the faults could be propagated or could be masked. The latter behavior is triggered in particular if the system has been designed taking in mind some types of protection techniques.

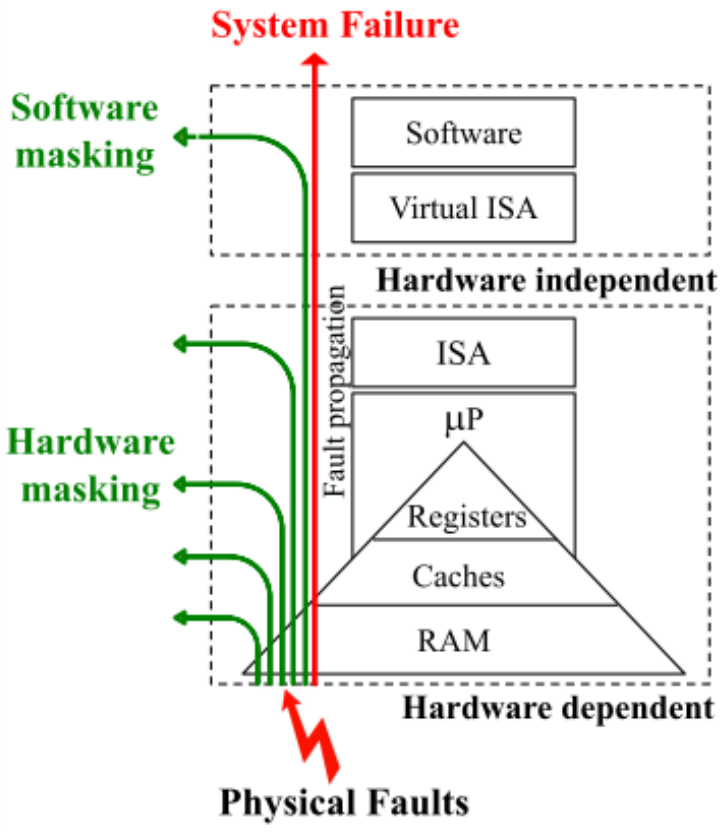

Figure 3.1. Hardware fault propagation

With fault injection we mean an iterative technique that inject one fault at a

time into a system, evaluating at the same time the results corresponding to some input stimulus. Convolutional neural network are very complex system, today's there exists many architecture that are accounted to be composed by many thousand of neurons, grouped in layers. Therefore performing an exhaustive exploration about all the erroneous conditions is not feasible. Moreover, for every fault injected into the network topology should be performed two separate inference computations: one to detect the reference value, another one to compute the faulty value. To overcome this issue we decided to bound our experimentation only to a subset of all the operations that are scheduled into a single cycle of inference, taking into account only those that could be source of unreliability.

As stated in previous chapter, inside a single layer, the weights values and the previous layer's activations are multiplied and summed-up in order to compute neuron activation. This is a very delicate step because one error raised during this operation lead to completely different results. Moreover, since DNNs have a hierarchical structure, these errors can heavily afflict successive computation, leading to a wrong classification of the input image.
At this point, two possible question are: what is the DNN fault tolerance degree, if we inject permanent fault into the weights? What is the DNN fault tolerance degree, if we inject permanent faults into the multiply-and-accumulate (MAC) operation? We try to answer those questions by performing several fault injections, bounding the analysis only to the effects of failures that could raise into those two scenarios that can be summarized into:

- *Permanent fault injection into weights*
- *Permanent fault injection into resulting value of multiplication*

Initially we have performed a permanent fault injection into the weights to have a preliminary evaluation of the effects of this faults. Afterwards we have decided

to enter into details, performing a more specific analysis by assessing the bahaviour under permanent fault at the multiplication output.
With this type of solution we detach the fault simulation from any particular hardware implementation. Actually it is not completely true, because there are many other factor that affect the computation in hardware, like scheduling, pipeline etc.

In order to reach a statistically relevant analysis of the results, the fault list to be fed to the fault injector has been statically generated. The fault to be injected has been selected randomly using a uniform error distribution. This has lead to a better evaluation of the possible system failures even without performing a complete analysis of all the possible cases.

3.1.1 Fault injection environment

Performing a fault injection simulation, either into a software or hardware system, is not a straightforward operation: it requires a deep knowledge of its properties and a clear understanding of its behaviour. Moreover, a correct configuration of the actors that take part into the simulation can be a burden, especially for those system regulated by many thousand parameters.
In order to minimize the potential sources of errors, during the fault simulation development we decided to take inspiration from other fault simulation tools, both industrial and academic, borrowing the high level structure of the components involved. One of the most interesting solution is explained into [7]. It has been selected as starting point for our work, implementing in software the fault injector.

So far has been done a high-level description of the simulator intended work, its configuration and what are the results that we expect to see. Now it's time to explain how the environment has been developed, introducing all the software modules involved during the simulation high-level description. We will talk about

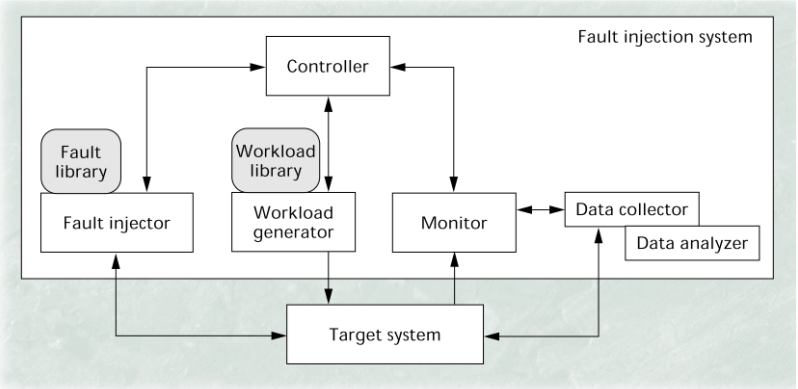

Figure 3.2. Fault injection environment

environment configuration and each module involved during our simulations. The simulator architecture can be synthesized by describing the following components:

- *Fault injector*
- *Fault library*
- *Monitor*
- *Data collector*
- *Data analyzer*

This implementation allows to control in every moment the fault injector job. This work is done by the monitor, it continuously supervise the entire simulation process by reading the information from the fault library. The fault library, or fault list, is a database that contains the information about the faults to be injected into the target system.

Despite every component has great importance, fault injector requires a more depth explanation since it is the "heart" of the fault simulation. This module, developed at software level, aims to place permanent data corruption at bit level

granularity, simulating system failure that can be addressed to glitches in hardware components. One property that we have ensured is the totally unawareness of the injector about the numerical representation selected at software level. It simply takes the fault from the list of fault and inject at run-time the data corruption into the variables that contains the results of computation.

The separation between the fault list and the fault injection allows to create a more scalable architecture. In fact a fault injector should be general enough in order to handle all the possible fault model and fault location. Otherwise, it results constrained to a fixed number of fault location,making subsequent modifications more difficult.
Monitor is component that handle the entire simulation cycle, selecting the examples to be fed into the system and the fault to be injected. While the monitor ensure the the correct behaviour of the fault injector, the Data collector read the target model output and store those value into several external files.
Data analyzer is the tool that is in charge to elaborate the information produced the simulator.

3.1.2 Simulation life-cycle phases

Once that our fault injector simulator has been developed, its possible to shift the attention to the next step. The fault injection simulation represents the core of the entire work, allowing to perform our fault tolerance evaluation on Convolutional Neural Network by means of fault injected into the model.
To better explain our simulation, we divided the simulation life-cycle in three phases:

- *Simulation set-up*
- *Fault injection*

- *Result analysis*

Before starting the actual fault injection, it's necessary to set-up the environment with a careful configuration of all the module that are involved during the simulation. During this phase should be chosen a CNN architecture model that describe the target neural network topology; a test set that contains the examples supplied to the model, and finally a fault list. The last configuration file is an array of records that explicitly informs the injector about fault locations and fault model to be injected. Afterwards, the model is fully trained feeding a training set until a satisfactory level of accuracy is reached. Otherwise, it's possible to set the network hyper-parameters by using pre-trained weights.

At this point is necessary to do a just little bit clarification about the meaning of test set in this context. Actually, for the experiments perspective there are two different test set, one is used to evaluate trained model skills, the other one instead contains the source input that fed the system under test during simulation. In order to perform a comparison between fault free and fault-injection results, a golden run is performed feeding the test set without injecting any fault, with the purpose of collecting values that come out from system under ideal conditions.

After the simulation set-up phase is completed, the actual fault injection starts by iterating over the entries contained into the fault list. This means that the simulator injects one fault at a time into the network and evaluates the results corresponding to the input stimulus applied to the model. The input fed to the CNN model are taken from a set of examples, which constitute the test set. Since our evaluation has been bounded to Convolutional Neural Network, the test should be created starting from data set of images already available on-line.

3.1.3 Fault tolerance metrics

A key point for a correctly experimentation is the choice of the metric to be employed. Choosing the correct metrics help to validate hypotheses and ensure that we make progress towards our goal. In our case, we don't need to make of complex metrics, instead we use simple criteria that allows to have a clear idea about the fault propagation. We define the *Silent Data Corruption* (SDC) probability as the probability that a fault injected affects the application (or system) correct behaviour. For our convolutional neural network fault injector, the application of this criteria is not straightforward as it may appears, since there are several parameters that are produced in output.
Convolutional neural network, by the machine learning point of view, is a classificator model. This means that given a input it return a value that represent a category the data belongs to. Generally speaking, each time a convolutional neural network perform an inference, it assigns a confidence score to each category. This value represents the probability that the model's input belongs to the specified class.
For this reason we decided to use adapt SDC metric taking into account not also the classification value, but both classification value and confidence score. Consequently, we decided to use this new criteria, as specified in [8]:

- *SDC-1*: The top ranked element predicted by the DNN is different from the predicted value of the fault-free execution.
- *SDC-10%*: The confidence score of the predicted element varies by more than -10/+10% of its fault-free execution.
- *SDC-20%*: The confidence score of the predicted element varies by more than -20/+20% of its fault-free execution.

This metric allow to perform a more fine-grained detection of faulty behaviour. In the first experiment we used a slightly different nomenclature, in fact we used

the Masked, Critical SDC and Non Critical SDC. Anyway, the meaning is the same as the previous metrics: Masked means that the fault injected has been masked by the model, i.e. the result is the same of the golden run. Critical SDC means that the output performed with the fault injected is different from the golden run output. At the end, Non Critical SDC means that the output is the same, but the confidence score is above or below the maximum threshold.

Chapter 4

Case study

The purpose of this chapter is to show to reader a detailed report of the work done, carefully explaining either implementation and simulation details. In particular, in next sections we try to explain the following two points, that were introduced into the first chapter of this dissertation:

- *Convolutional neural network fault injector development*
- *Lenet5 fault injection experiment*

In the first part we will explain the overall structure of the simulator, starting from its conceptual design and the implementation made to achieve our fault injection results. Instead, in the second part we will talk about the actual simulation performed using as target Lenet5, a very popular CNN topology developed by Yann LeCunn, explaining all the decisions made during the simulation setup as well the optimization performed in order to speed-up the process.
Afterwards, a reliability analysis was performed using Lenet5 as target topology, injecting either stuck-at 1 and stuck-at 0, before into the input weights, after into the output of the multiplicator. In next sections will be explained how to fault injection has been though, highlighting the implementation details and all the built-in modules offered by Darknet in order to perform a classification of images.

4.1 Darknet

Since our aiming is to perform a software fault-injection into a neural network model, we selected as target system for the simulation a framework named Darknet. This tool provides a software platform able to perform end-to-end processing for a generic DNN topology. Our strategy has been focused towards the customization of this tool to encompass our software fault injector, providing a full environment for simulation-based fault injection.

As stated in proposed approach chapter, our goal is not to assess fault tolerance of every DNN type. Instead, we focus our attention only to the evaluation of Convolutional Neural Network. For this reason, Darknet source code has been modified to perform fault injection only into the most relevant types of layers: Convolutional and Fully Connected. Other types of layer as well other functional units are left untouched. Anyway, the software fault injector has been developed in order to be scalable enough, ensuring the right compatibility with future work related to different types of fault model and architecture types.

As stated earlier, we used as starting point for our simulator Darknet. This framework has been developed by Joseph Redmond for academic purpose, aiming to propose a training and inference platform easy to use. Its strengths is undoubtedly the extreme flexibility: a complete prototyping of a user defined topology, starting from model definition, can be done in few steps. The core computational unit has been written in C language, making it very easy for being studied, understood and of course extended. Moreover, one interesting aspect is that has been developed to exploit GPU parallelism, for both inference and training, by using CUDA library. The entire source code has been released into a open source repository maintained by the author.

Since it has been developed as a framework, it allows to develop every kind of network topology by providing a full eco-system of neural network functionalities. It can be thought as a platform were each user can create her own topology simply by specifying, into a set of distinct configuration files, the sequence of layers to be included and the hyper-parameter to be used, if any. The framework is in charge to parse the configuration files, written in human readable format, and instantiate the correct topology by interconnecting the instance of each layer type.

4.2 Darknet fault simulator

The simulator heavily rely on the functionality offered by the Darknet framework. This not means that the original structure has been totally changed, instead we created the conditions to perform a fault injection into DNN target topology adding functionality that before were not include. Specifically the original structure has been left untouched, ensuring the same operation that were offered by the framework.
Our strategy behind the implementation of the simulator relies on the observation of the framework execution. We can split the framework life-cycle into two phases:

- *neural network building context*
- *neural network execution*

When we invoke the Darknet tool, the first thing that is done is the creation of a building context. It represent the starting point for all the successive evaluation on the network. Within this context, it is possible to use different tools to exploit the building blocks offered by the framework, allowing users to build its own architecture. This context can be thought as a skeleton that holds up all the successive computations made with DNNs.
After the DNN deployment into the Darknet context, its entire life-cycle is handled

by framework, offering to user all the possible actions to be performed with DNN, such as training, classification, testing. Despite each neural network deployed lives into the Darknet process, its behaviour is totally independent from the rest of the platform.

Its overall structure is composed by a front-end module, called *darknet.c* that controls the entire life-cycle of a neural network, acting as entry-point of all the successive computation. It's the main interface between the framework and users, offering all the operations to be performed on neural networks. During the simulator development, it has been the first component to be modified, in order to adapt Darknet to a fault injection simulation. This module has been extended, offering the possibility to perform a simulation simply by switching to "fault simulation" from command line.
Afterwards, into the source code we added a new module, named *fault_simualator.c*. This has been associated to the main *darknet.c* in order to be launched every time the user selects "fault simulation" from the contextual menu.
Into *fault-simulator.c* was created several new functions and data structures, the most important one is the *perform-simulation*.
Obviously the framework need the target network for the fault injection. For this reason in this context we used the same functionalities used for training and inference to deploy a new CNN topology.
It receives from user test set, fault list and the output directory where the simulation results should be placed, as well all the others configuration files that allows to build-up a neural network. It is composed by two nested while loop, the external one loop on the test set, the internal one instead iterate over the entries of the fault list. In this way the simulator changes the behavior of the network topology by injecting one fault a time, evaluating at the same time the output of the model.

The value resulting from this computation are then stored into the file system using the functionalities offered by the *data_collector.c* module. It is in charge to create the resulting file adding all the information returned by the neural network under test: confidence score and classification label.

4.2.1 Fault injector development

The fault injector is the key component behind the simulator. It was developed starting from the *convolutional_layer.c* and *fc_layer.c* modules of Darknet. In particular those files contains all the functions and data structure that are used by the parser to deploy a neural network. The most important functions are the backward and forward. The former is used only during training phase, the latter instead is used only during the inference phase.
Daknet, as many other framework, is not capable to perform a convolution between weights and input feature maps in a easy way. Indeed, to perform convolution and weights multiplication, it relies on some algebraic functions that allows to handle this operations in the form of multiplication between matrix. The core module used by the convolutional and fully connected layers is called *gemm.c*, it supplies a set of functions that performs multiplication in matrix form.
We added a new module called *gemm_faulty.c* that produces the same results of the *gemm.c*, implementing data corruption. Generally speaking the injection works by selecting a subset of all the multiplications, in agreement with the information specified by the fault list, acting into the result stored in the variables.
In particular, we didn't used any timer trigger since we didn't know when it was possible to inject the fault into the variable, for this reason we modified the matrix multiplication algorithm implemented in order to place data corruption at a fixed number of multiplications. We used a counter that count the number loops of the matrix multiplication. With this simple mechanism we bypassed more complex implementation achieving the same results.

4.3 Lenet5 fault injection experiment

The next step is to discuss the simulation carried out, describing in details all the configurations made. We recall that during our experiments we performed two different simulation. We performed a preliminary simulation, intended to evaluate the neural network resiliency by injecting permanent fault into the input weights. Afterwards, a simulation was performed with the injection of stuck-at 1 and stuck-at 0 into the result of the multiplicator.

All the simulations has been performed using a 16 cores CPU, disabling CUDA library for GPU inference mode. To further optimize the time required to perform a simulation, we decided to use a better strategy aimed to exploit the full power of our CPU. Instead of a single large simulation carried out over the entire test set, we performed several smaller simulations using disjoint subset of the original test set. Those simulations have been scheduled in parallel, earning a remarkable speed-up of result computation.

4.3.1 Simulation set-up

During this phase, the fault simulation is prepared by deploying all modules necessary for the experiment. The first thing to be done, in chronological order, is the CNN set-up. The neural network is created starting from the high-level description and a set of already trained hyper-parameters. Darknet parser read the description and interconnect the different instances of the layer between them, while loading in every convolutional and fully-connected layer the hyper-parameters specified in an external file. In order to speed up the simulation we decided to use a pre-trained set of weights, available on-line from [11].
Test set and fault list are loaded into the environment, simply b passing as argument those files to the Darknet executable. The first file contains the sequence

of examples to be provided to the simulator, while the second contains a list of entries which specify the fault model and the fault location. The fault list should be generated before the actual begin of the simulation.

Another important aspect that must not be overlooked in any way is the golden run. When we deal with simulation-based fault injection, we must compare the value resulting from the fault injection with a set of reference value. Those reference value should be generated before the actual fault injection, without any fault injected into the model. Obviously the input to be fed to the model must be the same of those used for the fault injection, otherwise it's not possible to compare the results.

4.3.2 Lenet5 topology

Once we have introduced the Darknet framework, it's necessary to talk about the topology used in this case study. For what concerns our experiment, the fault tolerance evaluation of a CNN has been made using a Lenet5 topology.
From its introduction, has been proposed many new model, more deeper and powerful that Lenet5. We have chosen it because it is composed by only 5 layers, 3 convolutional layers and 2 fully connected. In the table below we show the number of parameters that compose each layer of the architecture.

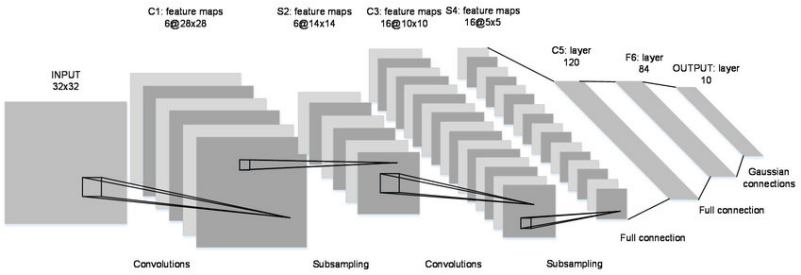

Figure 4.1. Lenet5 architecture

The input layer receive 32×32 pixel input image of hand-written character. Together with the fact that the topology is composed by only 5 layers, This property, together with the previous one, has led us to this topology. In this way it was much simpler the evaluation of the neural network behavior.

4.3.3 Test set

When we are dealing with simulation-based fault injection, one of the key point is the test set selection. The difference between a good test set and a bad one is the capacity to stimulate corner case, resulting in a better exploration of model under test.

In VLSI testing there exist techniques that automatically generate a test set, according to the digital design, in order to speed-up and reduce the effort required by humans. However, in our case, since the model is a convolutional neural network, it's not very easy to generate a test set. The first trouble resides to the fact that the input examples that must passed to the simulator are composed by 2D images, so we cannot manually selects input cases that can stimulate corner case. For this reason we have employed a subset of images contained in MNIST, 70000 pictures that represent handwritten digits, from 0 to 9.

4.4 Result analysis

The fault injector simulator produce a huge quantity of files, each one describe the result computed by the target system during the fault injection. Our simulations produced more than 1000 files, with a total size of 6,4 GB. Since a manual evaluation of those results is not feasible, we developed several scripts for a automatic evaluation of the information stored into the files.

Our script has been written in Python, using a data analysis library named *Pandas*.

Thanks to this library we joined all the information contained into several files into a single data structure. In this way it was much easier to compare the faulty results with the golden prediction.

Chapter 5

Experimental results

The aim of this chapter is to explain the main results achieved by our fault injection campaign performed using our Convolutional Neural Network injector. We will enter into the details by describing the results obtained, at the same time we will try to give a characterization about the fault tolerance of the Lenet5 CNN topology. The first section shows the main results achieved by our preliminary analysis, conducted with permanent fault placed into the input weights. In the second section we will show the results achieved trying to carry out a more complex simulation, which approaches the real behavior of a hardware platform. We injected both stuck-at 1 and stuck-at 0 at the output of the multiplication between weight and input feature map in both Convolutional and Fully Connected layer.

5.1 Weight value distribution

The CNN behaviour, like all the implementation of DNN, is strictly related to the value assumed by theirs hyper-parameters. Therefore, before our results analysis is important to have clear ideas about the overall structure of the Lenet5 architecture, in order to draw the right conclusions.

The image below depict the distribution of the weights value used for our simulations. As we can see the values are centered around the value 0, with a variation between -0.2 - 0.2.

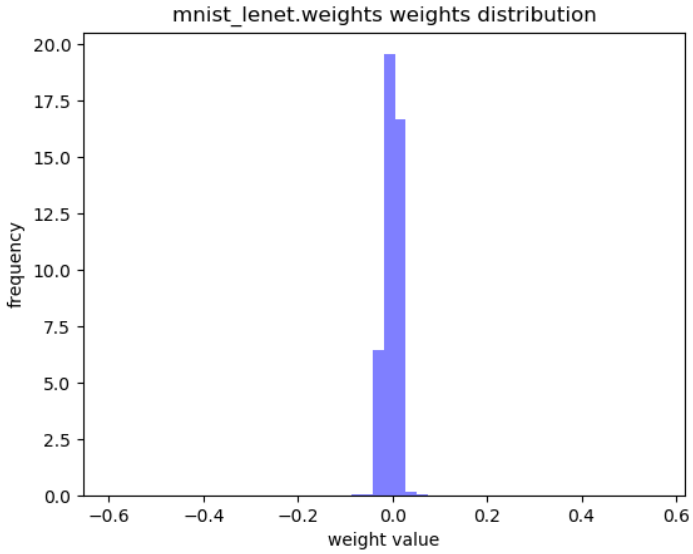

Figure 5.1. Weight value distribution

5.2 Weights corruption

Our preliminary analysis has been performed by randomly injecting stuck-at 1 and stuck-at 0 fault into the weight. The images below shows the result in terms of percentage of faults Masked, Non Critical SDC and Critical SDC. To have a clear idea about the behaviour of the network, we differentiate the fault injection charts per layer (from layer 0 to layer 3).

In those three charts, the first thing that can be noticed is that the percentage of Critical SDC is higher into the first 2 layers, i.e the convolutional layers. At the same time the percentage of Masked faults is higher in the last 2 layers, i.e.

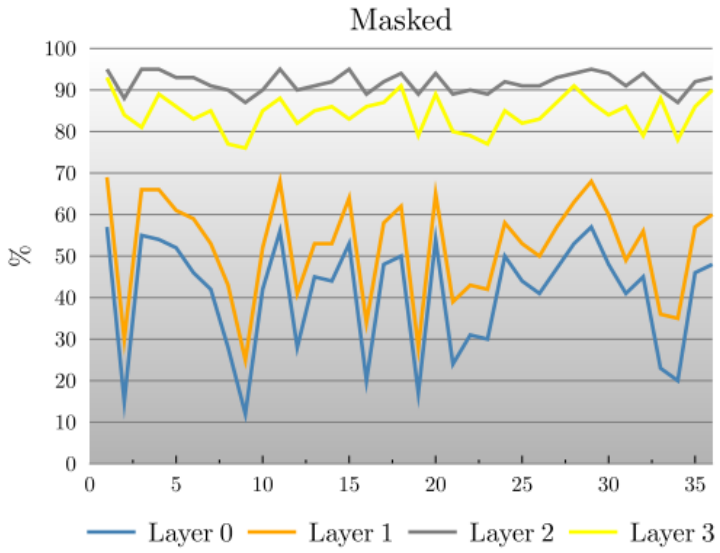

Figure 5.2. Masked faults

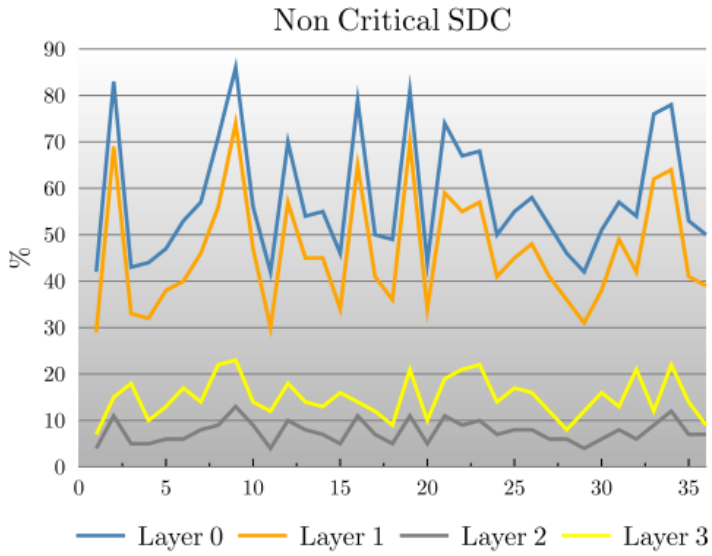

Figure 5.3. Non Critical SDC

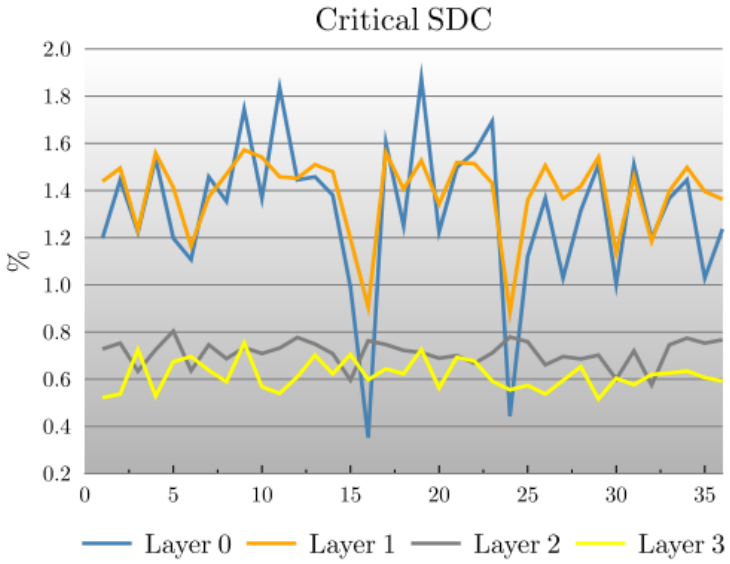

Figure 5.4. Critical SDC

the Fully Connected layers. Those results are quite interesting because shows a different trend with respect to the result obtained by soft error injection in [8]. Indeed, the convolutional layer are supposed to be more resilient to the presence of faults. This behavior is due to the fact that convolutional layers are in charge to extract the features from the input image, therefore permanent fault lead to a wrong elaboration of the information contained into the image.

Another important result highlighted from this analysis is the distribution of the most critical bits of the variable that store the weights. All the faults that lead to Critical SDC, belong to the exponential part (i.e. bit 30 down to bit 23) of the 32 bit floating point representation.

5.3 Multiplication corruption

In this second part we will show the results that we achieved during our analysis focused on the output of the multiplication between weights and input feature maps,

injecting permanent fault in both convolutional and fully connected layers.
This simulation is quite different from the previous one, indeed it has been computed by entering more into the details about the operation that happens into the hardware platform. The metric used in this analysis is very similar to the previous one. Moreover, we decide to perform an analysis to show the most predicted label produced by the model under faulty conditions.

5.3.1 Stuck-at 0 fault

We begin our result evaluation starting by the least critical fault model, stuck-at 0. The chart below shows that the probability of having a SDC is substantially the same for all the fault location, with the SDC-1 probability into the range of $0.0075\% - 0.02\%$.

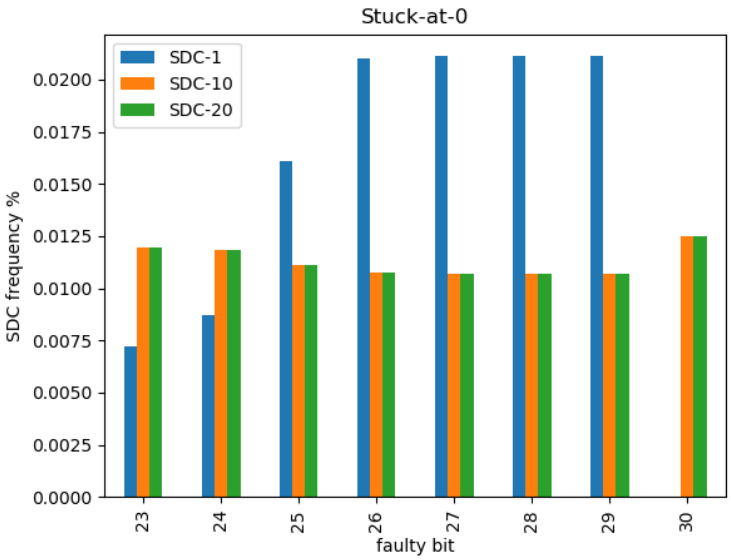

Figure 5.5. SDC frequency for stuck-at 0 faults

SDC-10% and SDC-20% are practically the same in every fault location, with slight variation of -0.0001% between the bit 26 and bit 29. Those results are in line with our hypothesis, since a the weights distribution says that the weight are

centered around the value 0. This means that the higher bits of the exponential part of the floating point are set to 0. For this reason a stuck-at 0 injected doesn't affects the CNN behaviour.

5.3.2 Stuck-at 1 fault

After the analysis of stuck-at 0 results, we can discuss the results that has been computed by the stuck-at 1 injection. Compared with the previous results, this chart show a great variation of SDC between bit 30 and 29, in particular if we inject stuck-at 1 into the bit 30 of the variable there is the probability of 12,3% to have a misclassification in output.

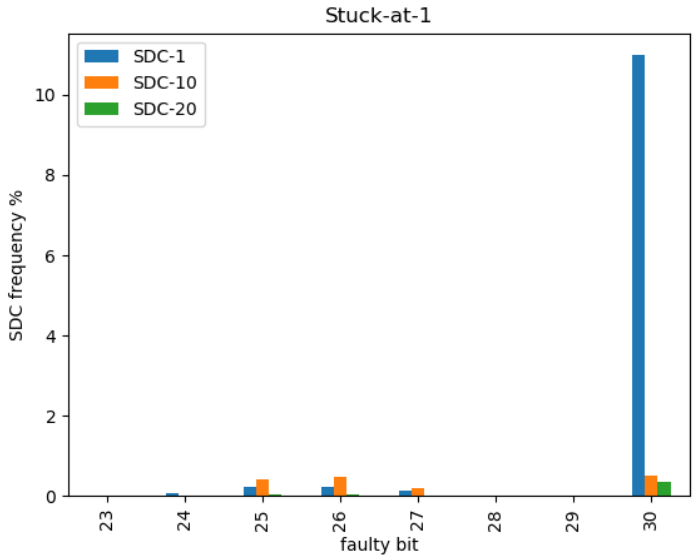

Figure 5.6. SDC frequency for stuck-at 1 faults

Within this section we also introduce the chart of the most predicted labels. The image below depicts the results of the analysis conducted in order to show the most predicted label after the injection of stuck-at 1.

We noticed that the more than the 90% of the stuck-at 1 result in a label

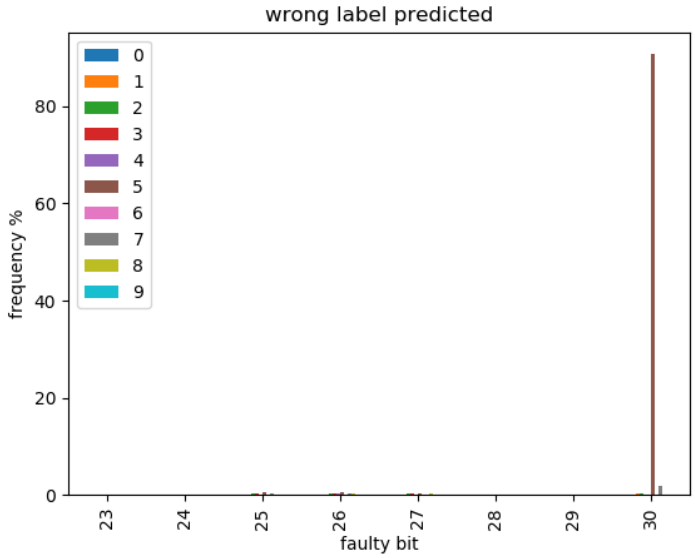

Figure 5.7. Frequency of faulty predicted labels

predicted of 5. After an investigation we found that the when we inject stuck-at 1 into the variable containing the activation it saturate to the maximum value. However we didn't find any explanation to this result since we didn't get the chance to use visualization tools for the hidden layers.

Bibliography

[1] W. S. McCulloch and W. Pitts, "Neurocomputing: Foundations of research," ch. A Logical Calculus of the Ideas Immanent in Nervous Activity, pp. 15–27, Cambridge, MA, USA: MIT Press, 1988.

[2] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *Journal of Physiology (London)*, vol. 160, pp. 106–154, 1962.

[3] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, 1980.

[4] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, pp. 2278–2324, 1998.

[5] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pp. 1–6, May 2014.

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

[7] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, pp. 75–82, Apr. 1997.

[8] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and

S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, (New York, NY, USA), pp. 8:1–8:12, ACM, 2017.