POLYTECHNIC OF TURIN

MASTER's Degree in ELECTRONIC OF ELECTRONIC SYSTEMS

MASTER's Degree Thesis

Robustness and Sensitivity Assessment of Deep Neural Networks

Supervisors

Prof. Edgar Ernesto SANCHEZ

Ph.D. Annachiara RUOSPO

Candidate

Angelo BALAARA

October 2020

Acknowledgements

I would like to express my gratitude towards those people that help me with the thesis project. I want to thank the professor Ernesto Sanchez since, he believed in me giving me full confidence. I want to also thank Annachiara Ruospo since, she spent time and effort helping me whenever I needed. Overall I want to thanks the above mention person because, they shared with me their knowledge and their human side.

I want to dedicated this thesis to my family and all the people that helped me out to be the person I am right now and carry me over this tough years.
A special mention goes to my closest friends.

Abstract

Nowadays, Deep Neural Networks (DNNs) are employed in many fields to perform different tasks, such as autonomous driving, always listening, augmented reality, computer vision and many others. Some of the listed tasks, as the autonomous driving, are safety-critical. Aware of that, ensuring the reliability of this technology is very important since, an error of assessment could endanger human lives.

DNNs are known to be intrinsically error resilient [1] since, their topology exploits data redundancy. Therefore, even if an error occurs the DNNs might be still capable to correctly predict the result. However, this may change whenever the DNNs are implemented in Hardware (HW) for the reason that, the target HW have limited processing resources in which multiple neurons are mapped and potentially be corruptible by a single fault.

The aim of this master thesis is to study novels Hardware Description Level (HDL) fault injectors able to reduce the expensive time cost of the state-of-art HDL fault injectors. The attention was drawn onto permanent and transient fault models. Indeed, the thesis will provide a pipelined fault injector for permanent fault and an optimized fault injector for transient fault. Both fault injectors were tested in fault injections campaign to asses their effectiveness. Such frameworks make feasible the reliability analysis at the HW level of systems based on neural network, reducing the required simulation time of at least 60%.

Table of Contents

List of Tables

List of Figures

Acronyms

AI artificial intelligence

NN Neural Network

DNN Deep Neural Network

CNN Convolutional Neural Network

FCNN Fully connected Neural Network

HW Hardware

SW Software

PULP Parallel Ultra Low Power

ReLU Rectifired Linear units

PE Processing Element

RTL Register Transfer Level

HDL Hardware Description LeveL

ANN Artificial Neural Network

IFM Input Feature Map

OFM Output Feature Map

SIMD Single Instruction Multiple Data

QNN Quantized Neural Network

FIU Fault Injector Unit

FCU fault Controller Unit

DVFS Dynamic Voltage Frequency Scaling

DMA Direct Memory Unit

MCU Micro Controlle Unit

FPGA field programmable gate array

SoC System on Chip

MAC Multiply and Accumulate

ALU Aritmetic Logic Unit

DOTP Dot Product

Introduction

The focal point of the following work is about an important aspect of the Deep Neural Network technology architecture as their reliability. In particular, the attention is drawn to the cohesion between the abstract software dimension with the real hardware world. From a high perspective DNNs are built by many layers of neurons connected with each other allowing the transmission of the computed information between a neuron to the others [2]. In such scheme, each neuron is independent and unique, meaning that if a neuron is damaged the DNN might be still capable to correctly predict the result and mask the error, thanks to its topology that exploits data redundancy. However, the issue is not that simple whenever the DNN is mapped on a hardware equipment. The target hardware may be multiple, FPGAs, MCUs, GPUs, ASICs and many others. Even if, the target hardware are different in terms of structures philosophy, performances and target applications, they have in common the limited processing element (PE)resources that leads to a different analysis of the DNNs architecture. Under those circumstances the neurons are no longer independent and unique. A cluster of neurons are evaluated by means of the same PE, for this reason, it is highly unlikely to have a single or a random group of damaged neurons. As the trend of shrinking the semiconductor to dimension of the nanometer order, the probability of physical damage grows to the point that it cannot be neglected. Furthermore, other potential elements able to corrupt a DNNs executions are various such as, the aging, electromagnetic radiations and many others.

Hence, a tool able to assess the error resilience of the hardware and software co-design is becoming more and more important, also for the reason that this technology is increasingly used in critical field as the autonomous driving [3], [4]. This tool is a fault injector able to perform fault injections campaign with different approaches at different levels of abstraction, from the application level to the Register Transfer Level. The main differences between the abstraction levels are the required time, that is considerably greater for the level strictly bonded to the HW and the degrees of freedom of the injected faults. For instance, at the application level it is possible to corrupt weights, biases, input and output data. At lower level instead, it is possible to have a full control of the signals and bits, stacking those signals to ‘0’ or to ‘1’ permanently or transiently. As mentioned before, is fundamental to evaluate the dependability at the HW level since, it is more accurate. But, the time needed to perform fault injections campaign at RTL is considerably high. For this reason, the simulations are usually performed at software level.

The aim of the following thesis is to develop and test novels faults injectors frameworks for transient and permanent faults. Considering the different nature of the faults model, two different strategy has been used. For the permanent faults a pipelined fault injector. In this framework each layer is seen as a pipeline stage computing its own workload. Those stages need to be synchronized with the each other. Instead, for the transient fault, a multi-layer fault injector. In this particular case the framework is divided in different abstraction levels to take advantage of the higher speed of the software level. The intent is to make those frameworks optimize as much as possible in terms of simulation time, as described in Chapter 3. Additionally, they need to be capable of automatically injecting faults within the system and classifying the inferences outputs.

The rest of the thesis is organized as follows. The background knowledge are described in Chapter 1. The Chapter 2 presents the proposed framework, Chapter 3 describes the case study. The experimental results are provide in Chapter 4. Finally, Chapter 5 concludes the thesis by outlining some of the possible future research directions and some consideration arising from the experimental results.

Chapter 1

Background

1.1 Artificial Neural Networks

Artificial Neural Network(ANN) is the technology architecture on which the Artificial intelligence (AI) is based. The fundamental idea is to have an artificial model that mimics the human Neural Network (NN) behaviour preserving its design. Since we don't have yet the full comprehension of the NN, the artificial replication is a simplification but, still it has a great computation power. This technology allows us to have a powerful alternative way to compute particular tasks, such as the classification and regression problems.

NNs need to be trained, meaning that they have to learn from data minimizing a cost function which describes the distance between the input data and the output result. This specific task is a simple problem already investigated from the data analyst. Anyhow, there is a problem of those algorithms since, during the minimal research the algorithm can be stack at an unwanted local minimal lowering the maximal reachable accuracy of the NN itself [5]. Though, the target of the NN is to describe into a function the relationship between the input and the output.

NNs architecture is complex and composed by many components interconnected with each other. The main component is the neuron which is replicated multiple times. Each neuron has a series of input connections, dendrites, and an output, axon. The whole system, dendrite, axon and neuron mould a synapse as is shown in Figure 1.1

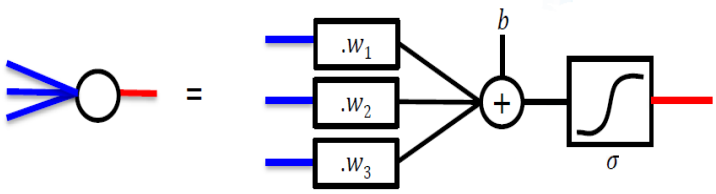


Figure 1.1: Synapse model

The neurons can be considered as a processing elements performing a certain function and the dendrites as the information stream [2]. All neurons can be distinguished by the weights in each dendrites, biases and the performed function. This functions are called activation functions and usually are in the range '0' to '1'. They can be linear or not. The most widely used is the ReLU function (Rectified Linear Unit), defined as the positive part of its argument [6], from an electronic prospective, it behaves like a diode.

NNs are made of layers of neurons interconnected with each other. A single layer of neurons can only represents simple linear weighted combinational input-output relationship. Concatenating, one after the other many layers, with different characteristic and functions, enable complex non linear input-output mapping performing complicated classification tasks.

Three types of NN can be done, Fully Connected Neural Network(FCNN), Convolutional Neural Network(CNN) and the mix of the two of them. The difference between the first two networks lie in the way the neurons of subsequent layers are connected with each other. In the FCNN the neurons of the previous layer are all the connected with each neurons of the following layer, instead, the CNN has only part of those connections. Additionally, this NNs topology have different performances in terms of train-ability and memory footprint.

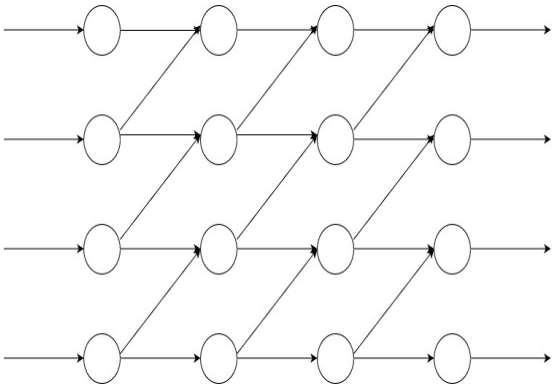


Figure 1.2: Convolutional Neural Network example

The NNs have an intrinsic issue originated by the feature extractor that has to be handcrafted. In this scenario, the feature extractor is not flexible since, is specific purpose and it is not necessarily the most efficient. Having a trained features extractor, trained by labeled data as the classifier, is the way for moving from the NN to the Deep Neural Network(DNN). With this approach, the whole network, composed by many layers, is trained simultaneously, with the same train data.

DNN allows to model high level abstraction in data. When the network is trained the lower level layers extract simple features, moving towards higher layer, the features becomes more and more complex. A further important element in the DNN architecture is the polling layer which has the purpose of downsample the data.

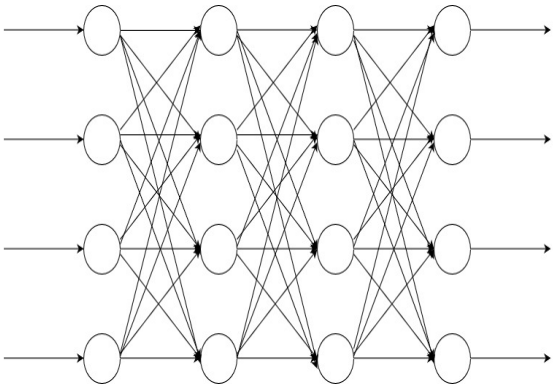


Figure 1.3: Fully connected Neural network example

Since, CNN outputs are sensible to the data location of the Input Feature Map (IFM), using the polling layer allows to reduce this sensibility, moreover, allows to reduced the memory usage in the inference process. With this layer the inference becomes more resilient with respect to data errors. Many techniques are applicable to this layer but the most used for his simple implementation is the max polling. It takes the highest values within subsets of the Output Feature Map (OFM) selected by a kernel that swipe over it[7].

1.2 PULP-NN Library

PULP_NN which was developed by the University of Bologna and the IIS lab of Zurich. As illustrated in their paper "PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors", the library is based on the CMSIS-NN. CMSIS-NN is a library that includes computational kernels needed for the inference computation, optimizing the DNN execution on a RISC-V core. Convolutional kernels, Maxpooling kernels and many other kernels are present. The authors of the above-mentioned paper[8], demonstrated that further optimization could be done to outperform the previous state of art DNN library in terms of computational and energy efficiency. This library beyond optimizing every kernel, improved the parallel computation of the ultra-low power RISC-V processors as well. Moreover, they optimized the kernels to increase the throughput by handling the parallel computation in a more efficient way and by taking maximum advantage from the data reuse of the input data and weights. However, the data reuse is physically limited by the hardware since the higher the amount of data to be reused, the bigger the memory footprint needed[8].

Data reuse is exploited using Single Instruction Multiple Data (SIMD) instructions approach, so that it's possible to compute multiple OFM data with only one load at the time of a certain IFM data with the corresponding weights. Therefore, it is possible to perform more than one MAC per load and store instructions.

The major innovation in PULP_NN is represented by the sets of kernels for

Quantized Neural Network (QNN) inference, targeting byte and sub-byte, from 8 bits to the atomic 1 bit. These sets of kernels were created to keep up with the general tendency to aggressively quantize data and QNNs are widely used nowadays for their overall performances[8]. A finite precision fixed point NN, or QNN, has a lower execution time with respect to an infinite precision floating point NN. In addition, the employment of a finite precision fixed point QNN brings about significant improvements from the low power perspective, which come from the target HW employ in the two different scenarios. The HW components for the QNN are simpler and a simple architecture entails a reduced number of components. Thus, the inference has a lower amount of HW computational steps, resulting in higher processing speed and greater power efficiency as a lower number of components have to change their value throughout the computational steps. Ultimately, the advantages of the fixed point quantized data lay in a faster inference computation and lower power consumption. The drawback in aggressively quantizing data is the accuracy drop. Sometimes, this accuracy reduction is insignificant and does not invalidate the application of the DNN. However, in safety-critical applications such a lack of accuracy can't be accepted thus, each case must be evaluated individually to get the best trade-off possible. The use of quantized fixed-point data is also an efficient way to reduce the memory footprint.

According to the paper, the RISC-V cluster with multiple parallel cores, with PULP_NN kernels, can reach 15.5 MAC/cycle on INT-8 data, improving the performances with respect to the sequential implementation on single core by a factor up to 63, making the execution considerably faster. The speedup, with respect to the single core, has a trend close to a linear speedup. Furthermore, they claim that by running the CIFAR-10 quantized model the inference reach a better energy efficiency, 14 times better with respect to a energy efficient MCU[8].

1.3 Fault Injection Methodologies

As discussed before, NNs are increasingly used in safety-critical fields and for this reason the reliability is an fundamental parameter of a NN that needs to be analysed and evaluated case-by-case for ensuring its dependability. Many approach at different abstraction levels can be employ, from the silicon level to the software level. This estimation is done by a fault injection campaign, that is the injections of several faults in the system one after the other. Afterwards, the results are then collected and classified to settle the overall NN reliability. As asserted in the paper "A Pipelined Multi-Level Fault Injector for Deep Neural Networks" The state-of-the-art faults injections methodologies can be classified in three main category as follow[9]:

1) **Simulation-based**

2) **Emulation-based**

3) **Platform-based**

The simulation-based is an effective methodology for evaluating the error resilience of a system, restoring accurate results. However, it may requires a preliminary investigation of the hardware and software design in order to define the right inputs parameters to supply as faults[10]. As mentioned in [9], the simulation-based can be further subdivided based on which abstraction level the faults injections are performed: Software or Hardware level.

The emulation-based methodology can be perform at different abstraction levels too. It is employed a prototype that it could be based on different hardware architecture (FPGAs, MCUs, etc...) and emulates the actual implementation. Supervising the injected faults can provide interesting information about the implementation robustness and dependability.

The platform-based methodology is perform in the actual platform with actual data. In this scenario would be more precise to speak of radiation injection rather than fault injection. The results can contain more information with respect to the software methodology since, it simulate the DNN deployed on the actual HW allowing a better understanding of the errors. Nevertheless, it is quite impossible to maintain time invariant the field conditions thus, there are some doubts about the validity of the results[10].

Additionally, taking as reference the paper "Fault and Error Tolerance in Neural Networks: A Review", the Fault Injectors are able to inject faults, errors and failures. The difference among them is the abstraction level that are going to compromise. Faults affects the physical level, errors the behavioral level and failures the application level. In this work the attention is focused on the hardware level then, towards faults. Even if, they are injected at low level, they may also affects and propagate toward higher levels thus, from the physical to the application level. Depending on the criticism of the fault, the DNN may be capable to masked it, restoring the correct results or the confidence levels of the predictions may change.

Faults can be divide into major blocks **permanent fault** and **transient fault**, both of them enclose different type of faults, as illustrated in the Table1.1 [11]:

FAULTS TYPE								
Permanent					**Transient**			
Stack at	Short	Open line	Bridging	Delay	**Intremittent**	**Timing**		
					Wearout	Bit-flip	Pulse	Delay

Table 1.1: Faults type

- **Stack at**: Simulates a signal line fixed to a logical level
- **Short**: Simulates the short circuit of two signals lines
- **Open line**: Simulates the division of a line
- **Bridging**: Simulates a open line short circuited
- **delay**: Simulates the permanent signal delay
- **Wearout**: Simulates the false contact
- **Bit-flip**: Simulates the signal state reversion
- **Pulse**: Simulates the spikes in the line
- **Delay**: Simulates the transitory signal delay

Chapter 2

Fault injection frameworks

2.1 Permanent fault injectors

Fault injector frameworks are constitute by all those commercial software that are able to mimic faults in the system under test. Despite such software are very powerful, they are only optimized to work with the regular architectures. For this reason, when it comes to the NNs architectures is going to be waste time during their simulations.

The architecture of the proposed fault injectors are divided in three main layers one on top of each other. The HW level is the lowest one in which the target hardware and the Fault Injector Unit (FIU) are describe in a Hardware Description language (HDL). FIU is the module that holds the faults to be injected and is the one that whenever it receives a signal from the synchronization level, inject the new faults. The application level is the intermediate level that has to communicate with the sync and the HW levels. This area is usually implemented in high-level language such as C or Python and it is where the NN executes the inferences. The sync level is responsible of the coordination of the inferences by starting the process describing the NN with the right timing. Additionally, this level has the Fault Classifier Unit (FCU), which has the task to gather the inferences results, classify them in such a way that it is possible to carry out statistics.

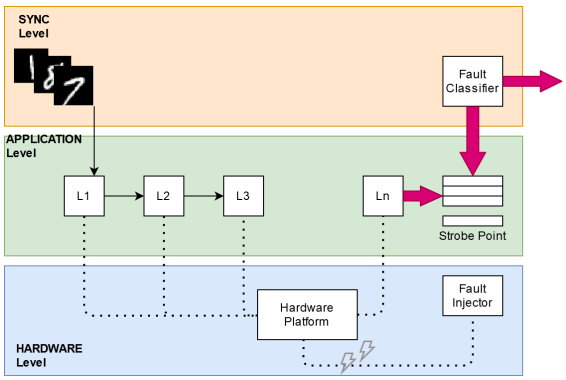


Figure 2.1: Sequential Fault Injector scheme

The state-of-art of the fault injector frameworks look at the NN system as a unique sequential big computational chunk even though, it is possible to subdivided this chunk in little blocks. As represented in Figure 2.1, even if each block represents a layer of the NN and they could be disjoint, the input image has to pass trough every single layer before starting a new inference with a new image. The sequential framework needs a Sync level to make the simulation starts right after it has finished the previous one and an application level directly connected to the hardware that allows the hardware simulator to emulate the behavior of the system.

Investigating how the computation flows in the NNs architecture, from a higher abstraction level, is clear that each layer can be seen as an independent, disjoint processing thread. Assuming this assessment correct, the pipeline technique is applicable to speed-up the simulation throughput at the cost of a higher latency. The general idea of pipelining is to make each block process its input data independently from a common starting point in time, stored the output in sync-registers, wait till all blocks are finished computing and then restart the subsequent computation. The data of the new computation step are taken from the sync-register to guarantee the natural data stream. A quick overview of the computational advantage is presented in the Figure 2.3

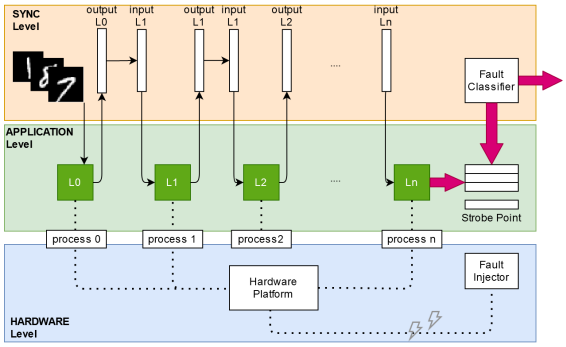
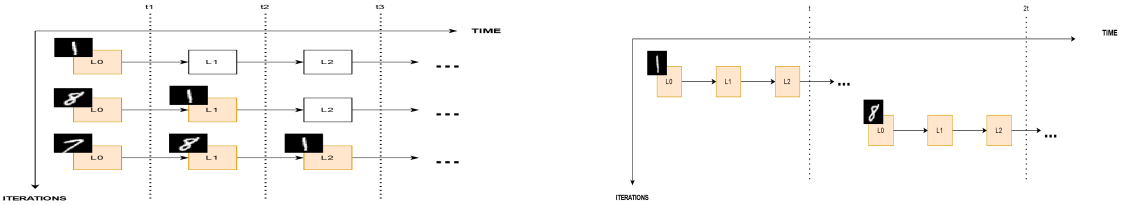


Figure 2.2: Permanent Fault Injector scheme

The Figure 2.2 illustrates the scheme of the novel architecture of a pipelined fault injector. It needs a Sync level to keep the system synchronized, storing and delivering the right data at the right time to the exact layer. As for the sequential framework the application level and the HW level are strictly bonded together but, in this case each layer is represented by an independent process.With this architecture the quicker blocks are penalized since they have to wait in order to start the new computation causing an increment of the latency.A further optimization would be reducing the total latency gathering more blocks together but, taking into account that the simulation time of this unification has to be comparable to the other blocks.

The drawback of this novel fault injector is given by the required system resources. The number of process go from one to the number of blocks in which the DNN has been divided. for this reasons, for complicated DNN the memory and CPU usage needs to be evaluated in order to make the fault injector works properly.



(a) Pipelining principle **(b)** Sequential principle

Figure 2.3: Simulation strategy comparison

2.2 Transient fault injectors

The state of art Fault injector frameworks for the transient fault are the same as for the Permanent Fault. For this simulation the time penalty is even worst with respect to the permanent fault case.

The state of art of the fault injector frameworks for the transient faults scenario treats the faults as if they were permanent fault settling a time constraints on the injected fault. As consequence the overall injection campaign time the same as the previous case. The major idea to optimize this particular framework is to move the portion of the NN not affected by the faults to the sync level decoupling those layer to the HW. Therefore, the highest and quicker level has to compute the information of all those unaffected layers, feed up and extract the right information to and from the "faulty" layer located in the application level. The sync level simulation takes no time compared with the simulation time at the HW level resulting in an overall simulation time equal to the execution time of the faulty layer. Hence, the pipeline technique for this framework is useless.

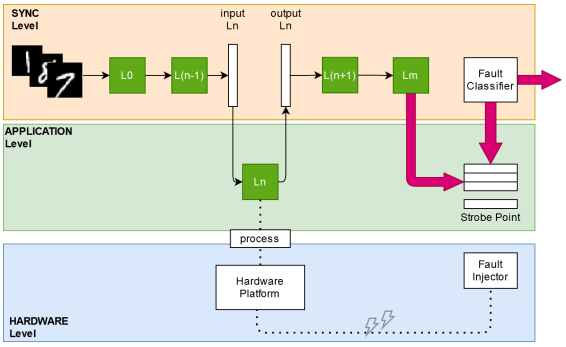


Figure 2.4: Transient Fault Injector scheme

Figure 2.4 shows the scheme of the transient fault injector framework. It is possible to observed that all the layers except one are placed in the Sync level and there is the presence of only one process that links to the hardware platform, the only one that can be affected by a Hardware fault.

Unlike the Permanent fault injector framework, the Transient Fault Injector Framework requires minimum resources. It needs one thread for the HW simulator and one thread for the higher abstraction level software per inference regardless of the DNN depth.

Chapter 3

Case study

3.1 NN with CIFAR-10 and MNIST

I have used a NN available in the GitHub repository[1] already trained with the CIFAR-10 dataset. All the weights and biases values were 8-bits signed data format. This NN is composed by 7 different layers with their own characteristic as detailed in the following table 3.1:

	Layers features			**Kernel**		
NN kernel	Input data	Output data	Activation function	dimension	stride	padding
Convolutional_1	32x32x4	32x32x32	ReLU	5x5x4	1	2
Maxpooling_1	32x32x32	16x16x32	None	3x3	2	0
Convolutional_2	16x16x32	16x16x16	RelU	5x5x32	1	2
Maxpooling_2	16x16x16	8x8x16	None	3x3	2	0
Convolutional_3	8x8x16	8x8x32	ReLU	5x5x16	1	2
Maxpooling_3	8x8x32	4x4x32	None	3x3	2	0
Fully connected	1x512	10	None	1x512	0	0

Table 3.1: CIFAR-10 Neural Network architecture

As presented from the Table 3.1 and the Table 3.2, the NN employed is a mixed classifier since, it has both convolutional and fully connected layers. To be specific, three convolutional layers and one fully connected. As activation function it is only used ReLU. The architecture envisages three maxpolling layers, each of which are put it right after the convolutional layers.

CIFAR-10 dataset consists in 60,000 RGB images 32x32 pixels in 10 different classes and is a subset of 80 million images dataset. The classes are mutually exclusive so there is no overlap between them. 50,000 images have been used to trained the network while 10,000 are used to test the accuracy of the training procedure.
Each images are characterized by a single byte label, followed by 1024 bytes

[1] https://github.com/pulp-platform/pulp-nn

corresponding to the red component of each pixel, 1024 bytes for the blue and 1024 for the green with a total of 3073 bytes.
The images of this dataset are low resolution images and sometimes they can be hardly recognizable by a human observer as is shown in figure 3.1[12]. The image 3.1 fall down to the boat class, at first sight this image is not easily identifiable.

Figure 3.1: Example of cifar10 image[1]

The accuracy results of this trained network with the CIFAR-10 dataset were awful, roughly 35%. For the purpose of testing the error resilience of the NN in an embedded HW it is not an adequate starting point since, without an high precision network is not possible to discriminates the errors coming from the injected fault or by the net itself. The network has been re-trained several times, however, the results didn't change significantly.

The solution of the accuracy problem comes out moving from the CIFAR-10 dataset to a simpler dataset as MNIST [2] is. This dataset consists in 70,000 of handwritten digits images 28x28 in grayscale in 10 different classes, from the digit zero to nine. Even here the classes are mutually exclusive so there is no overlap between them. 10,000 are available to test the accuracy of the the trained network. Each images are characterized by a label, in the range 0 to 9, describing the affinity of a specific class.

[2]http://yann.lecun.com/exdb/mnist

Figure 3.2: Example of images in the dataset[2]

Having changed the dataset, the NN has to be re-trained. Has been useful Python with the open-source Pytorch framework[13] since, it has been used to develop and train from scratch a prototype of the Neural Network running on PULP. After the development of the prototype two different approaches could have been chosen to train the NN. The first approach, is to make use of the 32 bit floating point, train the network and afterwards quantize everything to 8 bit fixed point data precision. The second approach, is to set up everything from the begging to work with the right data format. With the last approach would it be possible to get the highest accuracy possible, unfortunately it is under development right now. Then, I had to choose the first method employing 32 bits floating point data precision. The prototype trained with the MNIST dataset obtained a very successful result in term of loss or in other words, the distance between the input images with respect to the predicted images. The NN achieved approximately 99% of accuracy, with the 32 bits floating point data. However, being that PULP HW can work with 8 bits fixed point data precision, weights and biases have been quantized. This step slightly affect the accuracy of the network that still remains very good, 98.5%, allowing to export the prototype model, without any variation, to PULP framework making possible to perform meaningfull fault injections campaign in RTL.

The architecture of the NN with the MNIST dataset has been slightly modified compared with that described in the table 3.2 as follow:

	Layers features			**Kernel**		
NN kernel	Input data	Output data	Activation function	dimension	stride	padding
Convolutional_1	28x28x4	32x32x32	ReLU	5x5x4	1	4
Maxpooling_1	32x32x32	16x16x32	None	3x3	2	0
Convolutional_2	16x16x32	16x16x16	RelU	5x5x32	1	2
Maxpooling_2	16x16x16	8x8x16	None	3x3	2	0
Convolutional_3	8x8x16	8x8x32	None	5x5x16	1	2
Maxpooling_3	8x8x32	4x4x32	None	3x3	2	0
Fully connected	1x512	10	None	1x512	0	0

Table 3.2: CIFAR-10 Neural Network architecture

It is necessary to be highlighted that PULPs kernel works only with depth channel of multiples of four. Hence, the images from both MNIST and CIFAR-10 dataset requires to be pre-processed in order to fit in the PULPs kernel adding the right amount of "ghost" channels.

The following Figure 3.3 display a graphical block scheme view of the NN used by referring to the CIFAR-10 case:

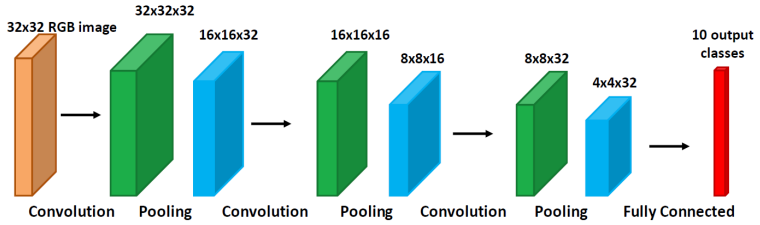


Figure 3.3: Neural Network block scheme[14]

The computation workload has been analyzed in order to understand how the evaluation of the neurons in the system is spread over the eight Core within the cluster. It was found that the neurons are mapped into the HW in a deterministic manner. Hence, the Cores are in charge of compute the same neurons every inference. From the image prospective it means that a pre-determined portion of the image is going to be process by a specific Core. For instance the next Figure 3.4 shows the Core0 image distribution workload.

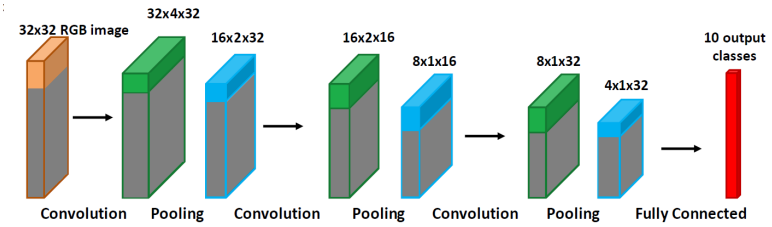


Figure 3.4: Core0 workload distribution [14]

The gray areas in Figure 3.4 represents the portion of the IFM computed by the other cores within the cluster. Every core has a similar deterministic workload distribution.

3.2 Pulp gap8

The SoC, PULP (Parallel Ultra Low Power), is a processing platform which is an open-source platform implementing an expanded version of the open-source RISC-V ISA think up for the NNs need. PULP lends itself to the development of intelligence devices employing an AI algorithm, such as autonomous driving, always listening, augmented reality, computer vision and many others.

The RISC-V hardware architecture is based on GAP8 SoC and is illustrated in his main building blocks in figures 3.5 and 3.6 .

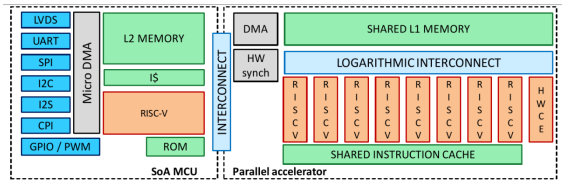


Figure 3.5: GAP8 overview[14]**(a)**

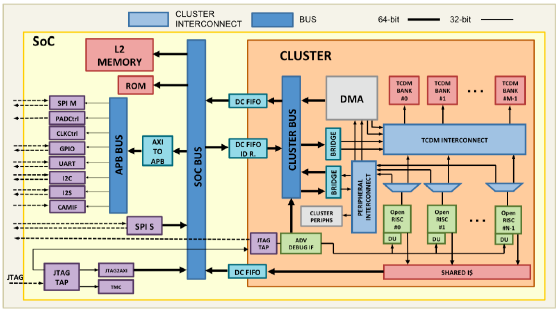


Figure 3.6: GAP8 overview[15]**(b)**

In the figure 3.6 it's possible to observe the presence of 8 cores in the cluster and one outside of it, for a total of 9 cores. The 8 cluster's cores have the task to carry out the parallel workload required by the inference steps of the DNNs, while the core outside the cluster, the so-called fabric controller, has to fulfill the communications and security functions. Each core can employ the DVFS (Dynamic Voltage Frequency Scaling), a low power technique that allows to operate at different voltage and frequency adapting quickly to the workload resources requirement.

There are two different memory levels, L2 and L1. L2 is the biggest memory in the system, accessible from every peripheral and elements within the processor. This memory level can transfer the data to the L1 memory with the DMA unit that allows fast data transmission without making use of the cores. Instead, L1 memory is smaller, it's shared among the 8 cores in the cluster and it's made up of many banks of memory interconnected via logarithmic interconnections.

Both FC core and the cluster's cores have their own instruction caches, respectively 1kB and 4kB. Generally, the cluster cores execute in parallel the same instructions on different data, working as an SIMD processor.

Another important unit is the uDMA that enables the data transfer from the different peripherals to the L2 memory without resorting to the FC core or with a limited need thereof, so that the FC core can perform its task in the meantime[15].

The Fiigure 3.7 illustrates the internal block of a RISC-V core. This core supports the standard RISC-V library as well as other specific instructions and it

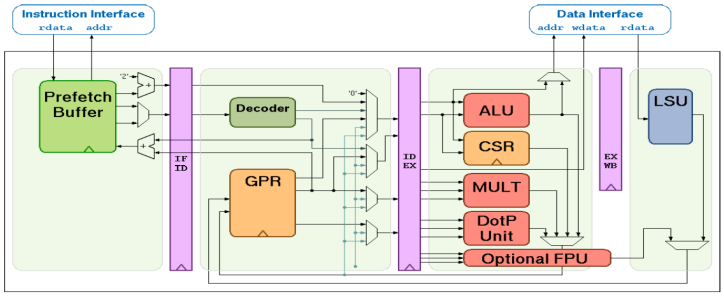


Figure 3.7: RISC-V Processor Block diagram[16]

has peculiar features, such as the HW loop and post-increment load and store[16]. The added instructions enable to work with vectors which are defined as groupings of sub-data. For instance, a vector of 32 bits can be defined as a group of four 8 bits data. This expedient helps exploiting the data parallelism, fundamental for making the computation for NNs efficient. Most of these functions are executed by the DotP Unit[16] which is essentially another ALU specialized to compute particular types of data. These data are usually sub-byte grouped in one single vector of 32 bits and computed simultaneously making the fixed point arithmetic encoding even more efficient. All OFM elements of every layer are calculated by means of a MAC operation done by the DOTP unit. Therefore, it's clear how important this unit is and how often it's employed during the inference.

Chapter 4

Experimental results

4.1 Framework for Permanent Faults

The dataset and NN described in Section 3.1 has been used. Moreover, the novel fault injector framework design has been employed with Modelsim as HW simulator. As mentioned in Section 3.1, the employed DNN is splitable in 7 sub-blocks each of which is different one by the other and characterized by its own execution time, as detailed in the Table 4.1. The times were obtained by simulating each layer independently. The different execution time are a consequence of the different amount of workload they have each layer has to fulfill.

Block	Kernel type	Execution time unit [[min]]
1	Convolutional	10:11
2	Maxpolling	4:00
3	Convolutional	9:39
4	Maxpooling	1:19
5	Convolutional	4:23
6	Maxpooling	1:07
7	Fullyconnected	1:39

Table 4.1: Layers execution time

The novel framework for permanent fault requires to have every blocks synchronized with each other, as consequence, the latency is no more the sum of the execution time of each blocks but, the multiplication of the greater execution time (T_{max}) times the number of layers. Hence, in this case the latency goes from about 25 minutes (T_{seq}) to about 70 minutes. Although the latency growth, from the moment when the pipelined is filled with functional data every T_{max} we have a new output instead of every T_{seq}. Therefore, the throughput is dramatically reduced, from 25 minutes to 10 minutes. A speed up close to 60%.

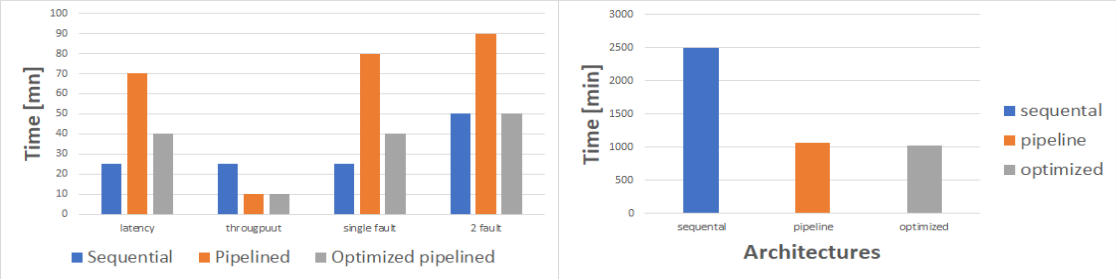
As described in Section 2.1, the last 4 blocks have been added together. Indeed, their execution time was still comparable to the others. In this way, the latency has been optimized by 40 minutes. A quick analytic comparison between the sequential framework, the pipelined framework and the optimized pipelined framework is

shown in the Table 4.2.

	Sequential	**Pipeline**	**Optimized pipeline**
latency [min]	25	70	40
Throughput [min]	25	10	10
Single fault [min]	25	70	40
2 fault [min]	50	80	50
100 fault [min]	2500	1060	1030

Table 4.2: Fault injectors summary

The the following Figure 4.1 is a graphical representation of the Table 4.2.



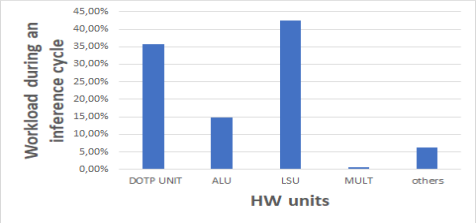
(a) Simulation time comparison **(b)** 100 faults simulation time

Figure 4.1: Frameworks simulation time comparison

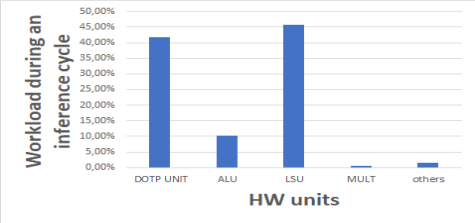
From the above Figures and Table is clear how this novel pipelined framework is an efficient solution for fault injections campaign of system based on NN. The advantage with respect the sequential framework comes after only a couple of faults. As long as the fault injection campaign expect to inject thousands of faults, the novel framework for a fault injections purpose is a straightforward choice.

The drawbacks of this novel fault injector are the required system resources. Right now to process the inferences the system has to run five process, four for the 4 blocks and one for the synchronization of them. For instance, with respect to the sequential case the memory usage goes from approximately 4.7G to 18.3G and as well the CPU has a heavier workload to fulfil. Optimizing the latency implicates the elimination of 3 process hence, the latency optimization as well as reducing the simulation time may helps to mitigate the resources problem. The previously mentioned data are relevant in relations of the size of the DNN used for this work. Despite the DNN is tiny and simple the memory usage is quite high. For this reasons, for more complicated DNN the memory and CPU usage needs to be evaluated in order to make the fault injector works properly.

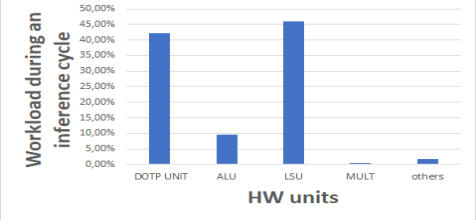
A purpose of this master thesis was to investigate the reliability of the NNs mapped onto the HW and prove wrong the general idea of error resilience of this technology due to their redundant structure. To this end, the framework previously described was used with the well-trained Neural Network defined in the Section 3.1. As a starting point, it was essential to analyze the workload of each processing element as well as the most used units. Examining every single assembly instruction, I came up with the following result embody in the Charts 4.2



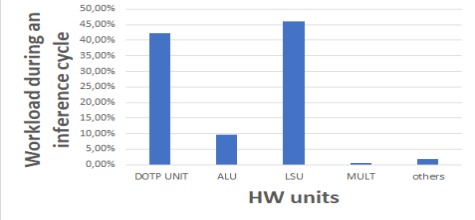
(a) Core0 Instructions distributions



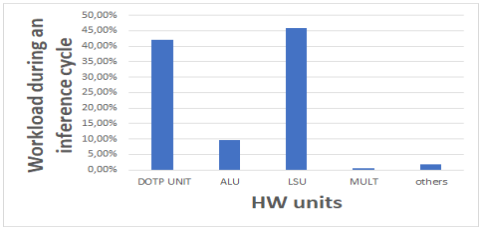
(b) Core1 Instructions distributions



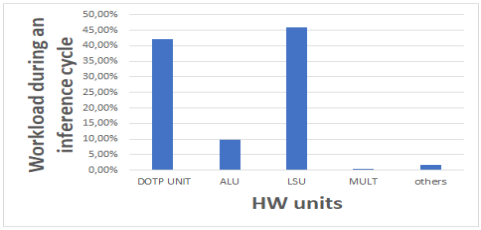
(c) Core2 Instructions distributions



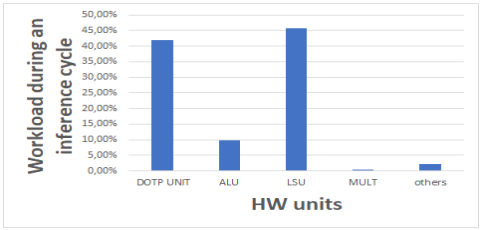
(d) Core3 Instructions distributions



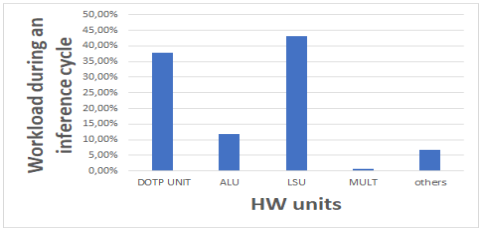
(e) Core4 Instructions distributions



(f) Core5 Instructions distributions



(g) Core6 Instructions distributions



(h) Core7 Instructions distributions

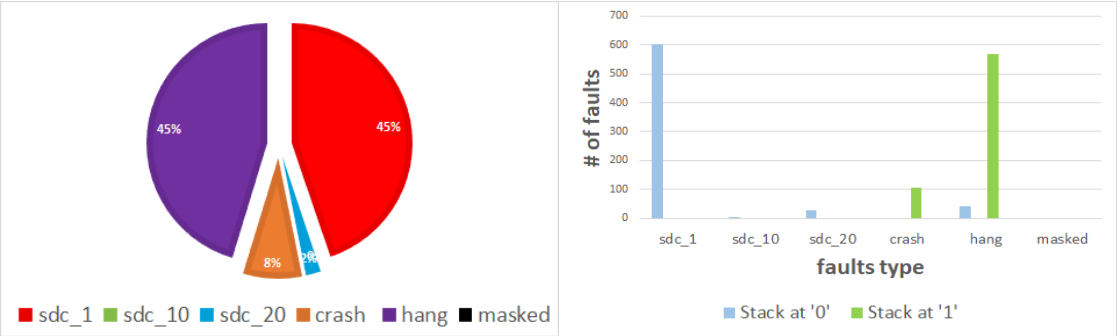
Figure 4.2: Instructions distribution from Core0 to core7

Figure 4.2 shows the units utilisation's analysis throughout the inference. As evidenced, the multiplier results to be the most used. This result was await since, the inference steps are based on MAC operations. Consequently, it has been decided to carried out a fault injection campaign on the multiplier of the Core 0 within the cluster, by relying on the faults classification proposed by [17] and [18]. The fault is said to be masked if the outcome of the inference does not change. In the opposite situation, the fault is detected. Whenever the fault is detected multiple scenarios are possible:

- **SDC-1**: Silent Data Corruption (SDC) occurs when the confidence level of the prediction deviates from the golden reference leading to a wrong image classification
- **SDC-10%**: Occurs when the confidence level of the prediction deviates from the golden reference within a range of +/- 10% but it is still correct
- **SDC-20%**: Occurs when the confidence level of the prediction deviates from the golden reference within a range of +/- 10% but it is still correct
- **hang**: Occurs when the fault makes the DNN execution looping in a certain state forever
- **crash**: Occurs when the fault makes the DNN execution stop

The permanent fault injections campaign has been conducted against the bits of the multiplier output of the Core 0. The bits have been individually stacked at '0' and at '1'. The dataset under test were a subset of MNIST composed by 21 different images. Therefore, the amount of injected faults were 1,344.

The following Figure reports the the results of the overall permanent fault injection campaign:



(a) Faults distribution **(b)** Stack at comparison

Figure 4.3: Permanent fault injection faults distribution

The Figure 4.3a shows the comprehensive outcome displaying the predominance of SDC faults and Hang faults. the Figure 4.3b shows the stack at '0' and '1' inference outcome. Is interesting how the stack at '0' and '1' produces a completely different outcome.

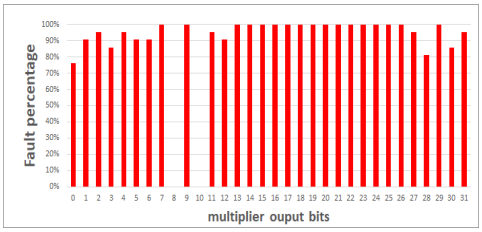
The following Table 4.3 illustrates the time required to extract the above results providing also a comparison with the time that it would be needed with the sequential framework.

SEQUANTIAL FRAMEWORK	**Total Injectios**	**T_latency [min]**	**T_througput [min]**	**images**	**Duration [h]**
	1344	∼25	∼25	21	∼560
PIPELINED FRAMEWORK	**Total Injectios**	**T_latency [min]**	**T_througput [min]**	**images**	**Duration [h]**
	1344	∼40	∼10	21	∼186

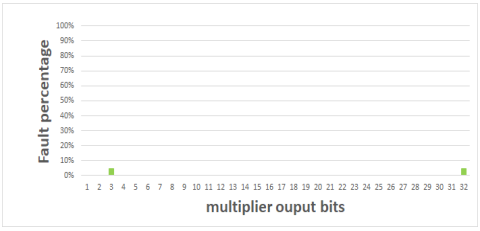
Table 4.3: Permanent time simulation

The last column of the Table 4.3 highlights the time efficiency of the new framework. As it is possible to deduce, the overall simulation speed up is around 68%. Previously, it was claimed that the overall speed up of the novel architecture is roughly around 60%. However, it is worth noting that it is the minimum advantage that you can achieve. Indeed, this is due to how the hang detection system has been adopted. The hangs are detected by means of a sort of watchdog which wipe the pipeline and starts a new inference if the DNN does not give a result in a certain amount of time.

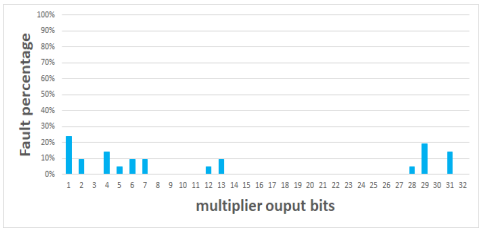
From the fault injection campaign It would have been expecting to have a regular pattern of SDC depending on the position of the faulty bit. However, as we can appreciate from the figure 4.4 and figure 4.5 apparently there is no correlation. This unexpected fault distributions behaviour is largely for the reason that the multiplier does not only calculates the neurons values but, it is also in charge to evaluate the instructions and data location. Consequently, an error in the results of the multiplier may have a huge impact in the system bringing it up to critical condition.



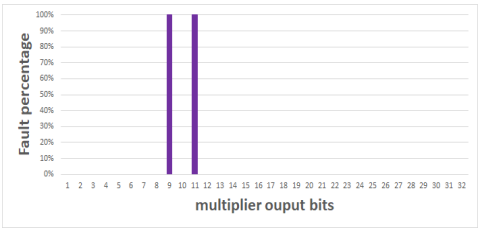
(a) stack at ’0’ SDC_1 faults



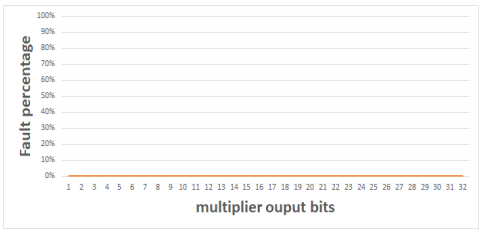
(b) stack at ’0’ SDC_10 faults



(c) stack at ’0’ SDC_20 faults

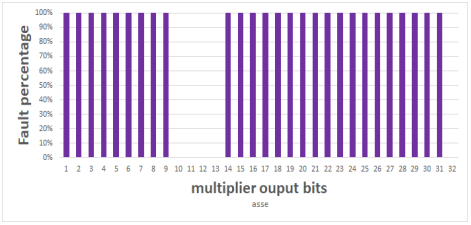


(d) stack at ’0’ hang faults

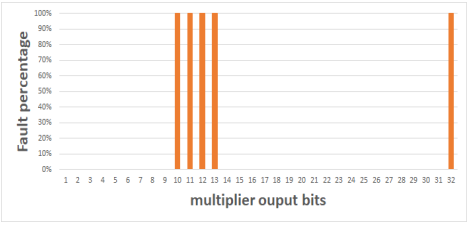


(e) stack at ’0’ crash faults

Figure 4.4: Stack at ’0’ faults distribution



(a) stack at ’1’ hang faults



(b) stack at ’1’ crash faults

Figure 4.5: Stack at ’1’ faults distribution

The faults for which there is not the graph, is due to they not occur for any bit. Is interesting to highlights the different outcomes when the bit is stack at ‘0’ and when is stack at ‘1’, figure 4.3b. In the first case the faults almost always create a miss-prediction or in other words an SDC faults. For the stack at ’1’ case the faults create a critical condition such as a hang or crash. Another thing to highlight is the fact that there is not a single masked error so, the network is not able to be unaffected by this type of fault. Then, considering the randomically nature of the errors, there is a 50% chance that a single permanent error in one of the multiplier unit inside the embedded HW can create a critical condition completely corrupting the NN functionalities. These results therefore, confirm the preliminary hypothesis of a low error resiliences of a NN mapped onto an embedded HW.

4.2 Framework for Transient Faults

The transient fault framework makes use of the same environment as the permanent fault framework. The only difference lies on the utilization of the higher abstraction level software represented by Python, as described in Section 2.2. For the permanent case Python is used for the synchronization purpose instead, for the transient case is used to perform the inferences of those NN's layers that are not going to be affected by the injected fault. Instead, the faulty layer belongs in the application level. As for the permanent framework case, the layers in the application level are those affected by the faults injection.

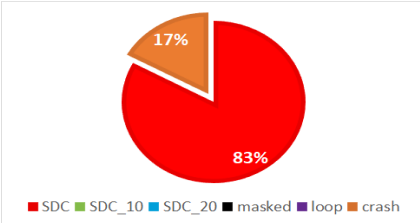
With this framework, the throughput corresponds to the latency and the fault injection simulation time is pretty much the execution time of the faulty layer under test. The execution time is illustrate in the Table 4.1. The insignificant overhead time is originated by the Python execution whose takes a few milliseconds. For the transient faults modes the NN described in Section3.1 compared to the sequential framework is very efficient. It is possible to reach an improvement that goes from about 60% to 95% . However, the hang detection strategy makes use of a pre-set countdown timer. The pre-set value has a positive slack compared to the execution time of the layer under test. Meaning that if the executions go in hang condition the time required to the fault injection campaign will be slightly higher.

This framework doesn't make use of the pipelining technique. Conceptually, it operates as the sequential framework since, to start a new inference with a new fault, the previous faulty inference has to be finished and classified. The simulations required only two threads at the time and the size of this threads depends on the faulty layer under test. Higher is the layer workload, higher will be the thread size. Anyway, the resources requisite are less severe and more affordable compared to the permanent faults mode.

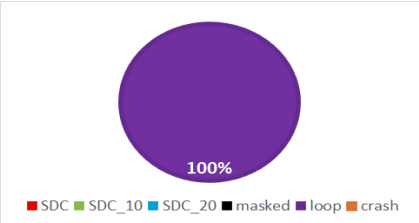
As mentioned before, a purpose of the thesis is to investigate the reliability of the NNs mapped onto the HW. In the previous section, Section 4.1, the fault injection campaign has been done with the permanent fault. However, it is not unlikely to have a transient fault affecting the system so, is important to complete the system reliability analysis taking into account the transient faults.

In this case the permanent fault injections campaign has been conducted against the first five bits of the multiplier output of the Core0 but, taking into consideration a computational layer at the time. The bits have been individually stacked at '0' and at '1' within the time range of the Modelsim layer execution time. The dataset under test were a subset of MNIST composed by 20 different images. Therefore, the amount of injected faults were 1,400.

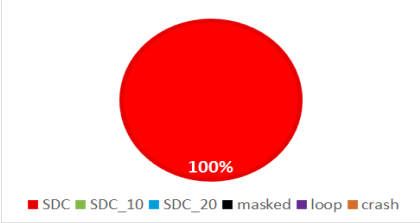
The following Figure 4.6, shows the faults distribution per layer of the transient faults.



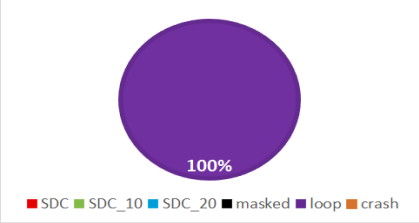
(a) Layer 1 stack at '0' faults



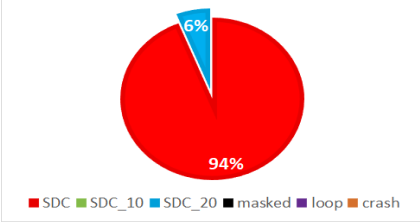
(b) Layer 1 stack at '1' faults



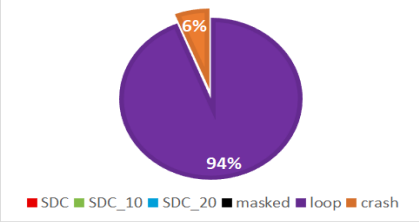
(c) Layer 2 stack at '0' faults



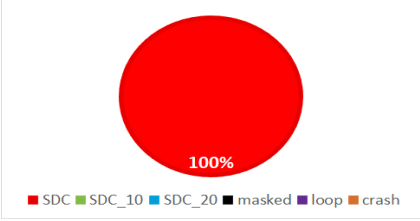
(d) Layer 2 stack at '1' faults



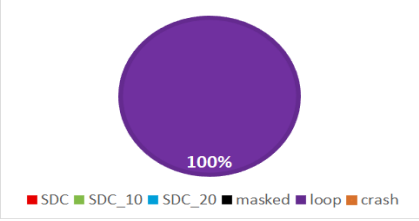
(e) Layer 3 stack at '0' faults



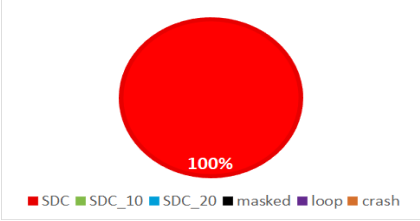
(f) Layer 3 stack at '1' faults



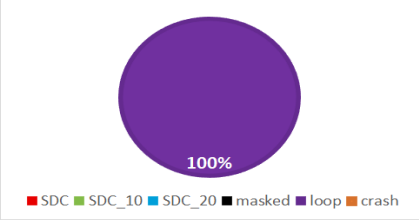
(g) Layer 4 stack at '0' faults



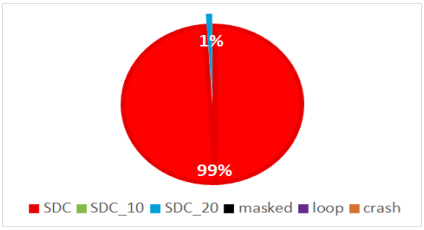
(h) Layer 4 stack at '1' faults



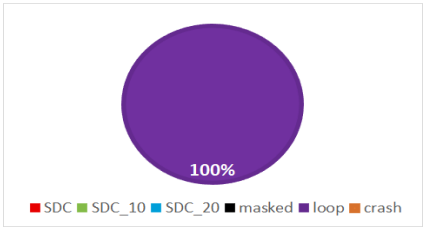
(i) Layer 5 stack at '0' faults



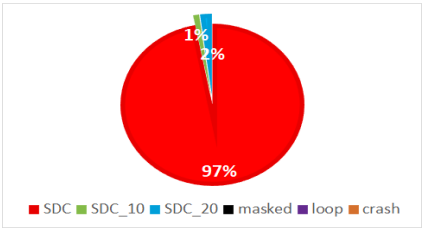
(j) Layer 5 stack at '1' faults



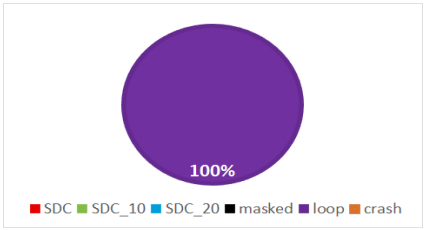
(k) Layer 6 stack at '0' faults



(l) Layer 6 stack at '1' faults



(m) Layer 7 stack at '0' faults



(n) Layer 7 stack at '1' faults

Figure 4.6: Transient faults distribution

The subfigures in pair represent the faults distribution of one layer. At the left the stack at '0' instead, at the right the stack at '1'. The way the Figure 4.6 displays the results highlights the radical behaviour differences of the system against the two types of faults. It is possible to appreciate that the SDC, for the stack at '0' and the hang, for the stack at '1' are practically the totality of the errors.

	Injectios per layer	**Layer1 [min]**	**Layer2 [min]**	**Layer3 [min]**	**Layer4 [min]**	**Layer5 [min]**	**Layer6 [min]**	**Layer7 [min]**	**Duration [h]**
Seq. F.	200	∼5000	∼5000	[min]	∼5000	∼5000	∼5000	∼5000	∼583
Opt. F.	200	∼2020	∼800	∼1878	∼238	∼843	∼214	∼278	∼104

Table 4.4: Transient time simulation

The Above Table 4.4 illustrates the time required to extract fault injections results providing also a comparison with the time that it would be needed with the sequential framework. Each columns describes the time required per layer. It is possible to observed a great time reduction thanks to the novel framework. The last column of the Table 4.4 highlights the time efficiency of the new framework. As it is possible to evaluate, the overall simulation speed up is around 82%, a value perfectly inside of the speed up range expectation. The saved time in terms of simulation days is about 20 days.

Chapter 5

Conclusion

5.1 Future work

The thesis presents two different software faults injector frameworks, one for the permanent faults and the other one for the transient faults. As discussed previously the novel fault injector framework are able to asses the dependability of the system injecting faults at the HW level. They have to take into consideration the system parameters as, weights, biases and other data. Furthermore, they take into account the target HW equipment as well.

The main purpose of this master thesis was to re-design the fault injector framework to optimally perform with the DNNs architectures. It has been exploited the injections of permanent faults and transients faults. The obtained results are extremely great in terms of frameworks optimization. The overall simulation time speed up is greater than two times than the old framework for the permanent fault simulation instead, the transient fault simulation time is correlated with the HW layer under test. The only drawback lies on the permanent fault injector which requires a huge amount of memory and computational power depending on the DNN size.

The experimental results shown two important aspects: The effectiveness of the novel fault injector frameworks and the incapability of the embedded HW, bonded with the DNN software design, to be in most of the cases immune with respect to the faults. Indeed, more than 80% of the cases results to be a Silent Data Corruption faults and hang faults. Moreover, there are not masked faults. Therefore, the system with the injected faults wasn't able to exploit it's peculiarity of data redundancy and masked the error. Besides, transient faults were expected to be less critical since, only one layer at the time was hit by the faults. Instead, the transient faults have more or less a similar faults distribution compared with the permanent faults.

As specified, this technology would be use in safety-critical fields application. However, even if has been conducted non-exhaustive simulations, the experimental results are not promising. From the obtained results it possible to say that: the system based on the NN needs only a fault, permanent or transient, to be untrustworthy and endanger human beings. For this reason, the HW and SW co-design needs to be improve or change.

Since, the faults injectors haven't been used extensively, an immediate future work can be a detailed faults injection of all the units within the SoC. The muliplier of the Core 0 has been the target of the fault injection campaign because it is the most used unit. Anyway, other units are largely use during the inference, as the ALU and the DOTP unit. So, with the new faults injectors framework, it will be possible to error-wise characterize the system behavior of the system inferences identifying the most critical units. This type of analysis can help the engineering designer to choose the most error resilient HW, or try to improve the the HW and SW co-design.

Appendix A

Code: How to use

PERMANENT FAULT FRAMEWORK

The NN's layers are divided in 7 folders: *test1*, *test2*, *test3*, *test4*, *test5*, *test6*, *test7*. Each folder corresponds to a layer of the NN.

The framework is composed by the following files:

- Principal file: *python_inference.py*
- Functions file: *function.py*
- Binary Dataset file: *t10k-images-idx3-ubyte*

The framework make use of the following files to load the information of the NN and to extract the computed data from layers output:

- *Conv*_weight.txt*
- *Conv*_Weight.txt*
- *Fullyconnected_weight.txt*
- *Fullyconnected_Weight.txt*
- *Conv*_mem8.txt*
- *Maxpool*_mem8.txt*
- *test1/build/pulp/tcl_files/inj_bak.txt*

The file *inj_bak.it* contains the faults to be injected. The framework starts whenever the Python file *python_inference.py* is launch. The principal parameters to set are the following:

- Number of images to compute: *Number_of_inference*
- Number of faults to injects: *Number_of_faults*

TRANSIENT FAULT FRAMEWORK

The NN's layers are divided in 7 folders: *test1*, *test2*, *test3*, *test4*, *test5*, *test6*, *test7*. Each folder corresponds to a layer of the NN.

The framework is composed by the following files:

- Principal file: *Transient_NN_MNIST.py*
- Functions file: *function.py*
- NN structure file: *mnist_net.pth*
- Dataset file for pythorch: *MNIST*
- Binary Dataset file: *t10k-images-idx3-ubyte*

The framework make use of the following files to load the information of the NN and to extract the computed data from layers output:

- *Conv*_weight.txt*
- *Conv*_Weight.txt*
- *Fullyconnected_weight.txt*
- *Fullyconnected_Weight.txt*
- *Conv*_mem8.txt*
- *Maxpool*_mem8.txt*
- *test1/build/pulp/tcl_files/inj_bak.txt*

The file *inj_bak.it* contains the faults to be injected. The framework starts whenever the Python file *Transient_NN_MNIST.py* is launch and *conda* enviroment is set. The principal parameters to set are the following:

- Faulty layer under test: *faulty_layer*
- Number of images to compute: *Inference_vector*
- Number of faults to injects: *Number_of_faults*

Bibliography

[1] E. Flamand et al. «GAP-8: A RISC-V SoC for AI at the Edge of the IoT». In: *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2018, pp. 1–4 (cit. on p. vii).

[2] Michael N. Vrahatis, George D. Magoulas, Konstantinos Parsopoulos, and V.P. Plagianakos. «Artificial Neural Networks for Beginners». In: (2000), pp. 1–2 (cit. on pp. 1, 5).

[3] Daniele Palossi et al. «A 64-mW DNN-Based Visual Navigation Engine for Autonomous Nano-Drones». In: *IEEE Internet of Things Journal* 6.5 (Oct. 2019), pp. 8357–8371. ISSN: 2372-2541. DOI: 10.1109/jiot.2019.2917066. URL: http://dx.doi.org/10.1109/JIOT.2019.2917066 (cit. on p. 1).

[4] N. Smolyanskiy, A. Kamenev, J. Smith, and S. Birchfield. «Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness». In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 4241–4247 (cit. on p. 1).

[5] Carlos Gershenson. «Introduction to artificial neural network training and applications». In: (2003), pp. 1–3 (cit. on p. 4).

[6] Abien Fred M. Agarap. «Deep Learning using Rectified Linear Units (ReLU)». In: (Feb. 2019), p. 2 (cit. on p. 5).

[7] B. Mehlig. «Artificial Neural Network». In: (Feb. 2019), pp. 3–140 (cit. on p. 6).

[8] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. «PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors». In: (Aug. 2019), pp. 1–12 (cit. on pp. 6, 7).

[9] Annachiara Ruospo, Angelo Balaara, Alberto Bosio, and Ernesto Sanchez. «A Pipelined Multi-Level Fault Injector for Deep Neural Networks». In: (July 2020), p. 2 (cit. on pp. 7, 8).

[10] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. «Fault Injection Techniques and Tools». In: (May 1997), pp. 75–76 (cit. on p. 8).

[11] CESAR TORRES-HUITZIL (Senior Member IEEE) and BERNARD GIRAU. «Fault and Error Tolerance in Neural Networks: A Review». In: (Sept. 2017), pp. 1–5 (cit. on p. 8).

[12] AlexKrizhevskyr. «Learning Multiple Layers of Features from Tiny Images». In: (Apr. 2009) (cit. on p. 15).

[13] Adam Paszke et al. «PyTorch: An Imperative Style, High-Performance Deep Learning Library». In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf (cit. on p. 16).

[14] Angelo Garofalo, Manuele Rusci, Francesco Conti, Davide Rossi, and Luca Benini. «PULP-NN: Open-Source Library for QNNs Inference on RISC-V Based PULP Clusters». In: (June 2019), pp. 1–12 (cit. on pp. 17, 18).

[15] Micrel Lab, Multitherman Lab, University of Bologna, Italy Integrated Systems Lab, and Switzerland ETH Zürich. «Pulp Hardware Reference Manual». In: (Feb. 2019), pp. 13–17 (cit. on p. 18).

[16] Andreas Traber and Michael Gautschi andPasquale Davide Schiavone. «RI5CY¦ User Manual». In: (Jan. 2020), pp. 1–20 (cit. on p. 19).

[17] Guanpeng Li et al. «Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications». In: SC '17. Denver, Colorado: ACM, 2017, 8:1–8:12. ISBN: 978-1-4503-5114-0. DOI: 10.1145/3126908.3126964 (cit. on p. 23).

[18] B. Du, J. E. R. Condia, M. S. Reorda, and L. Sterpone. «About the functional test of the GPGPU scheduler». In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. 2018, pp. 85–90 (cit. on p. 23).