

POLITECNICO DI TORINO

Master's Degree in Computer Engineer



Master's Degree Thesis

Errors in Deep Neural Networks for Deep Space: Observations, Explorations, and Remedies

Supervisors

Prof. LUCA STERPONE

Candidate

MIRIAM GRASSO

JULY 2021

Summary

Even more ambitious plans and missions are being conceived by farsighted researchers who dealing about the spce exploration and the consolidation of artificial intelligence methods in space engineering is certainly an enabling factor. Deep neural networks are responsible for some of the greatest advances in modern computing. Given that in space, devices are no longer protected from Sun's radiation by the Earth's atmosphere, this can cause spurious errors or stuck transistors in the device's circuitry. This thesis discussed the issues of a DNN working in deep space. My thesis goal is to study and investigate how DNNs work to the accumulation of radioactive dose, discovering their weaknesses and strengths, through the development of a fault injection (FI) platform that emulates the errors caused by radiation.

The study explores the weaknesses of the DNN and exploited the effectiveness of using pruning as a novel defense approach against faults.

Results have shown that pruning not only improves the resource efficiency of neural networks, but also the resilience against faults. In particular, we found that the resilience depends on the level of pruning, the model retains higher resilience where less trainable parameters remained after pruning.

Table of Contents

List of Tables	v
List of Figures	vi
Acronyms	ix
1 Introduction	1
1.1 Radiation effects in devices	2
1.2 Artificial Intelligence	3
1.2.1 Artificial Intelligence Methods in Space Engineering	5
2 The Problem	10
2.1 Radiation in deep space	10
2.1.1 Single-event upset	12
2.1.2 Physical hardening	13
2.2 Radiation in AI-based device	13
2.2.1 Single Event Upset in Neural Networks	15
3 Technologies	16
3.1 OVERVIEW OF DNN-S	16
3.1.1 Artificial Neuron	17
3.1.2 Convolutional Neural Networks (CNNs)	18
3.1.3 Convolutional layers	22
3.1.4 Pooling Layer	24
3.1.5 Fully Connected Layers	24
3.1.6 Popular DNN Models	25
3.2 DNN DEVELOPMENT RESOURCES	28
3.2.1 Frameworks	28
3.2.2 Models	32
3.2.3 Datasets for Classification	32
3.2.4 Dataset in my analysis	33

3.2.5	Traning	33
3.2.6	Inference	35
3.2.7	Tensor	36
3.3	HARDWARE FOR DNN PROCESSING	37
3.3.1	Overview of platforms	38
3.3.2	Inference on HW	39
3.4	METRICS	40
3.4.1	Metrics for DNN Models	40
3.4.2	Metrics for DNN Hardware	40
3.4.3	Fault tollerance Metric	41
4	Methodology	43
4.1	Fault injection	47
4.1.1	My FI technique	49
4.2	Accumulation of SEU-effects within the DNN	54
4.3	Fault Tolerance	57
4.3.1	My resilience techniques	58
5	Results	64
5.1	Single bit flip within the DNN results	64
5.1.1	The most vulnerable bits	64
5.1.2	Layer Vulnerability	68
5.2	FI results	71
5.3	Faults Tolerance results	73
5.3.1	Prining results	76
6	Conclusions and future work	79
	Bibliography	82

List of Tables

4.1	An example of Alexnet layer probability of being affected by an SEU/error/failure	52
5.1	Model comparision: Total parameters in models	73

List of Figures

1.1	Earth's Van Allen Belt	3
1.2	Deep Learning in the context of Artificial Intelligence.	5
1.3	Applications of AI/neural network accelerators	6
1.4	Radiation cause errors in the device's circuitry	8
2.1	Charged particle strike an integrated circuit	11
2.2	soft error effect in an integrated circuit	12
2.3	The vulnerable components and parameters of a single neuron to fault injection	14
3.1	Artificial Neural networks	17
3.2	Artificial Neuron	17
3.3	Convolutional Neural Networks	18
3.4	Convolutions in CNNs	19
3.5	Standard convolution	21
3.6	depth-wise convolution	22
3.7	Depth-wise separable convolution	23
3.8	ConvNet arranges its neurons in three dimensions	24
3.9	Pooling layer down-samples the volume spatially. The most common down-sampling operation is max, giving rise to max pooling.	25
3.10	<i>Example of MnasNet Architecture</i> – (a) is a representative model; (b) - (d) are a few corresponding layer structures. MBConv denotes mobile inverted bottleneck conv, DWConv denotes depthwise conv, $k3 \times 3/k5 \times 5$ denotes kernel size, BN is batch norm, $H \times W \times F$ denotes tensor shape (height, width, depth), and $\times 1/2/3/4$ denotes the number of repeated layers within the block.	27
3.11	(a) Standard convolutional layer with batch normalization and ReLU. (b) Depth-wise separable convolution with depth-wise layers followed by batch normalization and ReLU.	28
3.12	Shortcut module from ResNet	29
3.13	Imagenet dataset	32

3.14	The confidence scores for a image	35
4.1	Architecture of general DNN accelerators	44
4.2	Hardware Accelerators of Deep Learning Models	44
4.3	Threats to a DNN-based system	46
4.4	FI simulation - workflow	50
4.5	Hardware implementation of memory neural network	51
4.6	Golden output example	51
4.7	(a) Golden DNN (b), random single bit-flip ,(c) random multiple bit-flip	56
4.8	Synaptic Pruning in mammals	60
4.9	DNN pruning	60
5.1	Impact of bit-flip on models (MC and D)	65
5.2	Experimental setup for illustrating the impact of memory faults in DNN execution	66
5.3	Single precision floating point storage format used in DNN design .	67
5.4	Impact of bit flip errors on the accuracy of VGG-f network used for an image classification application	67
5.5	The results of 0'b to 1'b bit flip error injection in the most vulnerable bit	68
5.6	Hierarchical representation learning by a CNN where the initial layer detects simple patterns like edges and gradients while higher layers detect more abstract features	70
5.7	A deep neural network example and the general structure for a neuron.	70
5.8	Fault injection attack on DNN	71
5.9	Impact of bit-flip on Resnet through the layes	72
5.10	Impact of bit-flip on Alexnet through the layes (Degradation) . . .	72
5.11	Impact of bit-flip on Alexnet through the layes (MissClassification)	73
5.12	Failure rate function AlexNet	74
5.13	Failure rate function ResNet	74
5.14	Failure rate function MobileNet	75
5.15	Failure rate function MnasNet	76
5.16	Failure rate function ResNet (a) ResNet (original) (b) ResNet (25%) (c) ResNet (46%)	77
5.17	ResNet Pruning Results Summary	77
5.18	Failure rate function MobileNet (a) MobileNet (original) (b) Mo- bileNet (0.72x	78
5.19	Failure rate function MnasNet (a) MnasNet (original) (b) MnasNet (0.7x)	78

Acronyms

AI

artificial intelligence

DNN

deep neural network

CNN

convolutional neural network

DSP

digital signal processor

BRAM

Block Random Access Memory

FF

flip-flop

GPU

graphics processing unit

TPU

Tensor Processing Unit

ASIC

application specific integrated circuit

FPGA

Field Programmable Gate Array

SEU

single-event upset

SEEs

Single Event Effects

MAC

Multiply Accumulator

FI

Fault Injection

MC

Miss-Classification

D

Degradation

Chapter 1

Introduction

The appearance of the first intelligent civilizations in the world is also accompanied by some questions about our existence, the world and its nature. Man has always tried to understand its nature, its mysteries and the mysteries of the cosmos. Man is born an explorer, an explorer of lands, territories, borders, frontier, civilizations and culture.

The evolution of civilization has led to answering some questions, up to the awareness that our earth is nothing more than a tiny blue dot in the vastness of the cosmos. And if we have explored, studied, admired, understood our earth but still reserves many questions, the universe and the cosmos reserve even more. And this is where science and technology have come to help us, to understand its nature and preserve our land. And here, driven by the boundless fascination for this vast cosmos, we study techniques that allow us to expand his vision and knowledge.

There are so many factors that maintain the right balance that allows us to stand here and ask ourselves how the universe is made and what would happen if things were not as they are. In some way, we human beings are small pieces of the universe that reflect on the universe itself. We are the universe that becomes aware of itself and wonders who it is.

The Earth and the Sun are intimately connected not only by light and gravity. Our planet is immersed in the solar magnetic field, and on the wings of this field travels a continuous flow of particles with an electric charge that originates from the surface of the Sun, a real "wind" that blows in every corner of our planetary system. : the solar wind. These particles travel at over 400 kilometres per second,

about 1,500,000 kilometres per hour, and are very harmful to life. However, our planet is well protected: its magnetic field acts as a shield, deflecting the particles of the solar wind and thus ensuring a safe environment for terrestrial life in which to develop.

The Earth is well protected from radiation. But what happens when we cross the Earth's atmosphere?

Our curiosity has pushed us beyond the borders of our planet, and technology has come to meet us. We have developed devices capable of exploring hostile territories, studying them and feed our curiosity, to know their nature and monitor their activities. And it is here that we have sent extraordinary electronic devices into orbit, capable of withstanding hostile environments and able to tolerate the large charge of radiation they encounter.

New technologies are always studying new methods to make these devices resistant to the most hostile conditions.

Radiation is energy in the form of tiny, subatomic waves or particles. For space devices, the primary concern is radiation made by particles. Because we know spacecraft life is not easy, invisible and energetic particles can alter their electronic components, degrading their performance and in the worst case making them unusable.

The effect of radiation also depends on the path intended for a device, in general, the effect of solar particles is heavier as we get closer to the sun, in any case, galactic cosmic rays (particles ejected into space from distant stars) can meet anywhere.

And so, as most space agencies undertake missions for deep space exploration, to the edge of the solar system, radiation testing becomes more and more crucial, the basic question to ask is: "Will we be able to guarantee that humans, electronics, spacecraft and instruments, all that we are sending into space, will survive in the environment in which they will have to operate?"

1.1 Radiation effects in devices

In space, a charged particle may cause a disruption or permanent damage, it strikes a sensitive node of an electronic circuit.

Our sun releases many charged and uncharged particles which include protons,

heavy ions and neutrons. On the other hand, constantly, there is a stream of charged particles from cosmic rays that strike the Earth. Many of these charged particles get trapped in the Earth's magnetic field and constantly circle the planet. The Earth's Van Allen Belt, which traps these charged particles, is shown in Fig. 1.1.

Given that in space, devices are no longer protected from Sun's radiation by the Earth's atmosphere, this can cause spurious errors or stuck transistors in the device's circuitry.

Radiation damages the hardware either through its cumulative effects/total ionizing dose (TID) or through its transient effects/single event effects (SEE).

Radiation hardening (rad-hard) allows a compute component to withstand such errors. Rad-hard components are twice as slow and many times as expensive as their regular equivalent single part.

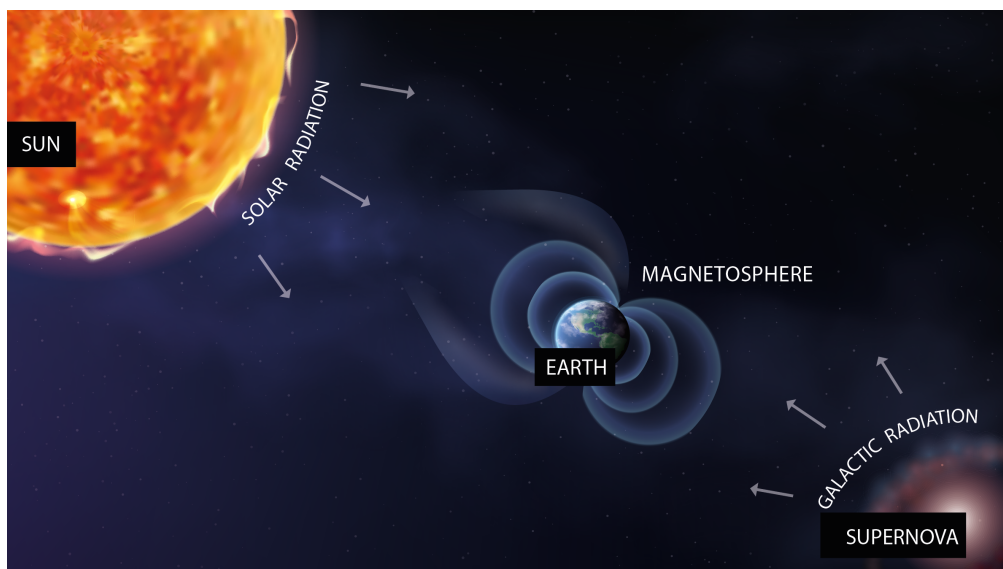


Figure 1.1: Earth's Van Allen Belt

1.2 Artificial Intelligence

The comeback of humans to the Moon and a future manned mission to Mars seem to be likely achievements we may witness in the next few decades. Meantime, even

more, ambitious plans and missions are being conceived by farsighted researchers who dream about the exploration and colonization of even farther planets. In the framework of these concrete future scenarios, the consolidation of artificial intelligence methods in space engineering is certainly an enabling factor.

The advances achieved by man in the field of space exploration are sensational and these also thanks to the use of technology, a technology that in recent decades has seen improvement in many respects. The last frontier of innovation sees growing interest in the use of neural networks for many fields of application.

Hardware platforms specialized for AI tasks have been applied in various applications and services, offered in the whole spectrum from the cloud to edge, close to the consumer; therefore, such hardware platforms have made commercial, industrial, and defence products feasible. End-user devices, such as autonomous cars, smartphones, and robots, are a few examples of such products, see Fig 1.2.

Deep neural networks are responsible for some of the greatest advances in modern computing. They have helped make substantial progress in long-standing problems: computer vision, speech recognition, and natural language understanding. By significantly improving the ability of computers to understand the meaning of the world, deep neural networks are changing not only the field of information technology but almost every field of science and human commitment.

As shows in Fig. 1.3, DNNs (also referred to as deep learning) are a part of the broad field of AI, which is the science and engineering of creating intelligent machines that can achieve goals as humans do.

Within artificial intelligence is a large subfield called machine learning and it represents the computer ability to learn without being explicitly programmed. So a single program will be able to learn how to do some activities outside the notion of programming. This is in contrast to purpose-built programs whose behaviour is defined by hand-crafted heuristics that explicitly and statically define their behaviour.

Within the machine learning field, there is an area that is referred to as brain-inspired computation ("the best 'machine' we know for learning and solving problems"). So a brain-inspired computation is a program or algorithm that takes some aspects of its basic form or functionality from the way the brain works.

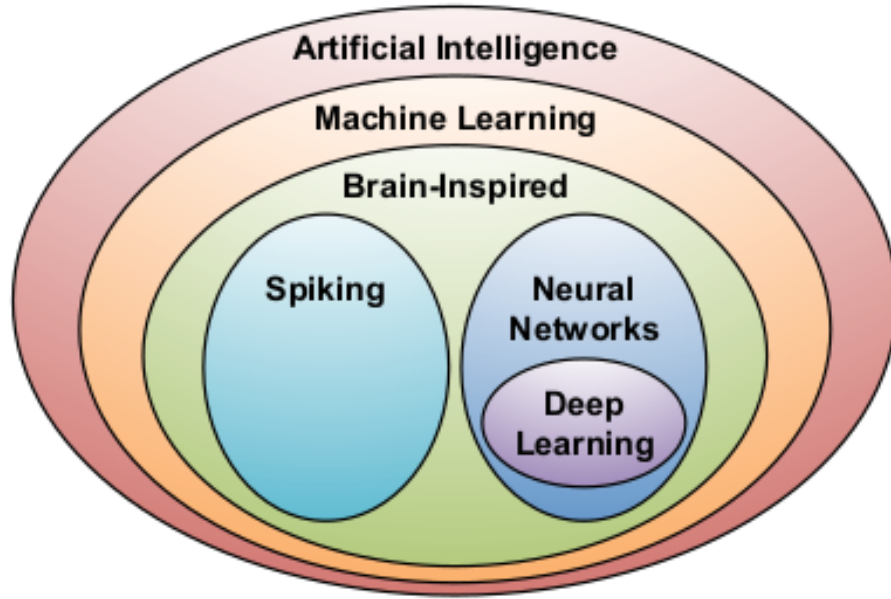


Figure 1.2: Deep Learning in the context of Artificial Intelligence.

1.2.1 Artificial Intelligence Methods in Space Engineering

The most successful AI implementations based on DL are rarely used in the space industry today.

ML still has a long way to go before it is widely used for space applications, but we are already starting to see it implemented in new technologies.

Potential applications of AI are also being thoroughly investigated in satellite operations. In particular, to support the operation of large satellite constellations, including relative positioning, communication and end-of-life management.

In addition, it is becoming more common to find ML systems analysing the huge amount of data that comes from various space missions. The data from some Mars rovers are being transmitted using AI, and these rovers have even been taught how to navigate by themselves.

Its development has come a long way over the last couple of decades, but the complicated models and structures necessary for ML will need to be improved before they can be greatly useful.

Generally, developing autonomous spacecraft that use artificial intelligence to take

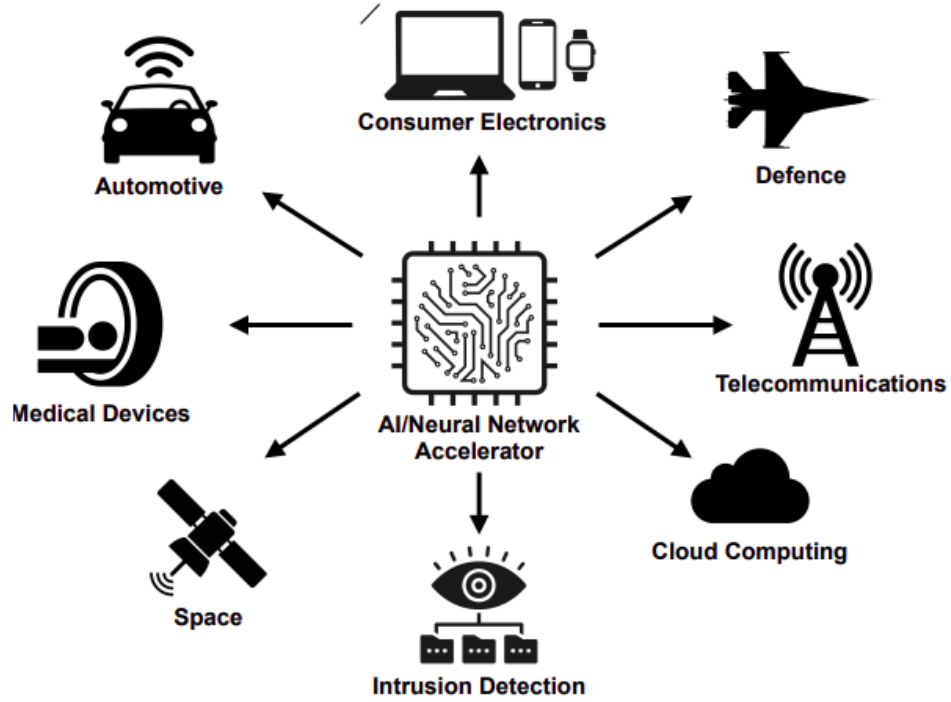


Figure 1.3: Applications of AI/neural network accelerators

responsibility for themselves would be very useful for exploring new parts of the Solar System and reducing mission costs.

Numerous studies have been conducted, that identified the necessary technology to improve automation, including autonomous navigation, automated telemetry analysis and software upgradability, also investigated using ML in the area of guidance, navigation and control. In particular, they looked into using big swarms of small robots that share their information in a network: if one robot learns from experience that a certain stratagem is beneficial, the entire swarm learns this.

AI also currently lacks the reliability and adaptability required in new software; these qualities will need to be improved before it takes over the space industry.

And this is where my work comes into play, the studies conducted in the field of ML and in particular in the study of DNNs refer to infinite uses on earth. These models may often not be suitable for a more hostile context, such as the one outside our planet.

My thesis goal is to study existing ML models and techniques and discover their

weaknesses and strengths, trying to build a robust and resilient model, which is well suited to this context.

We have established that in space the devices undergo radiations that alter the behaviour of the circuitry, alterations that can cause the overturning of bits. The consequences of altering the circuitry impact the operation of the real application. We have seen how new scientific research is pushing forward and involving the use of NN in support of space missions, but what happens when the behaviour of the circuitry is altered by the numerous radiations?

Let's make a brief introduction, a neural network model will be trained to perform one of the multiple possible tasks, this means that at the end of the training, I will have a model able to receive new inputs. Given the new inputs, the model will be able to measure the correct output, thanks to the parameters that have been trained to achieve this task. Parameters are therefore a crucial requirement for the application to operate in a real context.

Goal: our investigation starts by analyzing first what happens to the model when our charged particles have caused a bit-flip in the memory of our device, as shown in Fig. 1.4, we will explain in detail how this event happens in the next chapter. The alteration in the device memory is directly correlated to an alteration in the DNN parameters. What happens when a parameter of the NN is changed/modified? Will the model still work fine or will it make it impossible to work?

Answering this question needs tests, there are many tests carried out by simulating the possible positions that could impact the device on the hardware, the tests at the hardware level are hard and expensive. My work focuses on testing and studying the behaviour of the model by simulating the possible failures that can affect the NN through the software, in particular through the Monte Carlo technique to determine the probability that the NN is affected by failures.

Once I have studied the outcomes that a single bit-flip has on our DNN, I want to find out the outcomes that the accumulation of soft errors has on my DNN.

After a set amount of failures our network could have completely changed its functioning, it is therefore very important to find out the number of failures that the NN can tolerate. In particular, we could set a limit threshold of failures in

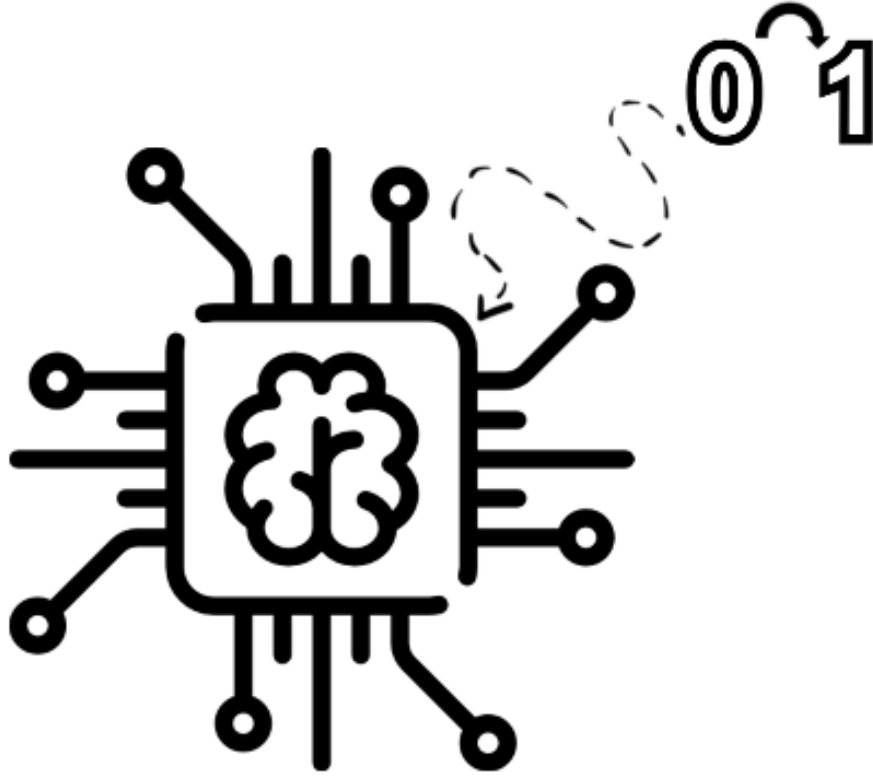


Figure 1.4: Radiation cause errors in the device's circuitry

order to re-program the device or apply refresh mechanisms to restore the initial weights and guarantee that the application always works accurately.

The first part of my thesis focuses on analyzing and studying some models of NN, knowing which DNN work best and what are the strengths to create an architecture that is best suited to an environment teeming with failures. The potential of the NN is also this, to study and design new models that optimize and improve the performance of some tasks, we will also study the tolerance limit of the network after which the model has completely broken its functioning.

The second part of the network will allow the exploration of methods and technologies that make the model more tolerable and resilient, trying to increase the tolerability limit, this phase opens the way to a vastness of future work to create models of NN that could open the way for the use of this new science in a context

such as space. Opening the way to infinite scenarios can lead devices to be more autonomous and improve the experience of space exploration by allowing us to know new boundaries and new discoveries to enrich our knowledge of the universe around us.

Chapter 2

The Problem

2.1 Radiation in deep space

If needed precaution in the form of shielding is not taken for space instruments, the charged particle striking a sensitive node of the circuit could strike transient or permanent damage.

These serious effects on electronic circuits can be classified under a phenomenon called “Single Effect Event (SEE) which can be further divided into subcategories as follows:

- *Single Event Upset (SEU)*: A transient effect, affecting mainly memories
- *Single Event Transient (SET)*: A temporary pulse traverses the circuit. Nothing can be done about it
- *Single Event Latchup (SEL)*: Can destroy the component, affecting mainly CMOS
- *Single Event Gate Rupture (SEGR)*: Potentially destructive, affecting sub-micron structure
- *Single Effect Breakdown (SEB)*: Has destructive impact, affecting mainly power MOSFET

Some of these effects are only disruptive, such as SEU and SET and some others can permanently damage a circuit such as SEL, SEGR and SEB.

Circuit quality control testing is required before the launching of space instruments to find out where the vulnerable nodes are and also to protect against the charged particle strike.

A charged particle can strike an electronic circuit and creating electron-hole pairs along its path, as shown in Fig. 2.1.

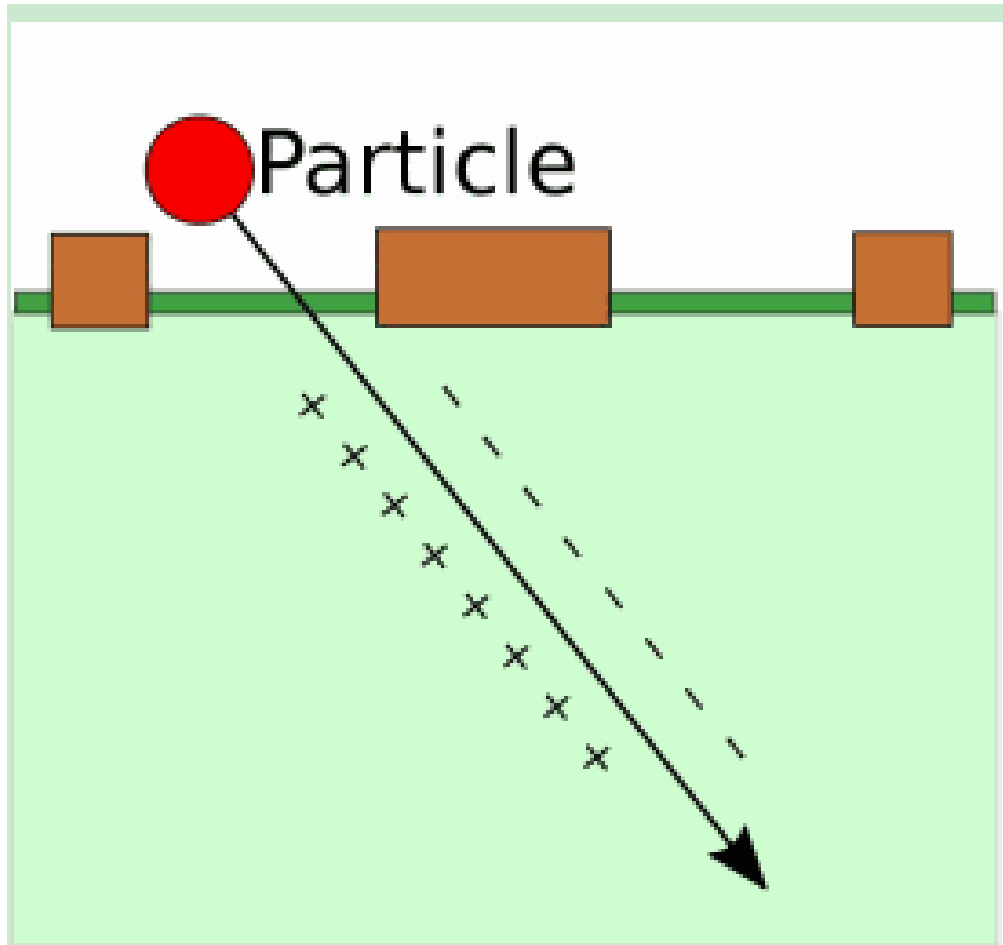


Figure 2.1: Charged particle strike an integrated circuit

When the charged particle hits sequential elements such as shift registers can be disrupted by SEU causing a **bit-flip** (shown in red) and combinational elements such as AND/OR gates can be affected by a SET which can upset another sequential element further ahead, as shown in Fig. 2.2.

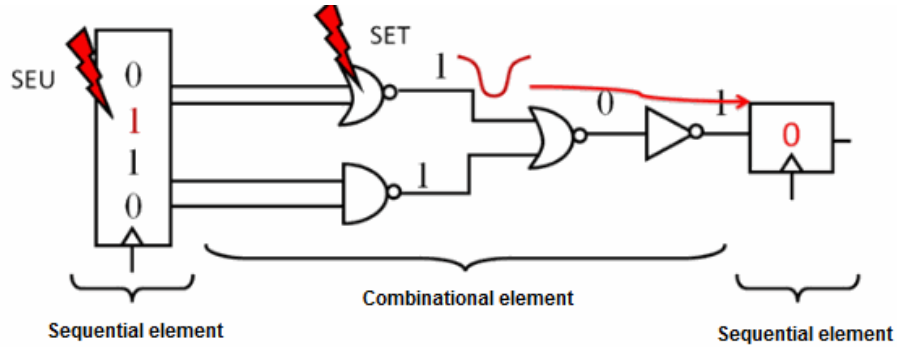


Figure 2.2: soft error effect in an integrated circuit

When it comes to protecting these charged particles, the integrated circuits can be hard-coated or can be implemented substrates additional such as Silicon on Sapphire (SOS) or Silicon On Insulator (SOI). However, since some nodes in the circuit are more sensitive than others, testing needs to be done before the launch of the space instrument to find out exactly where the vulnerable nodes are or to verify if the implemented radiation hardening has been strong enough to protect the integrated circuit against the striking radiation.

2.1.1 Single-event upset

Single-event upset (SEU) or transient radiation effects in electronics are state changes of memory or register bits caused by a single ion interacting with the chip. They do not cause permanent damage to the device but may cause lasting problems to a system that cannot recover from such an error.

In very vulnerable devices, a single ion can cause a *multiple-bit upset (MBU)* in adjacent memory cells.

SEUs can become *Single-event functional interrupts (SEFI)* when they upset control circuits, such as state machines, placing the device into an undefined state, a test mode, or a halt, which would then need a reset or a power cycle to recover.

2.1.2 Physical hardening

This means using different materials, for example, insulating substrates such as silicon or sapphire. Various approaches suggest shielding the circuit and alternative doping mechanisms. Circuit based hardening: this involves adding extra circuitry/logic to correct for the effects of SEEs. These include watchdog protection, over-current circuits, power control and error-correcting circuits (ex. CRC and forward error correction in communication boards). Error correction is implemented both at the hardware and software-levels, such as in the main memory where hardware-level ECC and software EDAC are applied in synergy.[1]

2.2 Radiation in AI-based device

The research is complete of studies on the effects of transient errors in devices. But our investigation goes furthermore, studying how these transient errors affect an NN application on a device that operates in space.

NNs are generally assumed to be tolerant against faults and imprecision to a particular degree due to their distributed structures and redundancy. In other words, NNs can be robust to noisy inputs and prove a low sensitivity to the faults, and therefore, the outcome of the computation is not drastically affected. However, a note of attention is needed here. Being fault-tolerant highly depends on not only the architecture and size of the NN, but also the training that the NN knows. For instance, small size NNs might not be fault-tolerant. Therefore, it cannot be claimed that NNs are in general fault-tolerant.

Furthermore, the robustness and the potential fault-tolerant parts of neural models invite for attention as permanent and transient faults, device variation, thermal issues, and ageing will force designers to abandon current assumptions that transistors, wires, and other circuit elements will perform perfectly over the entire lifetime of a computing system.

This work is relevant to investigate how faults/errors will affect the performance of hardware neural networks and whether the faults/errors can be mitigated by leveraging the intrinsic features of neural networks with complementary techniques. This analysis can be conducted on different levels of abstractions, from very specific low-level physical implementations to the high-level intrinsic fault-masking capacity

of neural paradigms. My analysis uses a high-level approach.

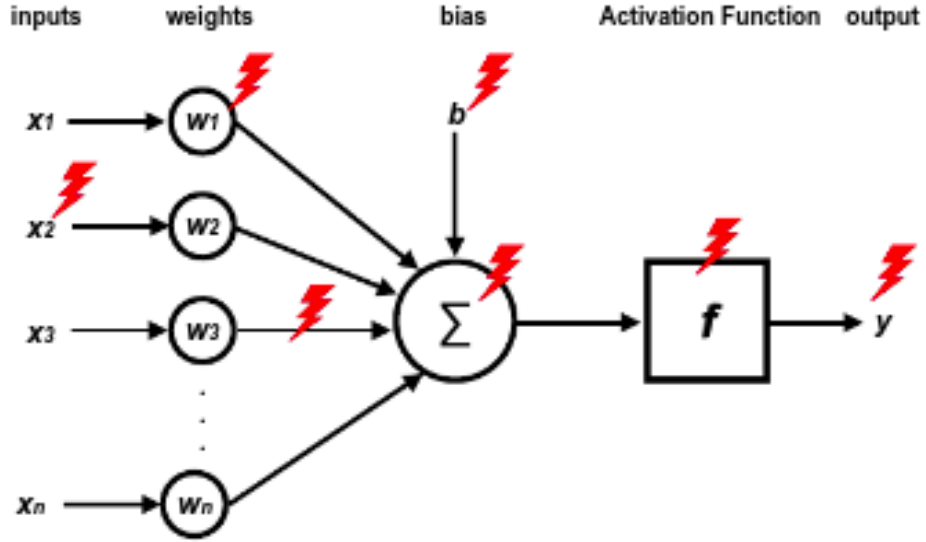


Figure 2.3: The vulnerable components and parameters of a single neuron to fault injection

Some papers, such as [2], [3], [4], study the vulnerable components of the network through low and high level injection mechanisms.

I have summarized in fig.2.3, the possible vulnerable components of a single artificial neuron. All translate by altering the output of the calculation. The errors can be different, they can for example have an error in the input of the first level or an error in an intrinsic computation component of the network. But my analysis focuses on the errors in the network parameters (weights and biases) which form the basis for the network to perform the task for which it was designed.

This decision is the result of a lot of reasoning, I want to test the tolerance of the overall NN and not simply the tolerance of some components. Fault simulation analysis could then simulate the error behavior of all possible vulnerable components.

Furthermore, the scope of my work focuses on the analysis of cumulative errors in the NN.

2.2.1 Single Event Upset in Neural Networks

[5] show the probability of at least one bit flip for all the parameters in one Neural Network, assuming that the flip of each bit is independent. Equation is:

$$\begin{aligned}
 P_{flip} &= 1 - ((1 - P_{single})^{NW})^{\frac{T}{t}} \\
 &= \sum_{n=0}^{n-1} \binom{n}{i} (-1)^{n-i+1} P_{single}^{n-i}, n = NxWx\frac{T}{t} \\
 &\simeq NxWx\frac{T}{t} P_{single}
 \end{aligned}$$

in which N is the number of trained parameters, W is the data width of each parameter, T is the device lifetime, t is the test time interval, and P_{single} is the probability for one bit-flip within t . The probability of P_{single} depends on which region of space the device will cross.

Chapter 3

Technologies

3.1 OVERVIEW OF DNN-S

DNNs come in an ample diversity of shapes and sizes depending on the application. The popular shapes and sizes are also growing rapidly to increase accuracy and efficiency. In all cases, the input to a DNN is a set of values representing the information to be examined by the network. For example, these inputs can be pixels of an image, sampled amplitudes of an audio wave or the numerical representation of the state of some system or game.

Neural networks are a set of algorithms based on the functioning of the human brain. Usually, what you see with your eyes is called data and is processed by the neurons in our brain, and identifies what is around you (data processing). In a comparable way, Neural Networks takes a large set of data, process the data (draws out the patterns from data), and outputs what it is.

Neural networks sometimes called Artificial Neural networks(ANN's), because they artificially simulate the nature and functioning of the human brain. ANN's are constituted of a large number of highly interconnected processing elements (neurones) working in cooperation to solve specific problems. An artificial neural network consists of three groups of units (layers): a layer of "input" units is connected to a layer of "hidden" units (where the hidden units are free to create their own representations of the input), which is connected to a layer of "output" units, as shown in fig 3.1.

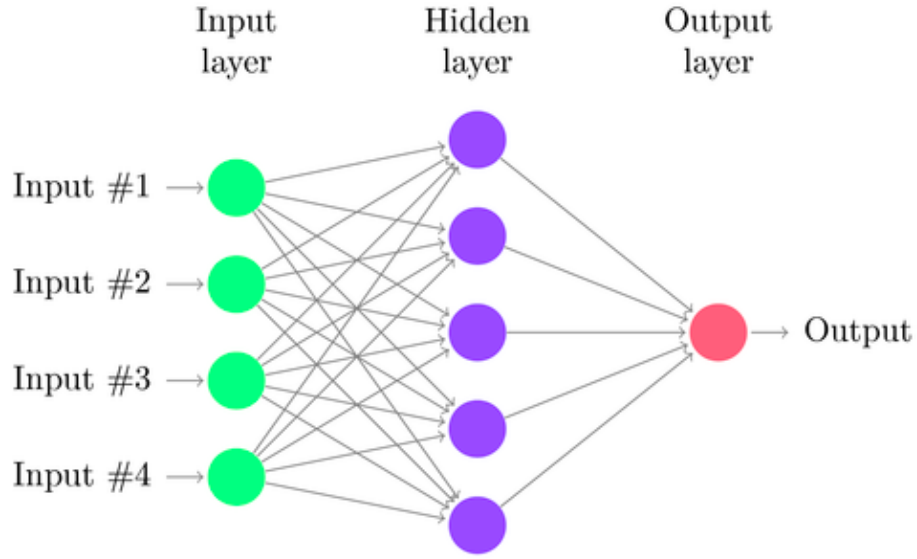


Figure 3.1: Artificial Neural networks

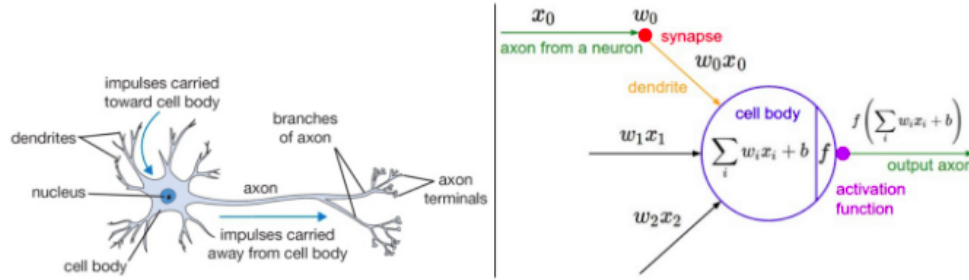


Figure 3.2: Artificial Neuron

3.1.1 Artificial Neuron

Before we examine Deep Neural Network let's look into a single artificial neuron. The neuron is the basic unit of computation in a neural network. Each input has a connected weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function f (defined below) to the weighted sum of its inputs as in fig. 3.2. It is represented by a mathematical function. It takes i -inputs x , and each of them usually has its own weight w . The neuron calculates the sum and it is passed through the activation function to the network further.

According to this model, the mathematical formula for the output of this neuron is:

$$\begin{cases} z = b + \sum_j x_j w_j \\ \text{output} = \sigma(z) \end{cases} \quad (3.1)$$

3.1.2 Convolutional Neural Networks (CNNs)

A common form of DNNs is Convolutional Neural Nets (CNNs), which are composed of multiple CONV layers as shown in Fig. 3.3. In such networks, each layer generates a successively higher-level abstraction of the input data, called a feature map (fmap), which maintains essential yet unique information. Modern CNNs are able to achieve excellent performance by operating a very deep hierarchy of layers. CNNs are widely used in a variety of applications including image recognition, speech recognition, gameplay, robotics, etc. This activity will focus on its use in image processing, specifically for the task of image classification. Convolutional Neural Networks are very similar to ordinary Neural Networks, but with a difference, CNN architectures make the assumption that the inputs are images, which allows us to encode some properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network.

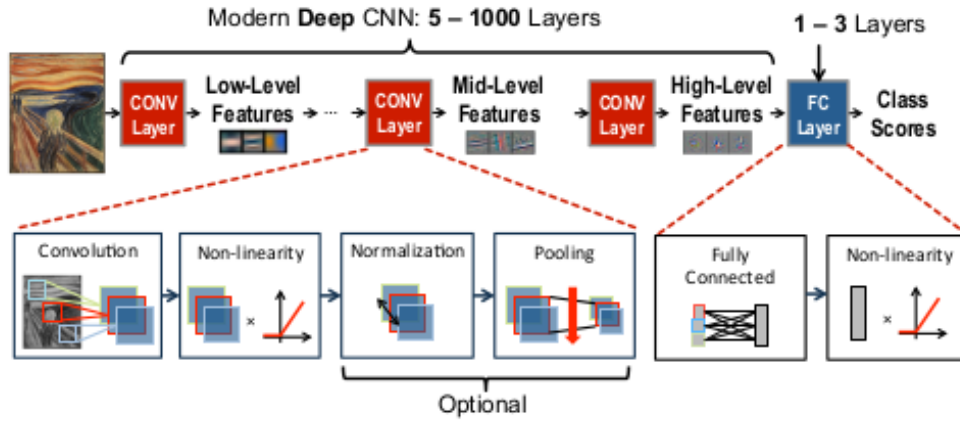


Figure 3.3: Convolutional Neural Networks

Each of the CONV layers in CNN is primarily composed of high-dimensional

convolutions, as shown in Fig. 3.4. In this convolution, the input activations of a layer are structured as a set of 2D input feature maps (ifmaps), each of which is called a channel. Each channel is convolved with a distinct 2D filter from the stack of filters, one for each channel, this stack of 2D filters is often referred to as a single 3D filter. The results of the convolution at each point are summed across all the channels.

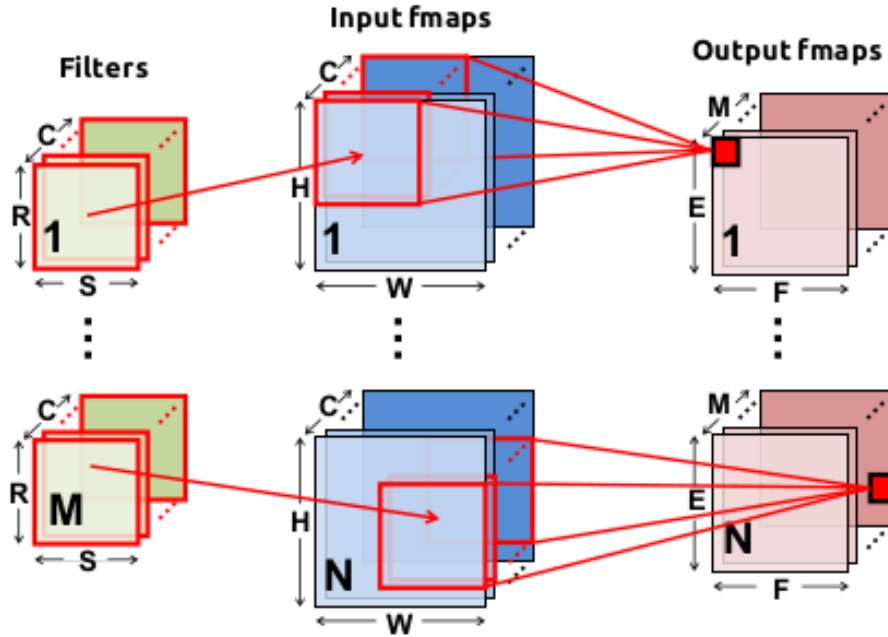


Figure 3.4: Convolutions in CNNs

A simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet architecture. Let's see in more detail how the level values work.

Depth-wise convolution

The standard convolution layer of a neural network involves $input * output * width * height$ parameters, where width and height are width and height of filter. For an

input channel of 10 and output of 20 with 7×7 filter, this will have 2800 parameters. Having so many parameters increases the chance of over-fitting. To avoid this, have been devised: *Depth-wise convolution* and *depth-wise separable convolution*.

To understand how it differs from the standard convolution, let's take an example: Suppose our input tensor is $3 \times 8 \times 8$ (*input = channel \times width \times height*). Filter is $3 \times 3 \times 3$. In a standard convolution, we would directly convolve in-depth dimension as well, as shown in Fig. 3.5.

Indeed, in depth-wise convolution, we use each filter channel only at one input channel. In the example, we have 3 channel filter and 3 channel image. Depth-wise convolution breaks the filter and image into three different channels and then convolve the corresponding image with the corresponding channel and then stack them back, as shown in Fig. 3.6.

To produce the same effect with normal convolution, Depth-wise convolution selects a channel, make all the elements zero in the filter except that channel and then convolve. We will need three different filters one for each channel. Although parameters are remaining the same, this convolution gives you three output channels with only one 3-channel filter while you would require three 3-channel filters if you would use normal convolution.

Depth-wise Separable Convolution

Depth-wise separable convolution uses depth-wise convolution and after that we use a 1×1 filter to cover the depth dimension, as show in Fig. 3.7.

One thing to notice is, how many parameters are reduced by this convolution to output the same numbers of channels. To produce one channel we need $3 \times 3 \times 3$ parameters to perform depth-wise convolution and 1×3 parameters to perform further convolution in-depth dimension. But, If we need 3 output channels, we only need 3 1×3 depth filters giving us a total of 36 ($= 27 + 9$) parameters while for the same numbers of output channels in normal convolution, we need 3 $3 \times 3 \times 3$ filters giving us a total of 81 parameters. Having too many parameters forces function to memorize lather than learn and thus over-fitting. Depth-wise separable convolution saves us from that.

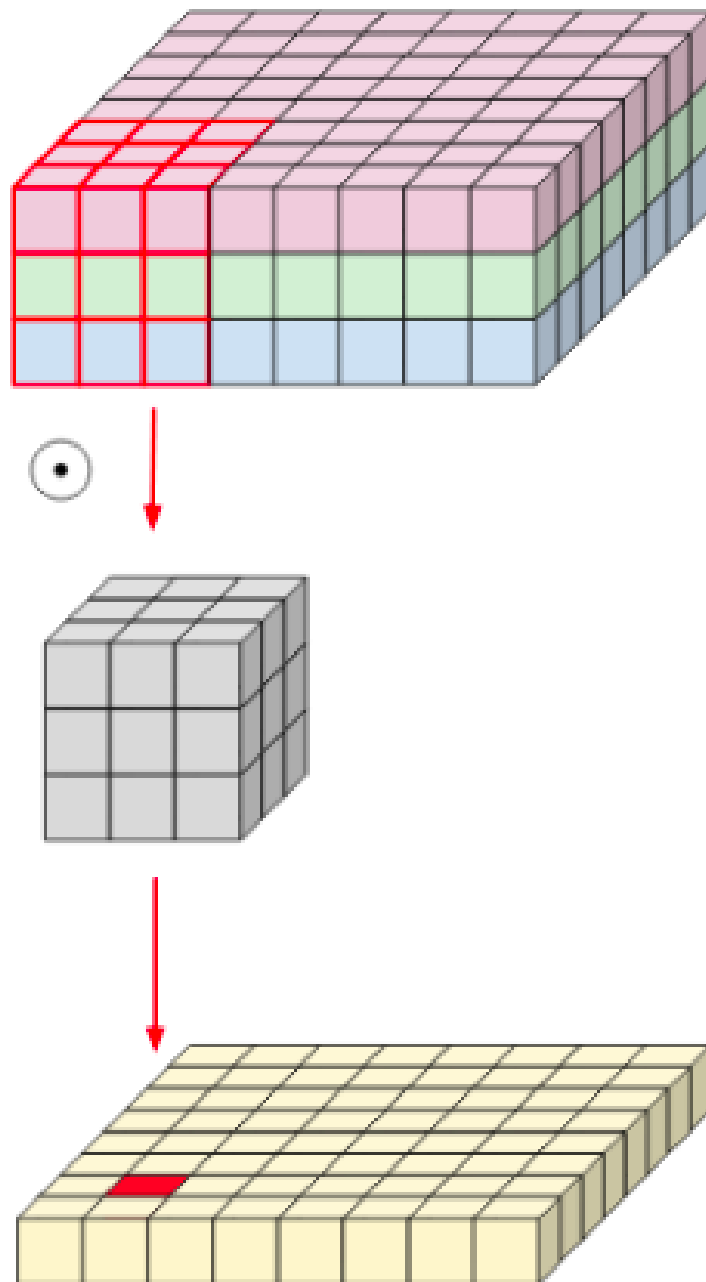


Figure 3.5: Standard convolution

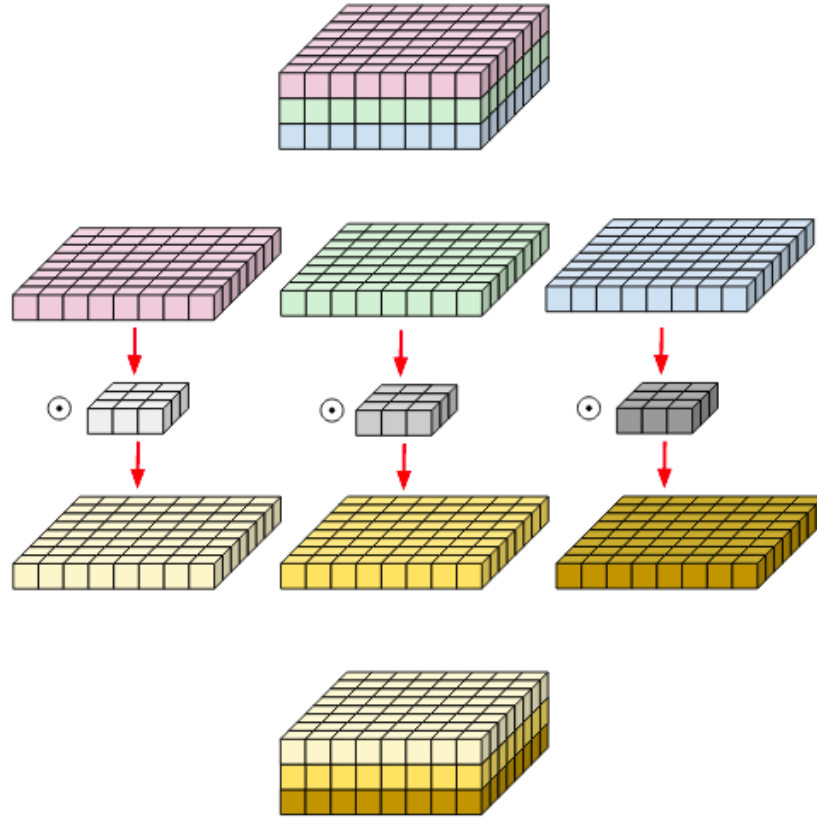


Figure 3.6: depth-wise convolution

3.1.3 Convolutional layers

In particular, the layers of a ConvNet have neurons arranged in 3 dimensions: **width, height, depth**, as shown in fig. 3.8. The Conv layer is the core architecture block of a Convolutional Network that does most of the computational work. The main idea is to connect each neuron to only a local region of the input volume. (a hyperparameter called the receptive field of the neuron), this is equivalent at the **filter size**. The size of the connectivity along the depth axis is always equal to the depth of the input volume. Instead, three hyperparameters control the size of the output volume: depth, stride and zero-padding.

- **Depth:** of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something

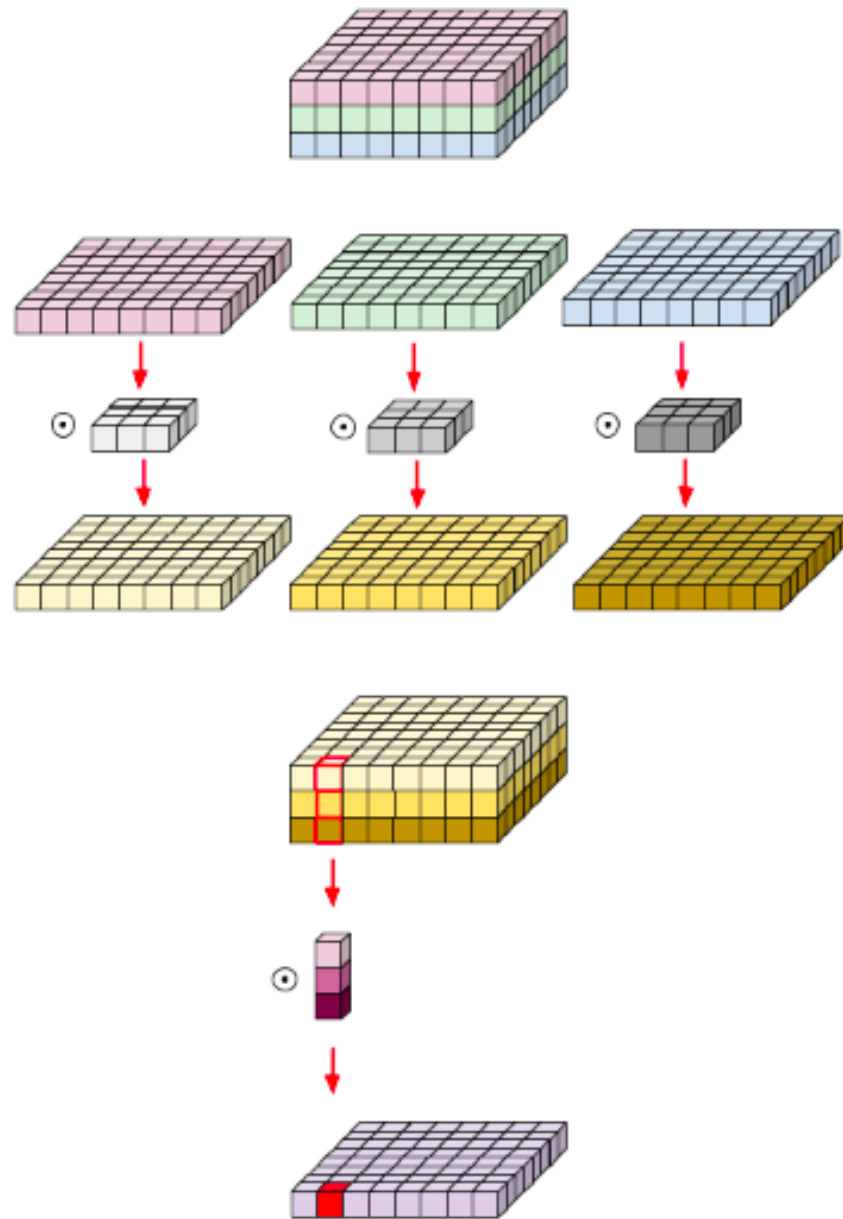


Figure 3.7: Depth-wise separable convolution

different in the input.

- **Stride:** with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. Result is smaller output volumes. spatially.

- **Zero-padding:** it will allow us to control the spatial size of the output volumes.

We can calculate the spatial size of the output volume as a function of the input volume size (W), the receptive field size of the Conv Layer neurons (F), the stride with which they are applied (S), and the number of zero paddings used (P) on the border. The correct formula for calculating how many neurons “fit” is given by $(WF + 2P)/S + 1$. [6]

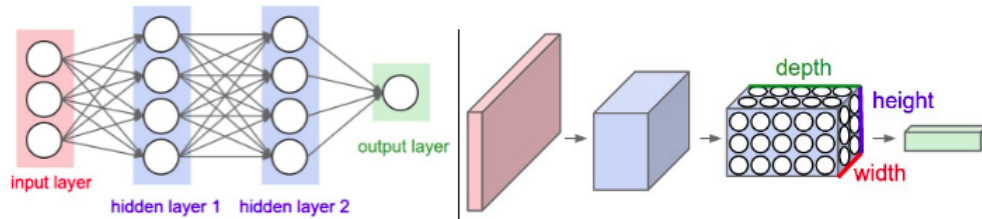


Figure 3.8: ConvNet arranges its neurons in three dimensions

3.1.4 Pooling Layer

A pooling layer is usually inserted between convolutional layers to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation and the depth dimension remains unchanged. An example of workflow can be seen in fig 3.9.

3.1.5 Fully Connected Layers

These layers are more possible generic: given S neurons and N inputs, all inputs are found to all neurons, the result is S output. Each neuron associated with N weights plus a bias because this is the number of inputs is received.

The weight associated with the inputs of each neuron obtained through learning techniques such as gradient descend, a cost function is used to minimize the output values error.

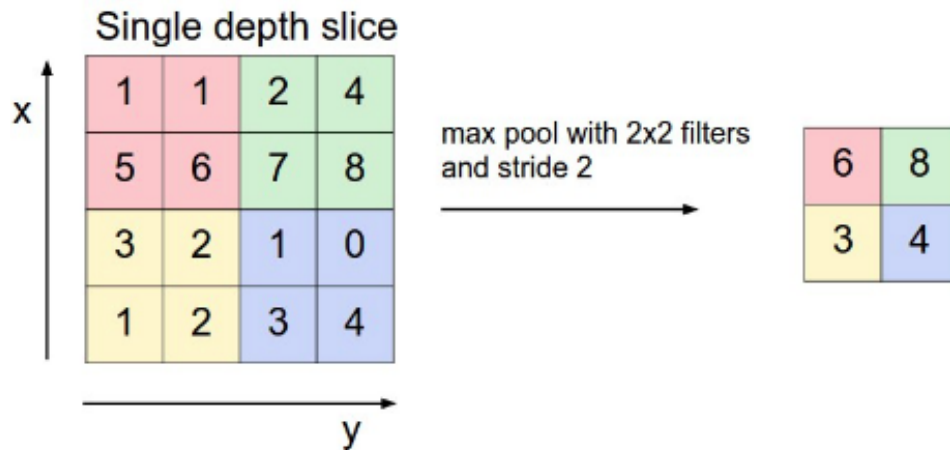


Figure 3.9: Pooling layer down-samples the volume spatially. The most common down-sampling operation is max, giving rise to max pooling.

3.1.6 Popular DNN Models

Numerous DNN models have been developed over the past two decades. Each of these models has a different ‘network architecture’ in terms of the number of layers, layer types, layer shapes (filter size, number of channels and filters), and connections between layers. Learning these variations and trends is important for consolidating the right flexibility in any efficient DNN engine.

- **Alexnet** was the first CNN to win the ImageNet Challenge in 2012. It consists of five CONV layers and three FC layers. Alexnet has a vast number of weights and the shapes vary from layer to layer. In the first layer, the 3 channels of the filter correspond to the red, green and blue components of the input image. A ReLU non-linearity is used in each layer. Max pooling of 3×3 is involved in the outputs of layers 1, 2 and 5. To reduce computation, a stride of 4 is used at the first layer of the network. Within each CONV layer, there are 96 to 384 filters and the filter size ranges from 3×3 to 11×11, with 3 to 256 channels each. In total, AlexNet requires 61M weights and 724M MACs to process one 227×227 input image. [7]
- **Resnet** (also known as Residual Net) utilises residual connections to go even deeper (34 layers or more). It was the first entry DNN in ImageNet Challenge

that exceeded human-level accuracy with a top-5 error rate below 5%. One of the challenges with deep networks is the vanishing gradient during training: as the error backpropagates through the network the gradient shrinks, which affects the ability to update the weights in the earlier layers for very deep networks. Residual net proposes a "shortcut" module that includes an identity connection such that the weight layers (ex. CONV layers) can be skipped, as shown in Fig. 3.12. Rather than learning the function for the weight layers $F(x)$, the shortcut module discovers the residual mapping ($F(x) = H(x)x$). Initially, $F(x)$ is zero and the identity connection is taken, but then gradually during training, the actual forward connection through the weight layer is used. ResNet also uses the "bottleneck" approach of using 1×1 filters to reduce the number of weight parameters. As a consequence, the two layers in the shortcut module are replaced by three layers (1Ö1, 3Ö3, 1Ö1) where the 1Ö1 reduces and then restores the number of weights. ResNet-50 consists of one CONV layer, followed by 16 shortcut layers (each of which are three CONV layers deep), and one FC layer. It requires 25.5M weights and 3.9G MACs per image. [8]

- **Mobilenet:** is based on exploring an automated neural architecture search approach for designing mobile models using reinforcement learning. To deal with mobile speed constraints, this approach explicitly combines the speed information into the main reward function of the search algorithm, so that the search can identify a model that performs a good trade-off between accuracy and speed (Pareto optimal solutions¹).

To strike the right balance between search flexibility and search space size, mnasnet is proposed as a novel factorized hierarchical search space, which factorizes a convolutional neural network into a sequence of blocks and then uses a hierarchical search space to determine the layered architecture for each block. In this way, this approach allows different layers to use different operations and connections. Meanwhile, this approach forces all layers in each block to share the same structure, thus significantly reducing the search space

¹This is a situation where no individual or preference criterion can be better off without making at least one individual or preference criterion worse off or without any loss thereof

size by orders of magnitude compared to flat per-layer search space. The network architecture is shown in Fig. 3.10.[9]

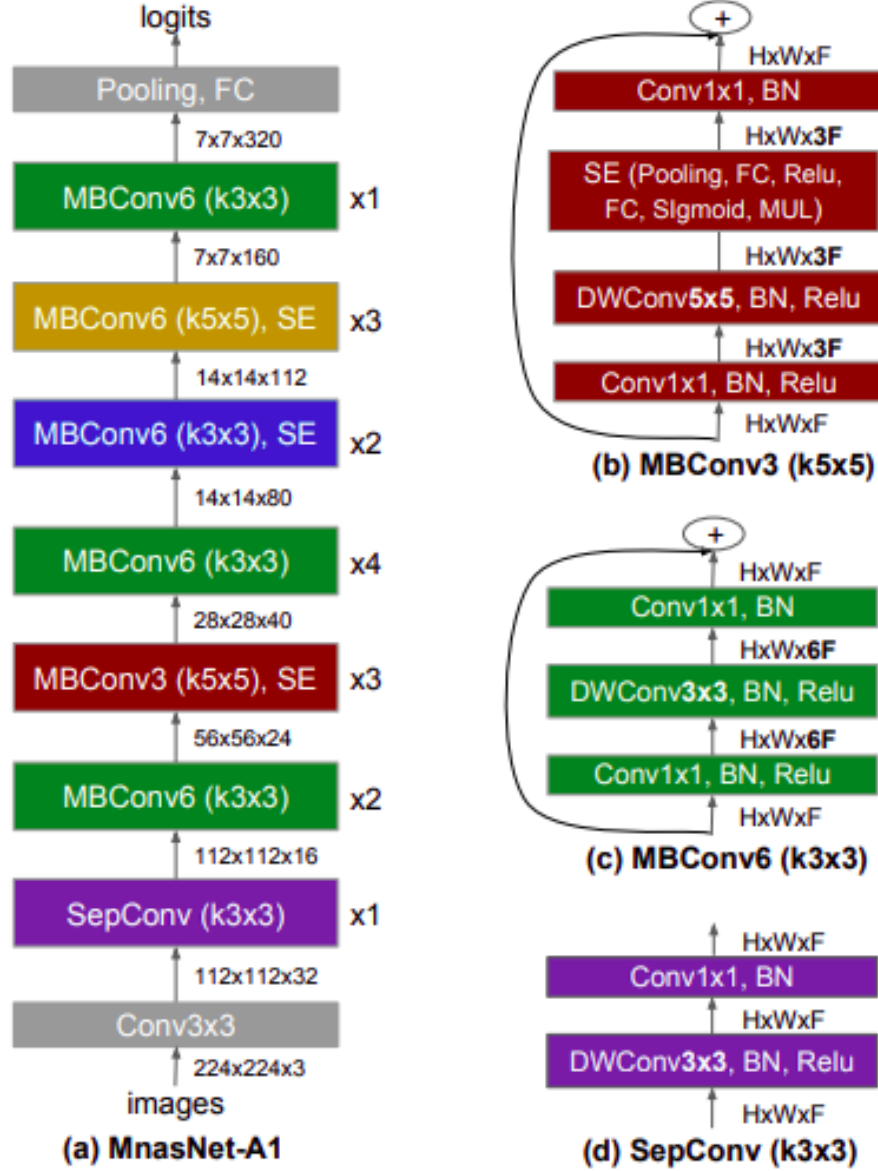


Figure 3.10: Example of MnasNet Architecture – (a) is a representative model; (b) - (d) are a few corresponding layer structures. MBConv denotes mobile inverted bottleneck conv, DWConv denotes depthwise conv, $k3x3/k5x5$ denotes kernel size, BN is batch norm, $HxWxF$ denotes tensor shape (height, width, depth), and $\times 1/2/3/4$ denotes the number of repeated layers within the block.

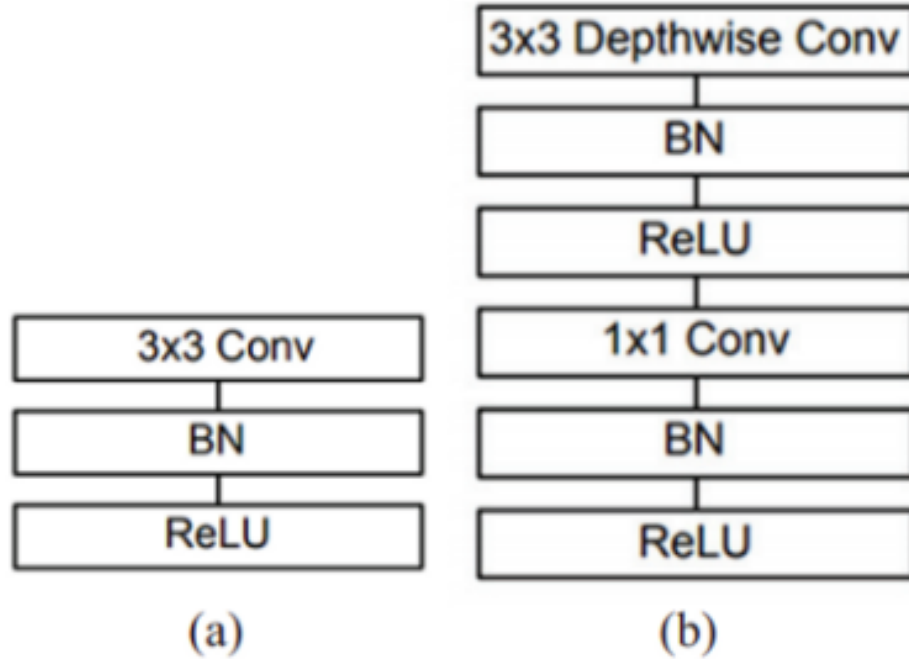


Figure 3.11: (a) Standard convolutional layer with batch normalization and ReLU. (b) Depth-wise separable convolution with depth-wise layers followed by batch normalization and ReLU.

3.2 DNN DEVELOPMENT RESOURCES

One of the key factors that have allowed the rapid expansion of DNNs is the set of development resources that have been made available by the research community and industry. These resources are also key to the development of DNN accelerators by providing characterizations of the workloads and promoting the exploration of trade-offs in model complexity and accuracy. This section will explain these resources such that those who are involved in this field can quickly get started.

3.2.1 Frameworks

For the prosperity of DNN development and to permit sharing of trained networks, some deep learning frameworks are developed from various sources. These open-source libraries include software libraries for DNNs. Caffe was made possible in

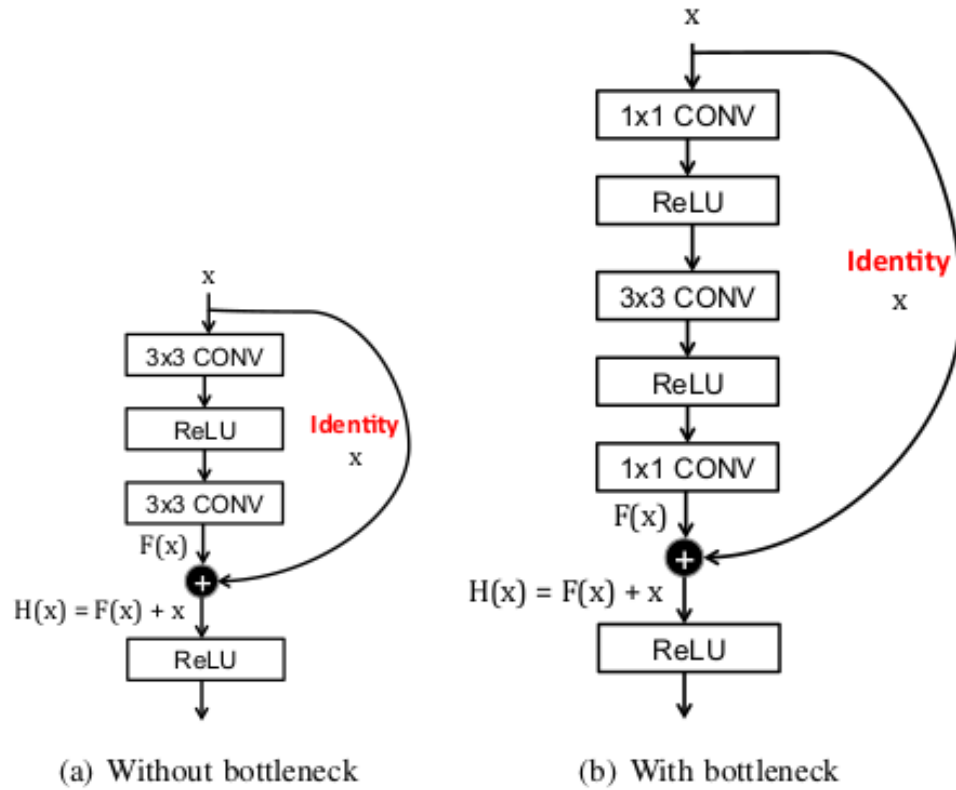


Figure 3.12: Shortcut module from ResNet

2014 from UC Berkeley. It supports C, C++, Python and MATLAB. Tensorflow was released by Google in 2015, and supports C++ and Python; it also carries multiple CPUs and GPUs and has more flexibility than Caffe, with the computation expressed as dataflow graphs to attain the tensors (multidimensional arrays). Another popular framework is Torch, which was released by Facebook and NYU and supports C, C++ and Lua. There are higher-level libraries that may run on top of the aforementioned frameworks to supply a more universal experience and faster development. One example of such libraries is Keras, which is written in Python and supports Tensorflow, CNTK and Theano.

The presence of such frameworks don't seem to be only a useful aid for DNN researchers and application designers, but they're also precious for engineering high performance or more efficient DNN computation engines. specifically, because

the frameworks make heavy use of set primitive operations, like processing of a CONV layer, they'll incorporate the employment of optimized software or hardware accelerators. This acceleration is transparent to the user of the framework. Thus, as an example, most frameworks can use Nvidia's cuDNN library for rapid execution on Nvidia GPUs.

Pytorch

PyTorch could be a library for Python programs that facilitates building deep learning projects. It emphasizes flexibility and allows deep learning models to be expressed in idiomatic Python. This approachability and easy use found early adopters within the research community, and within the years since its first release, it's grown into one in every of the most prominent deep learning tools across a broad range of applications.

As Python does for programming, PyTorch provides a wonderful introduction to deep learning. At the identical time, PyTorch has been proven to be fully qualified to be used in professional contexts for real-world, high-profile work. We believe that PyTorch's clear syntax, streamlined API, and simple debugging make it a wonderful choice for introducing deep learning.

At its core, the deep learning machine may be a rather complex function mapping inputs to an output. To facilitate expressing this function, PyTorch provides a core system, the tensor, which could be a multidimensional array that shares many similarities with NumPy arrays. Around that foundation, PyTorch comes with features to perform accelerated mathematical operations on dedicated hardware, which makes it convenient to style neural network architectures and train them on individual machines or parallel computing resources.

Why Pytorch?

Output Rephrased/Re-written Text More concretely, programming the deep learning machine is extremely natural in PyTorch. PyTorch gives us an information type, the Tensor, to carry numbers, vectors, matrices, or arrays normally. additionally, it provides functions for operating on them. We can program with them incrementally and, if we want, interactively, similar to we are accustomed from Python.

PyTorch offers two things that make it particularly relevant for deep learning: first, it provides accelerated computation using graphical processing units (GPUs), often yielding speedups within the range of 50x over doing the identical calculation on a CPU. Second, PyTorch provides facilities that support numerical optimization on generic mathematical expressions, which deep learning uses for training. Note that both features are useful for scientific computing generally, not exclusively for deep learning. In fact, we are able to safely characterize PyTorch as a high-performance library with optimization support for scientific computing in Python.

Jupyter Notebook

The Jupyter Notebook is an open source web application that you simply can use to make and share documents that contain live code, equations, visualizations, and text.

A Jupyter Notebook shows itself as a page within the browser through which we are able to run code interactively. The code is evaluated by a kernel, a process running on a server that's able to receive code to execute and remand the results, which are then rendered inline on the page. A notebook maintains the state of the kernel, like variables defined during the evaluation of code, in memory until it's terminated or restarted. The elemental unit with which we interact with a notebook may be a cell: a box on the page where we will type code and have the kernel evaluate it (through the menu item or by pressing Shift-Enter). We are able to add multiple cells in an exceedingly notebook, and therefore the new cells will see the variables we created within the earlier cells. The worth returned by the last line of a cell are going to be printed right below the cell after execution, and also the same goes for plots. By mixing ASCII text file, results of evaluations, and Markdown-formatted text cells, we are able to generate beautiful interactive documents. **Colaboratory** is a free Jupyter notebook environment that needs no setup and runs entirely within the cloud. With Colaboratory you'll write and execute code, save and share your analyses, and access powerful computing resources, all free from your browser.

Google Colab comes with collaboration backed within the product. In fact, it's a Jupyter notebook that leverages Google Docs collaboration features. It also runs on Google servers and you don't have to install anything. Moreover, the notebooks are saved to your Google Drive account.

3.2.2 Models

Pre-trained DNN models can be downloaded from various websites for the various different frameworks. It should be noted that even for the same DNN the accuracy of these models can vary by around 1% to 2% depending on how the model was trained.

Pytorch Model Zoo

This lists model archives that are pre-trained and pre-packaged, ready to be served for inference with TorchServe.

The models subpackage contains definitions of models for addressing different tasks, including: image classification, pixelwise semantic segmentation, object detection, instance segmentation, person keypoint detection and video classification.

3.2.3 Datasets for Classification

Imagenet : is a very large dataset of over 14 million images managed by Stanford University. All of the images are labeled with a hierarchy of nouns that come from the WordNet dataset (<http://wordnet.princeton.edu>), which is in turn a large lexical database of the English language. A subset of images is shown in Fig. 3.13.



Figure 3.13: Imagenet dataset

The accuracy of the ImageNet Challenge are described using two metrics: **Top-5** and **Top-1** error. Top-5 error means that if any of the top five scoring categories

are the correct category, it is counted as a correct classification. The Top-1 wants that the top-scoring category is correct.

3.2.4 Dataset in my analysis

I mainly used a test set, called Imagenette, because, I wanted a small vision dataset, I could use to quickly see if my ideas might have a chance of working. They normally don't, but testing them on Imagenet takes a long time for me to find that out, especially because **I'm interested in algorithms that perform particularly well at the end of training.**

- **Imagenette** is a subset of 10 easily classified classes from Imagenet (tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute).
- **Imagewoof** is a subset of 10 classes from Imagenet that aren't so easy to classify, since they're all dog breeds. The breeds are: Australian terrier, Border terrier, Samoyed, Beagle, Shih-Tzu, English foxhound, Rhodesian ridgeback, Dingo, Golden retriever, Old English sheepdog.

3.2.5 Training

Since DNNs are an instance of a machine learning algorithm, the essential application doesn't change because it learns to perform its given tasks. Within the specific case of DNNs, this learning requires learning the worth of the weights (and bias) within the NN and is spoken as training the network. Once trained, the program can perform its task by computing the output of the network using the weights determined during the training process. Running the program with these weights is noted as inference.

In this section, we are going to use image classification, as shown in Fig. 3.14, as a driving example for training and employing a DNN. When we perform inference employing a DNN, we give an input image and therefore the output of the DNN could be a vector of scores, one for each object class; the category with the best score shows the most likely class of object within the image.

The overarching goal for training a DNN is to work out the weights that maximize

the score of the right class and reduce the score of the wrong classes. Generally, when training the network the right class is usually known because it's given for the pictures used for the training set.

Furthermore, the gap between the perfect correct scores and also the scores computed by the DNN supported its *current weights* id said because of the loss(L). The goal of coaching DNNs is to search out a group of weights to attenuate the typical loss over an outsized training set.

The process to update weights is named gradient descent. A multiple of the gradient of the loss relative weight, which is that the derivative of the loss with relevance to the load, is employed to update the burden. This process indicates how the burden should change to cut back the loss. The process is repeated iteratively to scale back the loss.

An efficient thanks to computing partial derivates of the gradient is through a process called *backpropagation*, that operates by passing values backwards through the network to the computer how the loss is suffering from each weight.

Backpropagation requires intermediate outputs of the network to be preserved for the backwards computation, thus training has increased storage requirements. Second, thanks to the gradients use for hill-climbing, the precision requirement for training is generally over inference.

A variety of techniques are accustomed to improve the efficiency and robustness of coaching. for instance, often the loss from multiple sets of input files, a batch, are collected before a a single pass of weight update is performed; this helps to hurry up and stabilize the training process.

There are multiple ways to coach the weights. The most common approach, as described above, is termed *supervised learning*, where all the training samples are labelled (with the true class).

Unsupervised learning is another approach where all the training samples don't seem to be labelled and essentially the goal is to seek out the structure of clusters within the data. Semi-supervised learning falls in between the 2 approaches where only a tiny low subset of the training data is labelled (use unlabeled data to define the cluster boundaries and use the small volume of labelled data to label the clusters). Finally, *reinforcement learning* will be accustomed to the train weights such that given the state of this environment, the DNN can output what action the agent

should take next to maximise expected rewards; however, the rewards may not be available immediately after an action, but instead only after a series of actions. Another usually used approach to work out weights is fine-tuning, where previously-trained weights are available and are used as a start line so those weights are adjusted for a replacement dataset (ex. transfer learning) or a replacement constraint (ex. reduced precision). This leads to faster training than starting from a random place to begin, and might sometimes result in better accuracy.

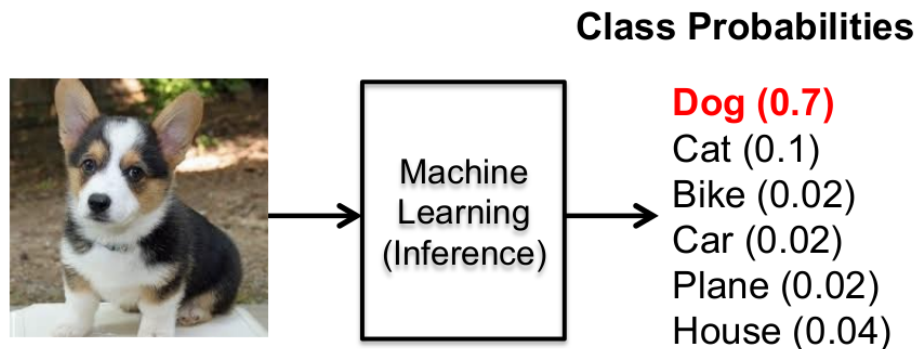


Figure 3.14: The confidence scores for a image

3.2.6 Inference

Inference applies knowledge from a **trained neural network model** and uses it to compute a result. So, given a replacement unknown data set as input through a trained neural network, it outputs a prediction supported predictive accuracy of the neural network.

In practice, the utilization of CNNs consists of two main tasks: training and inference. Training is that the process of “learning” the optimal set of weights that maximize the accuracy of the wanted task (ex. image classification, object detection, semantic segmentation).

Training may be a highly compute-intensive process often accelerated by GPUs. The inference is the process of employing a trained model (where parameters aren’t any longer modified) to create decisions on novel data. The inference could be a less compute-intensive process than training and has been performed on CPUs,

GPUs, and FPGAs.

Deep learning inference networks became a standard part of terrestrial systems today, large continuously trained models generally reside inside server clusters. Increasingly however frozen inference models are being deployed onto mobile and embedded devices.

DNN models created for image classification, tongue processing, and other AI jobs are large and sophisticated, with dozens or many layers of artificial neurons and millions or billions of weights connecting them. The larger the DNN, the more compute, memory and energy are consumed to run it, and also the length is going to be the response time (or “latency”) from after you computer file to the DNN until you receive a result. But sometimes the utilization case requires that inference run in no time or at very low power.

For example, a self-driving car must be able to detect and react within milliseconds in order to avoid an accident. And a battery-operated drone designed to follow a target or land in your hand needs to be power-efficient to maximise flight time. In such cases, there is a request to simplify the DNN after training so as to cut back power and latency, even if this simplification leads to a small reduction in prediction accuracy.

3.2.7 Tensor

Floating-point numbers are the way a network deals with information,so we need a way to encode real-world data of the kind we want to process into something digestible by a network and then decode the output back to something we can understand and use for our purpose.

To this end, PyTorch introduces a fundamental data structure: the **tensor**. In the context of deep learning, tensors refer to the generalization of vectors and matrices to an arbitrary number of dimensions (or multidimensional array). PyTorch tensors have the ability to perform very fast operations on graphical processing units (GPUs) and also distribute operations on multiple devices or machines.

The `dtype` argument to tensor constructors specifies the numerical data (d) type that will be contained in the tensor. Possible values for the `dtype` argument:

- `torch.float32` or `torch.float` : 32-bit floating-point

- `torch.float64` or `torch.double` : 64-bit, double-precision floating-point
- `torch.float16` or `torch.half` : 16-bit, half-precision floating-point
- `torch.int8` : signed 8-bit integers
- `torch.uint8` : unsigned 8-bit integers
- `torch.int16` or `torch.short` : signed 16-bit integers
- `torch.int32` or `torch.int` : signed 32-bit integers
- `torch.int64` or `torch.long` : signed 64-bit integers
- `torch.bool` : Boolean

Computations occurring in neural networks are typically executed with 32-bit floating-point precision. Higher precision, like 64-bit, will not buy improvements in the accuracy of a model and will require more memory and computing time.

The 16-bit floating-point, the half-precision data type is not present natively in standard CPUs, but it is offered on modern GPUs. It is possible to switch to half-precision to decrease the footprint of a neural network model if needed, with a lesser impact on accuracy.

Neural networks take tensors as input and produce tensors as outputs. All operations within a neural network and during optimization are operations between tensors, and all parameters (for example, weights and biases) in a neural network are tensors.

3.3 HARDWARE FOR DNN PROCESSING

DNN training and inference are computation-intensive processes but in very different ways. Training needs high throughput, thus is most often carried out by GPUs, given their massive parallelism, simple control flow, and energy efficiency. It is common to batch hundreds of training inputs (for example images in a computer vision task, sentence sequences in an NLP task or spectrograms in a speech recognition task) and perform forward or backward propagation on them as one unit of data simultaneously to amortize the cost of loading the network weights from GPU

memory across many inputs.

For inference, however, the paramount performance goal is latency. To minimize the network's end-to-end response time, inference typically batches a much smaller number of inputs than training, as automated services relying on inference are required to respond in near real-time.

While training is mostly dominated by GPUs, there are several players in the inference hardware market.

3.3.1 Overview of platforms

We can give a brief overview of each architecture:

CPU: Most current Multi-core CPUs operate in the same way as single-processor CPUs, using the shared memory paradigm for communication, with synchronisation achieved via a shared cache (or core-to-core cache coherency protocol). Each core hosts one thread at a time, with a set of registers containing thread state, an ALU dedicated to the current thread (containing a number of functional units), and a large unit devoted to management and scheduling tasks, such as branch prediction, instruction ordering, speculative execution, and so-on.

GPU: The idea behind GPUs is to dedicate the maximum amount of silicon area as possible to ALUs, by removing all the scheduling logic and caches required to take advantage of instruction-level parallelism and reduce memory latency in CPUs. Instead, thread-level parallelism is employed to cover latency, with each GPU executing up to 1024 threads without delay. The threads execute in batches of 32 threads called warps, providing SIMD style parallelism, but with the flexibility to independently enable and disable each thread within a warp, allowing each thread to execute different parts of the program. However, this batching comes at a cost: the fewer threads within a warp that is active, the fewer parallel operations are executed per cycle. It's critical to minimise thread divergence, by making sure that every one thread take the identical branch of conditional statements, and execute loops an identical number of times.

FPGA: FPGAs don't have any fixed instruction-set architecture. Instead, they provide a fine-grain grid of bit-wise functional units, which can be composed to make any desired circuit or processor. Much of the FPGA area is truly dedicated

to the routing infrastructure, which allows functional units to be connected at run-time. Modern FPGAs also contain a variety of dedicated functional units, like DSP blocks containing multipliers, and RAM blocks. These chips are manufactured for general use with configurable logic blocks (CLBs) and programmable interconnects. This implies you'll program and reprogram FPGAs to perform numerous functions after they need to leave the manufacturer and are getting used within the field.

ASIC: for application-specific microcircuit. This computer circuit is aptly named since an ASIC microchip is intended and made for one particular application and doesn't allow you to re-program or modify it after it's produced. This suggests ASICs aren't intended for general use, you need to have ASICs created to your specifications for your product. ASICs are available in some differing kinds, including gate array, primary cell and custom designs. These types are differentiated from one another by the amount of customization they provide during the planning process.

3.3.2 Inference on HW

GPU: GPU accelerators have a peak higher throughput than a single-socket CPU server.

To help developers better support its hardware, Nvidia's cuDNN library provides a series of inference optimizations for GPUs. In small-batch scenarios, cuDNN improves on the matter of convolution algorithms not having the ability to parallelize enough threads to fill the GPU. The standard algorithm like pre-computed implicit GEMM (generalized matrix-matrix product) is optimized for giant output matrices, and its default parallelization strategy suffers from the matter of not having the ability to launch enough thread blocks as long as batch size could be a multiplicative consider one in every of the output matrix dimensions.

The latest versions of cuDNN updated this algorithm by splitting in a further dimension, which reduces the number of computations per thread block and enables the launching of significantly more blocks, increasing GPU occupancy and performance.

FPGA: FPGAs often produce better performance per watt of power consumption than GPUs, especially for sliding-window computations like convolution and pooling. This makes them particularly attractive to industry users who ultimately care more

about reducing costs for big-scale applications and also the ability to customize the inference architecture for a specific application.

Traditionally not competitive against GPUs on peak floating-point performance, the sphere of FPGA for DNN inference is improving fast.

ASIC: researchers and industrialists is investing heavily in ASICs (Application Specific Integrated Circuits) further, believing that an infatuated chip design would yield ultimately superior performance for one single variety of computational workload.

Google's TPU is one such example. TPU often delivers 15x to 30x faster inference than CPU or GPU, and even more per watt at a comparable cost level. Its outstanding inference performance originates from four, among others, major design optimizations: Int8 quantization, DNN-inference-specific CISC instruction set, massively parallel matrix processor, and minimal deterministic design. [10]

3.4 METRICS

3.4.1 Metrics for DNN Models

To evaluate the properties of a given DNN model, we should consider the following metrics:

- The *accuracy* of the model in terms of the top-5 error on dataset. (In case, type of data augmentation used)
- The *network architecture* of the model, including number of layers, filter size, number of filters and number of channels.
- The *number of weights* impact the storage requirement of the model.
- The *number of MACs* that need to be performed. It is somewhat indicative of the number of operations and potential throughput of the DNN.

3.4.2 Metrics for DNN Hardware

To measure the efficiency of the DNN hardware, we should consider the following additional metrics:

- The *power and energy consumption* of the design should be reported for various DNN models. The DNN model specifications should be provided including which layers and bit precision are supported by the hardware during measurement. In addition, the amount of off-chip accesses (ex DRAM accesses) should be included since it accounts for a significant portion of the system power. It can be reported in terms of the total amount of data that is read and written off-chip per inference.
- The *latency and throughput* should be reported in terms of the batch size and the actual run time for various DNN models, which accounts for mapping and memory bandwidth effects. This provides a more useful and informative metric than peak throughput.
- The *cost* of the chip depends on the area efficiency, which accounts for the size and type of memory (e.g., registers or SRAM) and the amount of control logic. It should be reported in terms of the core area in squared millimeters per multiplier along with process technology.

In terms of cost, different platforms will have different implementation-specific metrics. For instance, for an FPGA, the specific device should be reported, along with the utilization of resources such as DSP, BRAM, LUT and FF, performance density such as GOPs/slice can also be reported.

3.4.3 Fault tolerance Metric

A neural network N performing a computation H_N is said to be fault tolerant if the computation $H_{N_{fault}}$, performed by a faulty network N_{fault} obtained from N , is close to H_N . Formally, for $\epsilon > 0$, N is called ϵ -fault-tolerant, if it tolerates faulty components (for instance neurons/ synapses) for any subset of size at most n_{fails} :

$$\left\| \mathcal{H}_N(\mathcal{X}) - \mathcal{H}_{N_{fault}}(\mathcal{X}) \right\| \leq \epsilon, \quad \forall \mathcal{X} \in \mathcal{T} \quad (3.2)$$

where X is any stimuli, applied to the networks N and N_{fault} , that belongs to the training set T or is part of the input data to be processed by the networks. Given a problem, the goal for fault tolerance is to determine the network N that performs

the required computation and has the additional property that is ϵ -fault-tolerant with respect to T .

Moreover, in a strict sense, a neural network is absolutely or complete fault-tolerant to a class and number of faults if their effects measured by the chosen figure of merit is null. The complete fault tolerance requirement can be minimised toward easy degradation if we allow that the increase in the error is below a predefined threshold as stated in equation 1. Thus, recall that, when a statement about fault tolerance is made, it should be implicitly assumed a failure condition or criterion of the network functionality, which is the threshold below which it can no longer perform its function according to the specification. As such, fault tolerance in neural networks depends on the definition of the acceptable degree of performance and its intended application [11].

Chapter 4

Methodology

Deep neural networks have promising applications for data analytics in industrial applications, but they must consider the safety and reliability standards of the industries where they are applied.

This challenges us to measure and learn the error resilience aspects of these DNN systems.

My thesis focuses on two steps: (1) doing a software-level fault injection to discover network vulnerabilities and tolerance limits and (2) looking for a fault tolerance technique that makes the system more resilient.

Previous work for both phases is based on simulation and mitigation techniques at the hardware level.

Usually, to measure the robustness of a circuit there are radiation tests, which however are very expensive, fault injection techniques based on analysis tools or software programs describe a valid option to evaluate the reliability of a project, safety and fault coverage. And usually, traditional methods to protect computer systems from soft errors typically replicate the hardware components (Triple Modular Redundancy/TMR). While these methods are useful, they often influence large overheads in energy, performance and hardware cost.

Before starting the discussion, let's make some assumptions. As we saw in the previous chapter, many specialized accelerators have been proposed for DNN inferencing. Generalizing, a DNN accelerator consists of a global buffer and an array of parallel processing engines, as shown in Fig. 4.1.

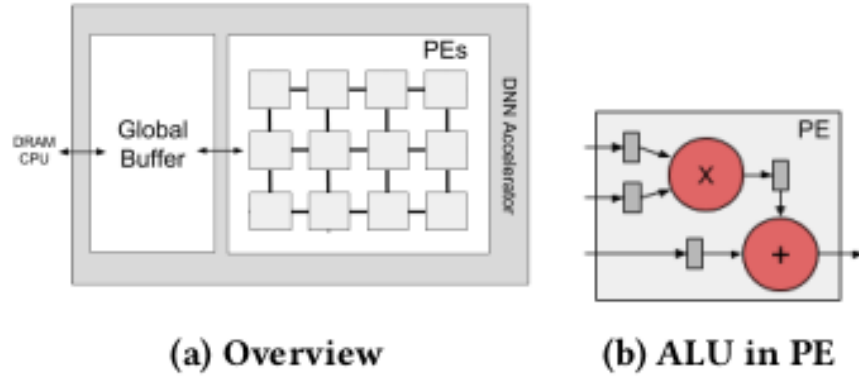


Figure 4.1: Architecture of general DNN accelerators

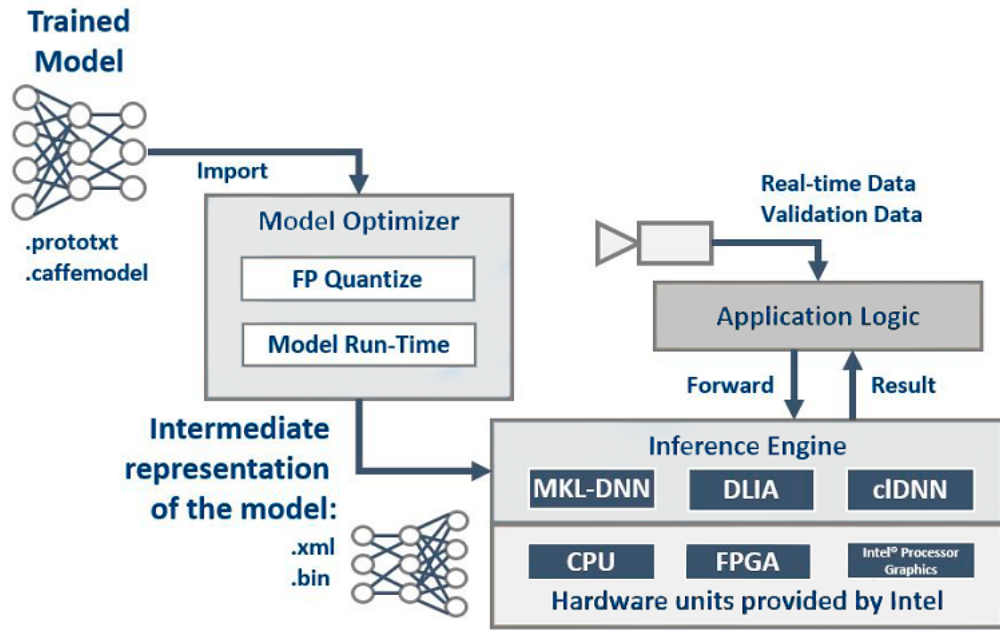


Figure 4.2: Hardware Accelerators of Deep Learning Models

Although DNNs can be performed on different hardware solutions, as can be seen in Fig 4.2. My discussion does not assume that our system runs on a particular device. We take it for granted that regardless of the hardware used, a charged particle that impacts the device also impacts the software application.

The consequences of soft errors that occur in DNN systems can be catastrophic as many of them are safety-critical, and error mitigation is required to meet certain reliability objectives. For example, in autonomous devices, a soft error can lead to miss-classification of objects, resulting in a wrong action taken by the device. In our fault injection experiments, we found many cases where an object can be miss-classified under a soft error.

Related Work

Laser beam and row hammer attacks are two common techniques used for injecting faults into memory. Both of them can alter the logic values in memory with high precision (single bit flip).

Laser beam can inject fault into SRAM. By exposing the silicon under laser beam, a temporary conductive channel is formed in the dielectric, which in turn causes the transistor to switch state in a precise and controlled manner.[12]

Row hammer can inject fault into DRAM. It exploits the electrical interactions between neighboring memory cells. By rapidly and repeatedly accessing a given physical memory location, a bit in its adjacent location may flip. By profiling the bit flip patterns in DRAM module and abusing the memory manage features, row hammer can reliably flip a single bit at any address in the software stack. [13] Several studies have been conducted towards modifying the network parameters to attack DNNs. Liu et al. in [4] proposed two fault injection attacks, i.e., Single Bias Attack (SBA) and Gradient Descent Attack (GDA). Studies have also been conducted for injecting faults in the computations during the execution of the DNNs. Towards this, Breier et al. in performed an analysis of using a laser to inject faults during the execution of activation functions in a DNN to achieve misclassification. The attack proposed by Breier et al. can be employed even when the DNN is unknown. [2] The literature goes to investigate the attachment points of a DNN, so DNNs have several inherent vulnerabilities.[14] In fig 4.3 are shown fault can affect a DNN system, as data stored in the memory, the control path of a DNN-based system, or the computational blocks. These faults can be injected using well-known techniques (variations in voltage, Electromagnetic (EM) interference, and heavy-ion radiation).

Damage can occur in the pre-processing phase of the image, an SEU introduces a noise at the input of the trained DNN during the inference. This imperceptible noise can either perform targeted misclassification or maximize the prediction error. Since these attacks cause vulnerabilities only at one image, therefore, this problem is not addressed in my discussion. Network parameters or computations can also be affected by SEUs. The core of DNN operation lies in the parameters, the parameters that have been trained to perform one of the many possible real-world tasks. For this reason, our discussion will focus on the impact of soft errors in parameters.

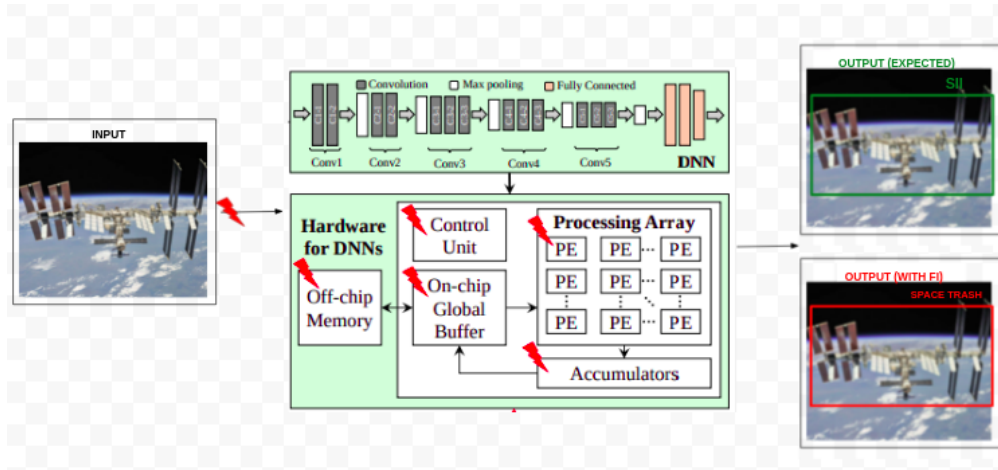


Figure 4.3: Threats to a DNN-based system

As shown in the chapter "The Problem", the points of sensitivity of a neural network are many. Given DNN threats, as shown in Fig. 4.3, these correspond to an alteration of the value that enters the artificial neuron. My analysis generalizes the problem, we estimate the robustness of the network to the alteration of its parameters through fault injection mechanisms that simulate its behavior in a real situation.

4.1 Fault injection

Failure injection aims to understand and verify if the response of a system conforms with its specifications under normal stress conditions. It's a technique that was first used to induce faults at the hardware level, specifically, at pin-level by changing the electrical signals on hardware devices.

In software engineering, failure injection helps improve the resiliency of a software system and enables the correction of potential failure tolerance deficiencies in that system. This is called fault removal. It also helps evaluate the potential impact of failure before it occurs in production. This is called failure forecasting. Failure injection has several key benefits:

- Understand and practice contingency and incident response.
- Understand the effects of real-world failures.
- Understand the effectiveness and limits of fault tolerance mechanisms.
- Remove design faults and identify single points of failure.
- Understand and improve the observability of the system.
- Understand the blast-radius of failures and help reduce it.
- Understand failure propagation between system components.

Fault injection attacks are also a popular attacks against cryptographic circuits and can be applied to bypass security, one example is the related works we have explored. Many different fault-injection techniques have been exploited that can be categorized as: software-based, hardware-based, simulation-based, emulation-based or hybrids [15].

Software Fault Injection

Hardware errors have become more conspicuous with reducing feature sizes. However, tolerating them exclusively in hardware is pricey. Researchers have explored software-based techniques for building error-resilient applications for hardware

faults. However, software-based error resilience techniques need configurable and accurate fault injection techniques to judge their effectiveness.

My analysis is predicated on a simulation of injections of faults like how it's exhausted related works but simulating the injection already at a high level.

For the feasibility study of my software simulation platform, I found recent articles to be explored. Particularly, PyTorchFI may be a run-time perturbation tool for deep neural networks (DNNs), implemented for the popular PyTorch deep learning platform. PyTorchFI allows users to perform perturbations on weights or neurons of DNNs at runtime. It also implements an extensible interface, enabling researchers to choose from multiple perturbation models (or design their own custom models), which allows for the study of error (or general perturbation) propagation to the software layer of the DNN output.

Additionally, PyTorchFI is very versatile: it demonstrates how the platform is applied to 5 different use cases for dependability and reliability research, including resiliency analysis of classification networks, resiliency analysis of object detection networks, analysis of models robust to adversarial attacks, training resilient models and DNN interpretability.[16]

This type of approach has some limitation: PyTorchFI operates at the application level of DNNs, which is useful for modeling high level perturbations and understanding their effect at the system level. Lower level perturbation models, such as register-level faults, cannot be captured at this level. However, we can still use PyTorchFI to model lower level faults by mapping them to either single or multiple bit-flips (in single or multiple neurons). Recent studies have shown that high level models can be used to study the effect of errors at the system level [17], [18]. At the same time, higher level models can run 4-6 orders of magnitude faster and are less expensive compared to low-level implementation.

Even if it is impossible to simulate some hardware failures at the software level, what we want to analyze is the impact of perturbations that occur on the network as a whole. They study the fault tolerance in the various networks and in the various levels of the network.

4.1.1 My FI technique

I present my FI technique, an open-source perturbation tool for DNNs implemented for the PyTorch deep learning framework. My FI technique is an easy-to-use, extensible, fast, and versatile tool for compute resiliency analysis performing perturbations in parameters of DNNs before inference/test following the Monte Carlo approach.

Monte Carlo approach

Monte Carlo Simulation, also called the Monte Carlo Method or a multiple probability simulation, could be a mathematical technique, which is employed to estimate the possible outcomes of an uncertain event. The Monte Carlo Method was invented by John von Neumann and Stanislaw Ulam during war II to boost deciding under uncertain conditions. It was named after a well known casino town, called Monaco, since the element of chance is core to the modeling approach, the same as a game of roulette.

Since its introduction, Monte Carlo Simulations have assessed the impact of risk in many real-life scenarios, like in *artificial intelligence*. They also provide variety of benefits over predictive models with fixed inputs, like the power to conduct **sensitivity analysis**. Sensitivity analysis allows decision-makers to work out the impact of individual inputs on a given outcome.

Unlike a traditional forecasting model, Monte Carlo Simulation predicts a collection of outcomes supported an estimated range of values versus a group of fixed input values. In other words, a town Simulation builds a model of possible results by leveraging a probability distribution, like a regular or distribution, for any variable that has inherent uncertainty. It, then, recalculates the results over and over, on every occasion employing a different set of random numbers between the minimum and maximum values. during a typical Monte Carlo experiment, this exercise may be repeated thousands of times to supply an oversized number of likely outcomes. Monte Carlo Simulations are utilized for long-term predictions because of their accuracy. because the number of inputs increase, the quantity of forecasts also grows, allowing you to project outcomes farther call at time with more accuracy. When a Monte Carlo Simulation is complete, it yields a spread of possible outcomes

with the probability of every result occurring.

One simple example of a town Simulation is to contemplate calculating the probability of rolling two standard dice. There are 36 combinations of dice rolls. supported this, you'll manually compute the probability of a selected outcome. employing a town Simulation, you'll simulate rolling the dice 10,000 times (or more) to realize more accurate predictions.

Single bit flip within the DNN

The first analysis is important to the understanding of the impact within the DNN of SEU-induced perturbation.

It is crucial to select an appropriate fault model to measure the impact of SEU-induced failures to DNNs. On overview of my FI simulation is shown in Fig. 4.4.

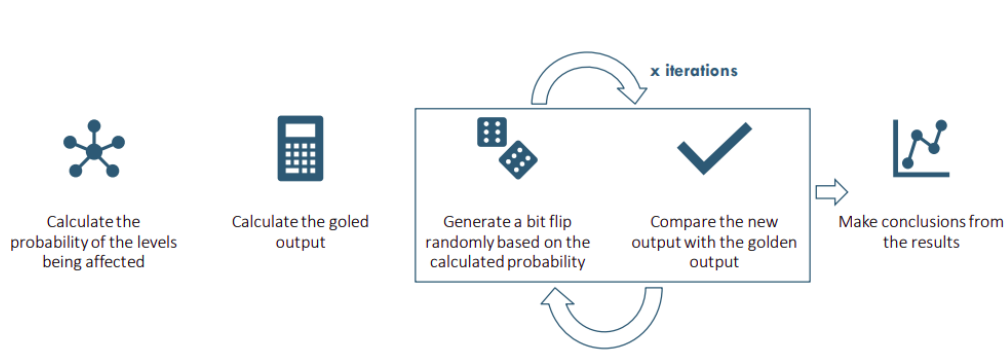


Figure 4.4: FI simulation - workflow

My simulation is divided into the following steps:

- **Calculate layer probabilities:** Regardless of the device on which they will be implemented, the weights will be stored on the device. Generally, our weights can look like in fig. 4.5. Network levels have a different number of parameters (weights + bias). Typically, fully connected levels are made up of a greater number of parameters.

In my discussion, I have assumed that if a level has a greater number of weights, this means that it also has a greater air to memorize. This implies that if I have a larger air, SEU's probability of hitting that air is greater.

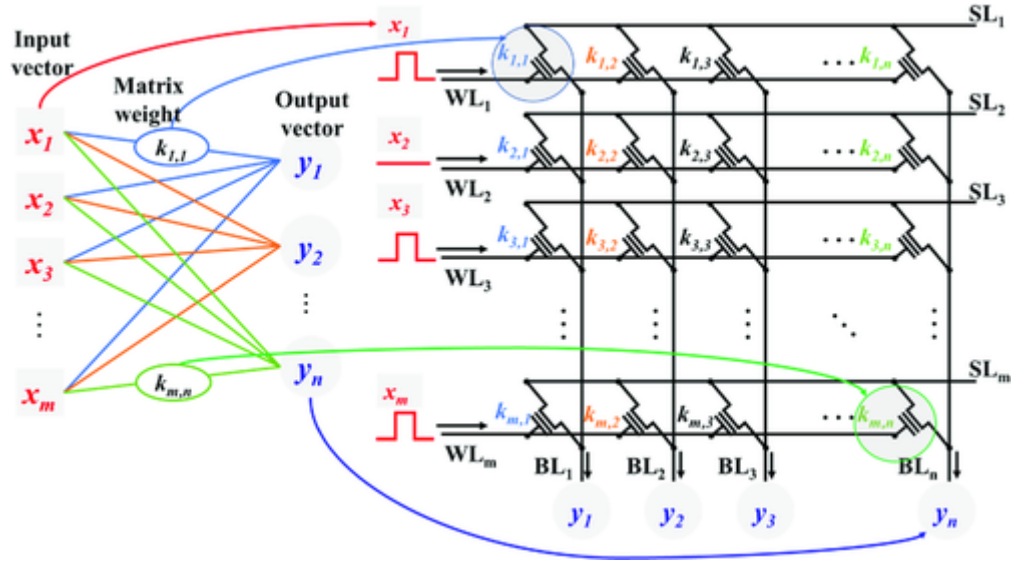


Figure 4.5: Hardware implementation of memory neural network

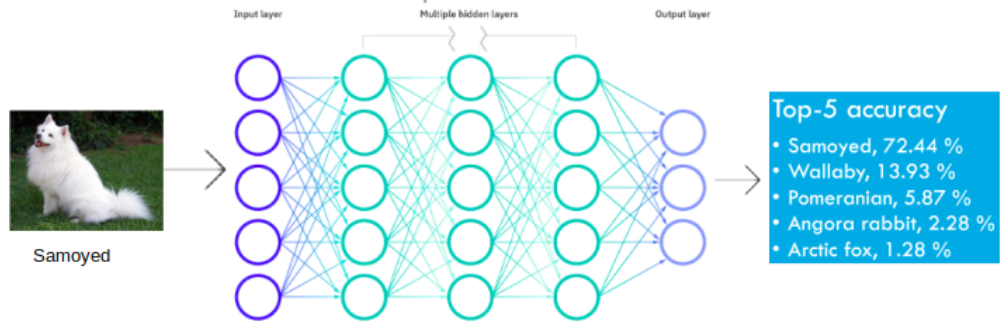


Figure 4.6: Golden output example

Hence, I have assumed that each level has a different probability of being affected by SEU. Given:

$$X = \{layer_i\}, i = 1, \dots, L \quad (4.1)$$

where L is the number of layers in DNN. The probability is calculated according

to the following equation:

$$P_{SEU}(layer_i) = \frac{N_{layer}}{N_{DNN}} \quad (4.2)$$

where P_{level} is the layer probability of being affected by an SEU/error/failure, N_{level} is the number of layer parameters and N_{DNN} is the total number of DNN parameters. In probabilities, the connection between each outcome for a random variable and their corresponding probabilities is explained as a probability distribution.

Each probability $P(layer_i)$ must be between 0 and 1:

$$0 \leq P_{SEU}(layer_i) \leq 1 \quad (4.3)$$

The sum of all the possible probabilities is 1 :

$$\sum_{i=1}^L P_{SEU}(layer_i) = 1 \quad (4.4)$$

Layers	Weights	Biases	Upset Probability[%]
First Convolutional Layer	23,232	64	0.038%
Second Convolutional Layer	307,200	192	0.5%
Third Convolutional Layer	663,552	384	1%
Fourth Convolutional Layer	884,736	256	1.44%
Fifth Convolutional Layer	589,824	256	0.96%
First Fully Connected Layer	37,748,736	4096	61.79%
Second Fully Connected Layer	16,777,216	4096	27.46%
Third Fully Connected Layer	4,096,000	1000	7.7%

Table 4.1: An example of Alexnet layer probability of being affected by an SEU/error/failure

- **Calculate the golden output:** Some images from the ImageNet dataset or a set of images from Imagenette has been used as the test vector . As a first validation experiment, I compute the results obtained from the DNN models in a normal condition, without injection. These outputs have been taken as

golden results in order to compare this with next results. The output is calculated on the basis of the top-5 accuracy, as shown in Fig. 4.6.

- **Generate a fit flip:** A simulation is the imitation of the operation of a real-world process or system over time. In my analysis, I don't want to hit a weight, a particular level, or a particular bit. I want to simulate a real context. In a real context, it has a network and its data is completely exposed to vulnerabilities. After calculating the probability that a failure occurs in the various sections of the DNN, I randomly generate a bit flip on one of the 32 bits of the parameters in one of the network levels, according to the probability calculated in the previous point. A layer is selected randomly using a probability distribution.

The result of this operation is a degraded network in one of its parameters.

- **Compare wrong output and golden output:** A single change in the network makes the network different from the original and gold model. The new injection model will have a new output (*wrong*). Which compared with the previous one produces the following outcomes:

- Best case: **No degradation**, in this case, my model does not suffer any degradation caused by the injection. The wrong output and the golden output are identical.

$$\begin{cases} |Y_{golden} - Y_{wrong}| = 0 \\ T_{golden} = T_{wrong} \end{cases} \quad (4.5)$$

- Average case: **Degradation**, in this case, the injection caused a small degradation in the accuracy of the output, but the prediction of the class is always the correct one.

$$\begin{cases} |Y_{golden} - Y_{wrong}| > \epsilon \\ T_{golden} = T_{wrong} \end{cases} \quad (4.6)$$

- Worst case: **Miss Classification**, in this case, even a single injection is enough to cause our application to malfunction. Which produces a very

different output from the golden one and causes miss-classification

$$\begin{cases} |Y_{golden} - Y_{wrong}| \gg \epsilon \\ T_{golden} \neq T_{wrong} \end{cases} \quad (4.7)$$

where Y_{golden} and Y_{wrong} are the output following the top-5 metric, T_{golden} and T_{wrong} are the predicted repetitive classes of the golden model and the model with FI.

- **Make consideration:** The purpose of my discussion is to measure the resilience and fault tolerance of the network. Through the Monte Carlo method, I am able to measure the resilience of a network to the injection of a single bit-flip. Through the Monte Carlo method, I can define what will be the probability of having one of the 3 outcomes described above: $X = \{ND, D, MC\}$. Where ND is probability not to have degradation, D is probability to have degradation and MC is probability to have miss-classification with a single bit-flip. About this method, other investigations have been carried out to discover the vulnerability points of the NNs and bits.

4.2 Accumulation of SEU-effects within the DNN

After having explored the methodologies used and the problem to be addressed, we are going to create a software simulation platform that simulates the soft errors and the behavior of the NN to them.

The simulation platform will be useful for two reasons:

- *Study the architect's vulnerabilities and strengths.* This point is critical to creating a new network model that is inherently resilient to failure
- *Find out the tolerance limit of the network.* This point is fundamental to always guarantee the functioning of the network, through parameter refresh mechanisms

The goal of the developed Monte Carlo analysis algorithm is to measure the average number of SEs affecting the DNN parameters that can be tolerated by the DNN,

before an SEFI is reported on the application system. The approach is particularly suitable for being applied before inference phase.

The methodology is based on a Monte Carlo algorithm that generates distributions of SE's within all the possible parameters bits controlling their effects within DNN and the new output produced with that of the golden network (original).

Algorithm 1 Accumulated SEUs simulation algorithm. $K=100000$ is a good number for iterations in Monte Carlo method β is the max bit in data representation, ω is the parameter injected

```

1: procedure FISIMULATION
2:    $\triangleright Y_{gold}$  is the golden DNN output
3:    $Y_{gold} \leftarrow calculategoldenoutput()$   $\triangleright P$  is the vector layers probability
4:    $\triangleright N$  is max number of accumulated SEUs
5:    $N \leftarrow 100$   $\triangleright$  start FI
6:   for  $n < N, n++$  do
7:      $\triangleright$  start  $K$  simulation
8:     for  $K$  do
9:        $\triangleright$  generate n injections based on P
10:      generateFIs()
11:       $Y_{wrong} \leftarrow calculatewrongoutput()$  end for
12:       $\triangleright$  Calculate Miss classification/Degradation Probability for each n
13:       $Pn_{missClassification} \leftarrow \frac{\sum_{i=1}^K MC}{K}$ 
14:       $Pn_{Degradation} \leftarrow \frac{\sum_{i=1}^K D}{K}$ 
15:    end for
16:  end procedure

```

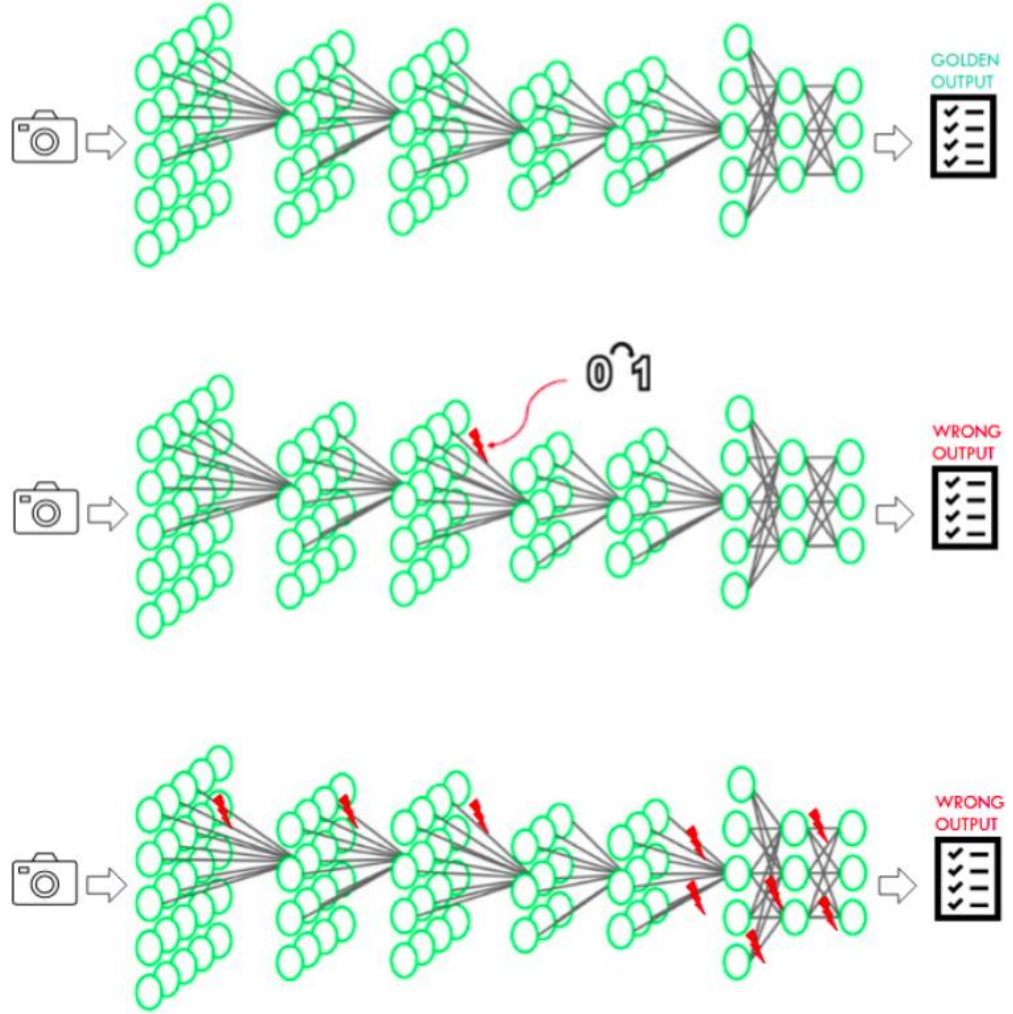


Figure 4.7: (a) Golden DNN (b), random single bit-flip ,(c) random multiple bit-flip

The Fig. shows the difference between the three scenarios.

1. The first where I have the original DNN, designed and trained to perform a task.
2. In the second scenario, a bit flip is generated randomly. The probability of failure and the sensitivity to levels and bits are studied.
3. Third scenario goal is to find the tolerance limit of the DNN.

Through this last simulation it is possible to construct a function that represents the error rate when the SEUs accumulate.

In our discussion we construct two functions. The first function represents the probability of being affected by degradation, the second function represents the probability of being affected by miss-classification.

These functions are metrics for analyzing the DNNs resilience. The functions also establish the limit of SEU that a DNN can tolerate before the system becomes unusable.

4.3 Fault Tolerance

This phase of the discussion is an exploratory phase, as many techniques can be experimented and combined with each other to obtain better performance.

Fault Tolerance is a crucial property for Neural Networks to ensure reliable computation for a long duration with graceful degradation over time. Typically, well generalized models have the parameters with low variance ensuring equal computational weight to all nodes in the network. Hence, the loss of some of the nodes can be compensated by other nodes without a significant loss in performance.

The state-of-the-art neural networks are vulnerable to perturbation [19]. That is, small perturbation on input forces neural network to provide adversary output. To defeat degradation example problem, recent countermeasures aim at improving the generalization capability of neural network by manipulating DNN's training procedure. For instance, Papernot et al. [20] proposed to train the same neural network twice and the soft labels used in the second train process are able to improve the generalization capability. Gu and Rigazon, and Goodfellow et al. [21]

proposed to regularize the objective function in training procedure to manually restrict the predicted class for data points around training samples.

Though no previous works have studied fault injection attack on DNN, there are many works improving DNN’s fault-tolerance capability against random faults on the weights.

Though no previous works have studied fault injection attack on DNN, there are many works improving DNN’s fault-tolerance capability against random faults on the weights. These works can be mainly divided into three groups:

- training with artificial fault [22]
- neuron duplication [23]
- and weight restriction [24], [25]

Training with artificial fault intentionally injects faults into DNN during training process, so that obtained DNN can perform correctly even in faulty case.

However, the cost of enumerating all faulty cases is prohibitive when the number of faulty cases is exponentially larger, like enumerating multi-bit faults, where the number of exploitable faulty cases is large.

Neuron duplication improves the redundancy of DNN by duplicating internal neurons and scaling down corresponding weights.

At last, weight restriction determines a range to which weights should belong during training, and any weight being outside the range is forced to be its upper limit or lower limit. In this way, weight restriction can mitigate the interference from fault imposing large perturbation on a weight.

4.3.1 My resilience techniques

The papers cited above lay the foundations for some food for thought. Scrolling through the literature numerous techniques are applied to reduce the energy and memory to run the network. The results of these experiments show how techniques to improve resource models make the network inherently more resilient. In general, we show that generalization is a strong ally of robustness. Some papers seem to support these results [26], [27]. Neural networks are both computationally intensive and memory intensive, making them difficult to deploy on embedded systems

with limited hardware resources. To address this limitation, [28] introduce "deep compression", a three stage pipeline: pruning, trained quantization and Huffman coding, that work together to reduce the storage requirement of neural networks by 35x to 49x without affecting their accuracy. This method reduced the storage required by AlexNet by 35x, from 240MB to 6.9MB, without loss of accuracy. This allows fitting the model into on-chip SRAM cache rather than off-chip DRAM memory.

Pruning

Deep Learning models nowadays need a major amount of computing, memory, and power which becomes a bottleneck within the conditions where we'd like real-time inference or to run models edge devices and browsers with limited computational resources. Energy efficiency may be a major concern for current deep learning models, one of the methods for tackling this efficiency is enabling inference efficiency.

Pruning is one in all the methods for inference to efficiently produce models smaller in size, more memory-efficient, more power-efficient and faster at inference with minimal loss in accuracy, other such techniques being weight sharing and quantization. Out of several aspects that deep learning takes as an idea from the realm of Neuroscience.

Pruning in artificial neural networks has been taken as an idea from *Synaptic Pruning* in the human brain where axon and dendrite completely decay and die off resulting in synapse elimination that occurs between early childhood and the onset of puberty in many mammals. Pruning starts near the time of birth and continues into the mid-20s, as shown in Fig. 4.8.

DNN pruning is shown in Fig. 4.9. Networks generally look like the one on the left: every neuron in the layer below has a connection to the layer above, but this means that we have to multiply a lot of floats together.

Ideally, we'd only connect each neuron to a few others and save on doing some of the multiplications; this is called a "sparse" network. Sparse models are easier to compress, and we can skip the zeroes during inference for latency improvements. If you could rank the neurons in the network according to how much they contribute, you could then remove the low ranking neurons from the network, resulting in a

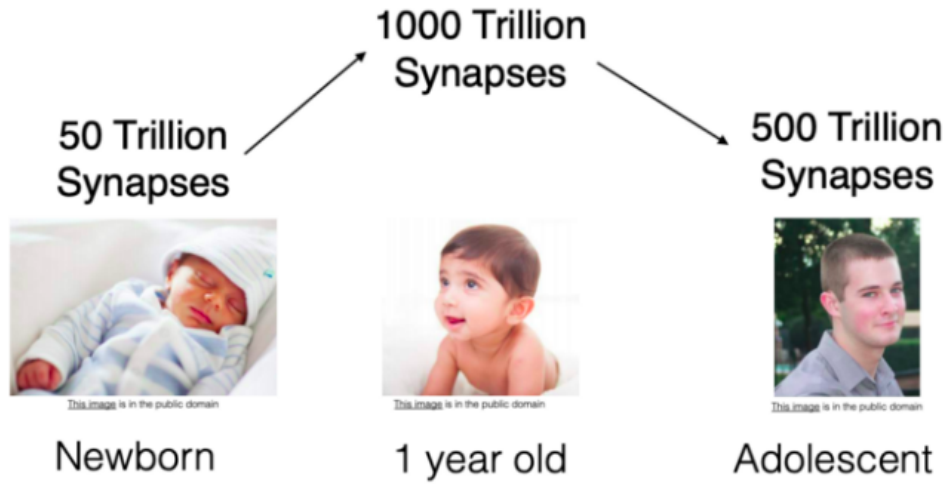


Figure 4.8: Synaptic Pruning in mammals

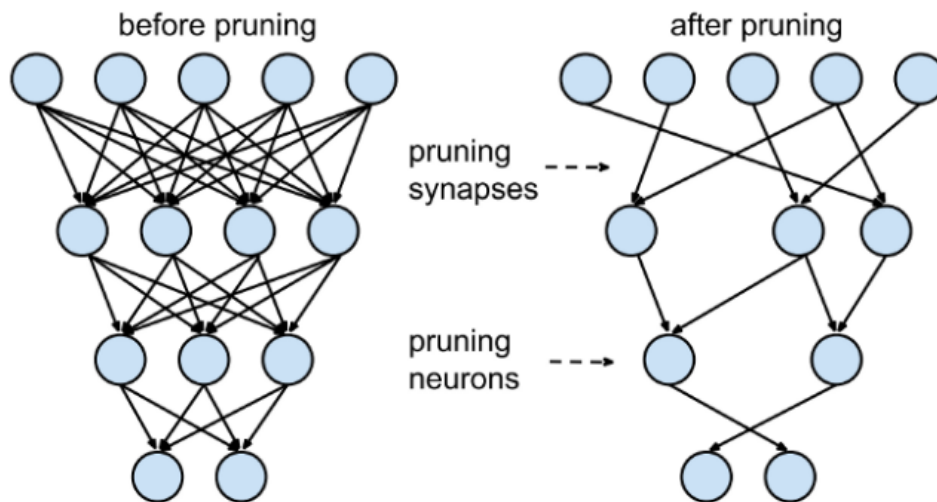


Figure 4.9: DNN pruning

smaller and faster network.

Getting faster/smaller networks is important for running these deep learning networks on mobile devices.

- **Weight pruning:** Set individual weights in the weight matrix to zero. This

corresponds to deleting connections as in the figure above. Here, to achieve sparsity of $k\%$ we rank the individual weights in weight matrix W according to their magnitude, and then set to zero the smallest $k\%$.

- **Unit/Neuron pruning:** Set entire columns to zero in the weight matrix to zero, in effect deleting the corresponding output neuron. Here to achieve sparsity of $k\%$ we rank the columns of a weight matrix according to their L2-norm and delete the smallest $k\%$.

Naturally, as you increase the sparsity and delete more of the network, the task performance will progressively degrade.

- **Iterative Pruning:** The ranking, for example, can be done according to the L1/L2 norm of neuron weights. After the pruning, the accuracy will drop (hopefully not too much if the ranking is clever), and the network is usually trained-pruned-trained-pruned iteratively to recover. If we prune too much at once, the network might be damaged so much it won't be able to recover. So in practice, this is an iterative process often called 'Iterative Pruning': Prune / Train / Repeat.

Pruning in Pytorch

Some methods implement pruning in PyTorch, but they are doing not result in faster inference time or memory savings. The rationale for that's that sparse operations aren't currently supported in PyTorch (version 1.7), and then just assigning weights, neurons or channels to zero doesn't result in real neural network compression. Thus, experiments of this method give theoretical improvements and not real ones.

The idea of pruning is to scale back the scale of an outsized neural network without sacrificing much of predictive power. It might be done by either removing (=pruning) weights, neurons or perhaps entire channels in a very neural network. There are multiple possibilities of a way to have it away starting from randomly pruning all weights to pruning weights/neurons/channels supported some metrics.

1. *Unstructured pruning of random weights*: will prune the **random** percentage of the connections in the parameter named **weight** and/or **bias** in a layer. The module is passed as the first argument to the function; **name** identifies the parameter within that module using its string identifier; and **amount** indicates either the percentage of connections to prune (if it is a float between 0. and 1.), or the absolute number of connections to prune (if it is a non-negative integer). Pruning acts by removing **weight** or **bias** from the parameters and replacing it with a new parameter called **weight_orig** and/or **bias_orig**. **weight_orig** and/or **bias_orig** stores the unpruned version of the tensor.
2. *Unstructured pruning of the smallest weight*: will prune the **smallest** percentage of the connections in the parameter named **weight** and/or **bias** in a layer. The module is passed as the first argument to the function; **name** identifies the parameter within that module using its string identifier; and **amount** indicates either the percentage of connections to prune (if it is a float between 0. and 1.), or the absolute number of connections to prune (if it is a non-negative integer). Pruning acts by removing **weight** or **bias** from the parameters and replacing it with a new parameter called **weight_orig** and/or **bias_orig**. **weight_orig** and/or **bias_orig** stores the unpruned version of the tensor.
3. *Structured pruning*: It is possible to pass a dimension (**dim**) to specify which channel should be dropped. For fully-connected layers **dim=0** corresponds to “switching off” output neurons. Therefore, it does not really make sense to switch off neurons in the last classification layer. For Convolutional layers like **dim=0** corresponds to removing the output channels of the layers.

Quantization

Deep learning features a growing history of successes, but heavy algorithms running on large graphical processing units are faraway from ideal. A comparatively new family of deep learning methods called quantized neural networks have emerged in answer to the current discrepancy.

Neural networks are composed of multiple layers of parameters, each layer transforms the input image, separating and contracting the feature space, leading to the

separation of input images to their distinct classes. Perhaps the foremost notable of deep learning problems are image classification, object detection and segmentation. Deep learning for classification tasks means training the parameters of a neural network specified the algorithm learns to discern between object classes. This is often performed by feeding many images of labelled data to the neural network, while updating the parameters to extend performance on a smooth objective function. A drawback is that an outsized number of parameters are used, compared to more traditional algorithms. Thus enters quantization as a way to bring the neural network to an affordable size, while also achieving high performance accuracy. This is often especially important for on-device applications, where the memory size and number of computations are necessarily limited.

Quantization for deep learning is that the process of approximating a neural network that uses floating-point numbers by a neural network of low bit width numbers. This dramatically reduces both the memory requirement and computational cost of using neural networks.

The neural network can be quantized after training is finished. However, by far the most effective method for retaining high accuracy is to quantize during training [29], who describe the main idea of neural network parameter quantization during training.

We don't need to stop with just the parameters of the quantized network quantizing the inputs to each convolutional layer of the neural network results in massive reduction of necessary computation. To be able to use quantized inputs for each convolution layer, the activation function is replaced by a quantization function [30].

Chapter 5

Results

5.1 Single bit flip within the DNN results

The first analysis is important to the understanding of the impact within the DNN of SEU-induced perturbation. This analysis shows two important results:

- Typically, the first layers are most vulnerable than the last layers
- The most significant bit of the exponent has a big impact on the miss classification

The results and observations about these are shown below.

5.1.1 The most vulnerable bits

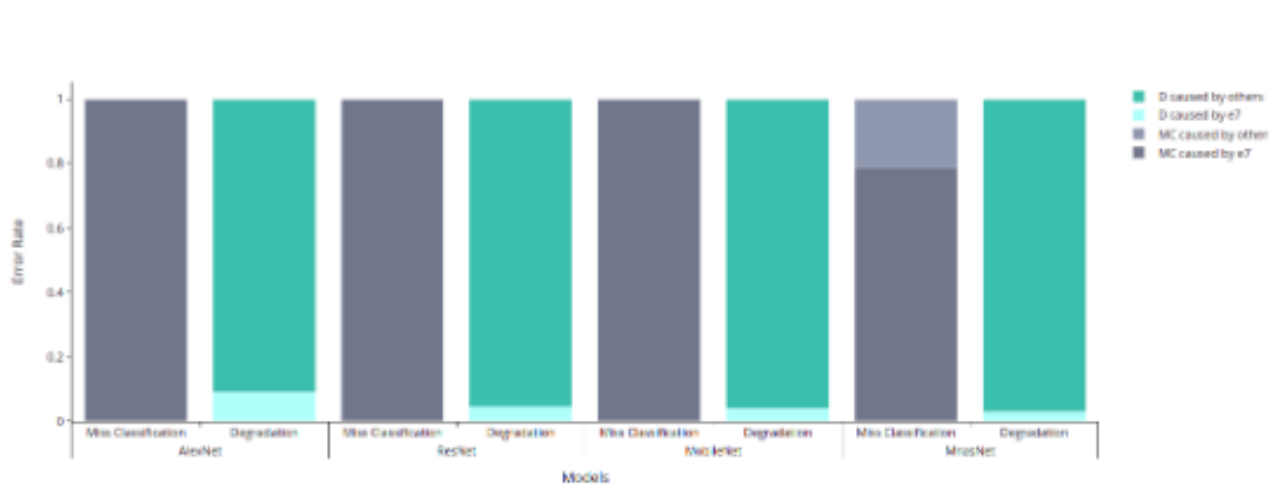


Figure 5.1: Impact of bit-flip on models (MC and D)

The Fig. 5.1 shows the great impact of flipping the most significant bit on e7 (the most significant bit of the exponent). Almost 100% of the miss classification is caused by an FI on e7, only in a few cases does it cause degradation, probably because it is located in the last levels of the DNN.

These results can be checked by some papers. Rakin et al. in [31] proposed a methodology, Bit-Flip Attack (BFA), for attacking DNNs by flipping a small number of bits in the weights of the DNN.

BFA focuses on identifying the most vulnerable bits in a given DNN that can maximize the accuracy degradation while requiring a very small number of bit-flips in the binary representation of the parameters of the DNN. It is designed for quantized neural networks, i.e., where the weight magnitude is constrained based on the fixed-point representation.

For floating-point representation, even a single bit-flip at the most significant location of the exponent of one of the weights of the DNN can result in the network generating completely random output.[32] present a case study where they simulate a neural network in the presence of memory faults (only in the region where the parameters, weights, of a network are stored).

The case-study is based on image classification on the ImageNet dataset using the VGG-f network. For the evaluation, they assumed 32-bit floating point precision for both weights and inputs. An illustration of the scenario is shown in Fig. 5.2.

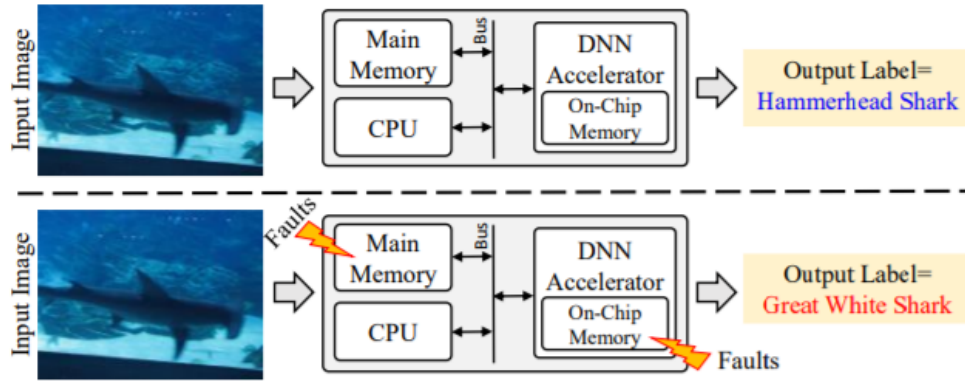


Figure 5.2: Experimental setup for illustrating the impact of memory faults in DNN execution

The structure of the floating point number system used for this analysis is shown in Fig. 5.3.

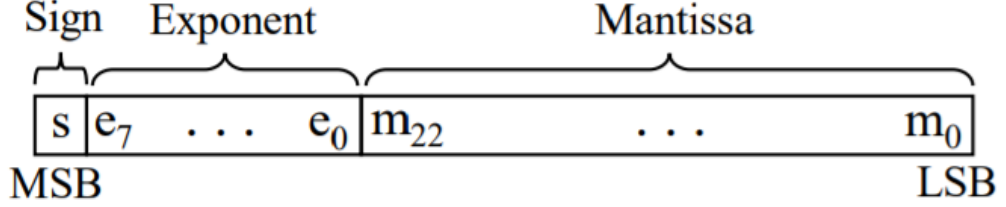


Figure 5.3: Single precision floating point storage format used in DNN design

They divided the analysis into two parts: (1) Bit flips from 0'b to 1'b; and (2) Bit flips from 1'b to 0'b. Also, to analyze the impact of the bit flips at particular locations in the words, they injected bit-flip errors individually at different bit location of a fraction of weights of a network layer. The results of 0'b to 1'b bit flip error injection in the first layer of the network is presented in Fig. 5.4.

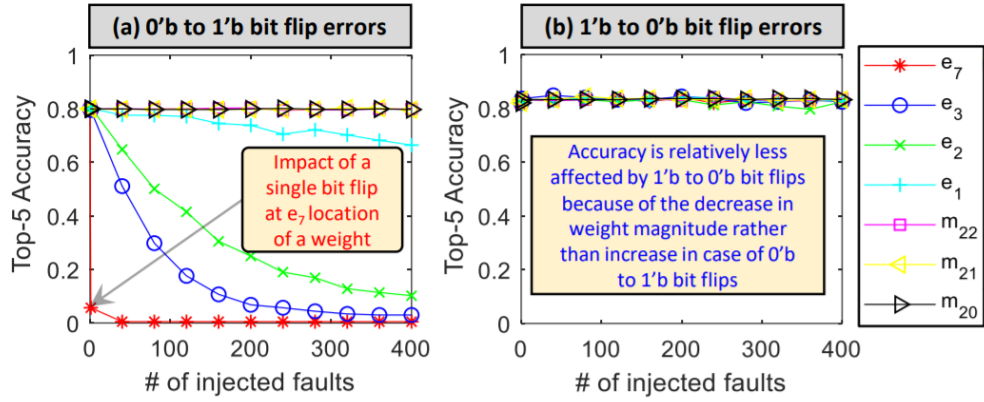
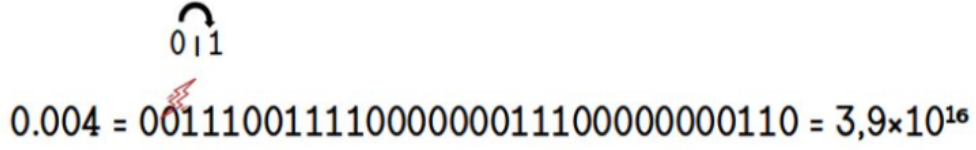


Figure 5.4: Impact of bit flip errors on the accuracy of VGG-f network used for an image classification application

Accuracy is relatively less affected by 1'b to 0'b bit flips because of the decrease in weight magnitude rather than increase in case of 0'b to 1'b bit flips.

In DNN, parameters are stored on 32 bit floating-point, there aren't value very large, but there are value very small. A change in a small parameters generate a

greater distortion, as shown in Fig. 5.5.



$$0.004 = 00111001111000000011100000000110 = 3,9 \times 10^{16}$$

Figure 5.5: The results of 0'b to 1'b bit flip error injection in the most vulnerable bit

The aforementioned case-study highlights the need studying the impact of different types of bit-flip faults in different components of a DNN systems, and leveraging this knowledge for designing and optimizing fault-tolerant methods that can help avoiding errors which lead to catastrophic results in case of safety-critical applications.

As we will show in next section, fault-tolerant method to mitigate the impact of 0'b to 1'b bit flip error injection in the most vulnerable bit can be removing the smallest values (vulnerable parameters) in DNN parameters to decrease probability to have bit flip in vulnerable parameters.

In [33]

5.1.2 Layer Vulnerability

The analyzes conducted show a greater sensitivity when SEU afflict the first levels of the network.

They are shown in Fig. 5.9 and Fig. 5.10, the results on AlexNet and ResNet to show how the error curve tends to smooth over the levels. Demonstrating how fully connected levels are more tolerant than convolutional levels and demonstrating how the first levels are very sensitive to errors. Typically, this problem could be due to several causes:

1. The first levels learn *stronger* characteristics
2. The error in the first levels propagates in the network

In the literature, we can find possible explanations:

1. The human brain processes a huge amount of information the second we see an image. Each neuron works in its own receptive field and is connected to other neurons in a way that they cover the entire visual field. Just as each neuron responds to stimuli only in the restricted region of the visual field called the receptive field in the biological vision system, each neuron in a CNN processes data only in its receptive field as well. The layers are arranged in such a way so that they detect simpler patterns first (lines, curves, etc.) and more complex patterns (faces, objects, etc.) further along.

CNNs learn a hierarchical representation of the input in which the initial layers detect very basic patterns like edges and gradients, while layers located on top of the hierarchy learn complex patterns which are useful for the classification task at hand. The concept of hierarchical feature learning is illustrated in Fig. 5.6 where the network was not specifically trained for fish images therefore, patterns found by the network are generic.

A mistake in learning simpler patterns can have a greater impact in the long run.

2. Given a general deep neural network as in Fig. num, we treat the input layer and final predicted probability distribution on candidate classes as two multidimensional vectors X and Y respectively, then the whole DNN is a parameterized function F , and $Y = F(X)$, where represents all the weights and biases in DNN. Neurons are connected by links with different weights and biases, characterizing the strength between neurons. A neuron receives inputs from many neurons in the previous layer, applies its activation function on these inputs, and transmits the result to neurons in next layer. Formally, a neuron's output y is given by:

$$y = g(u)$$

$$u = \sum_{i=1}^n w_i x_i + w_0$$

where x_1, x_2, \dots, x_n are neuron outputs in the previous layer, w_1, w_2, \dots, w_n are weights on respective links, w_0 represents the bias, and g is the activation

function.

Considering this condition, this means that the degradation propagates in the DNN, as shown in Fig. 5.8.

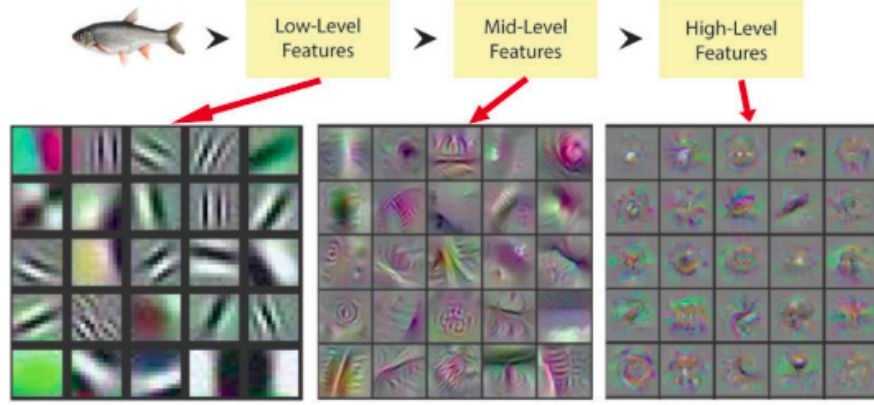


Figure 5.6: Hierarchical representation learning by a CNN where the initial layer detects simple patterns like edges and gradients while higher layers detect more abstract features

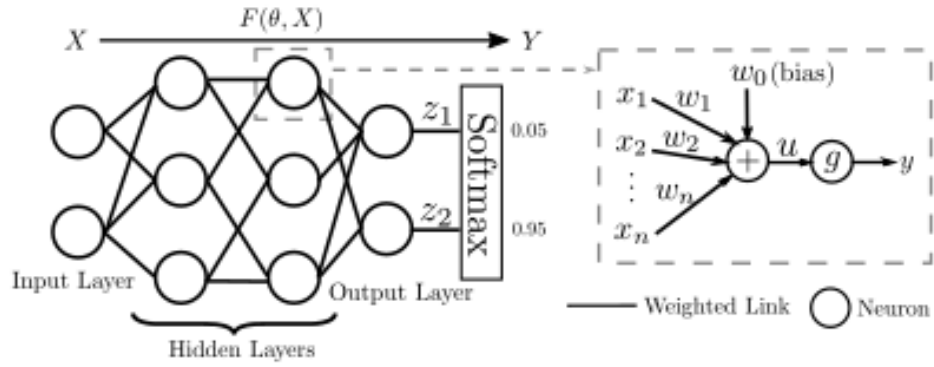


Figure 5.7: A deep neural network example and the general structure for a neuron.

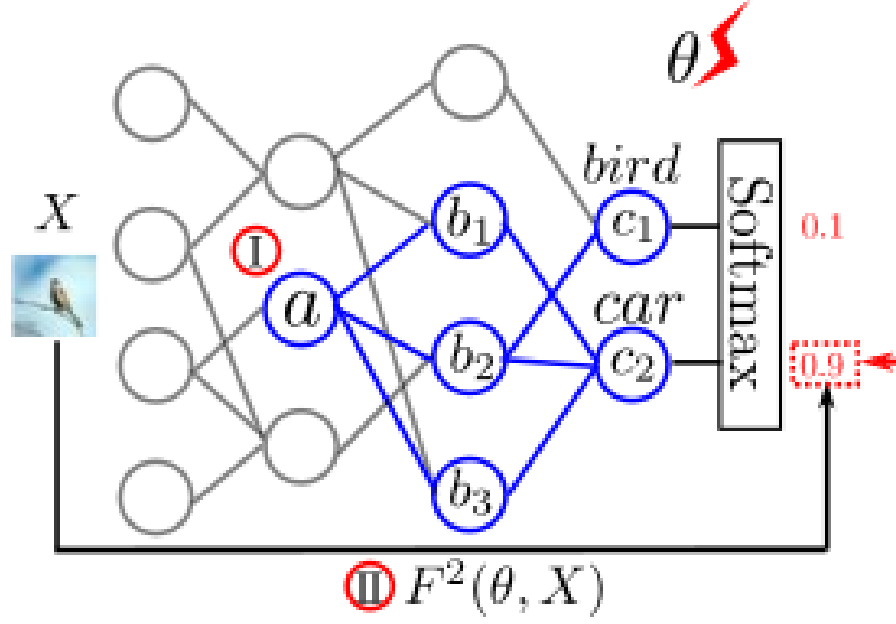


Figure 5.8: Fault injection attack on DNN

5.2 FI results

Finally we come to the final results. The final results are aimed at the study of CNN to provide for the implementation of a solution on radiation resilient devices in order to limit the degradation of the network due to the accumulation of radioactive dose.

Analyzes on different networks were performed. The analyzes investigate network degradation through different models.

The models differ in technology and size. The size factor is very important because the size of the injection air also depends on it.

The table below is an example:

The results will then be normalized according to the size of the injection air.

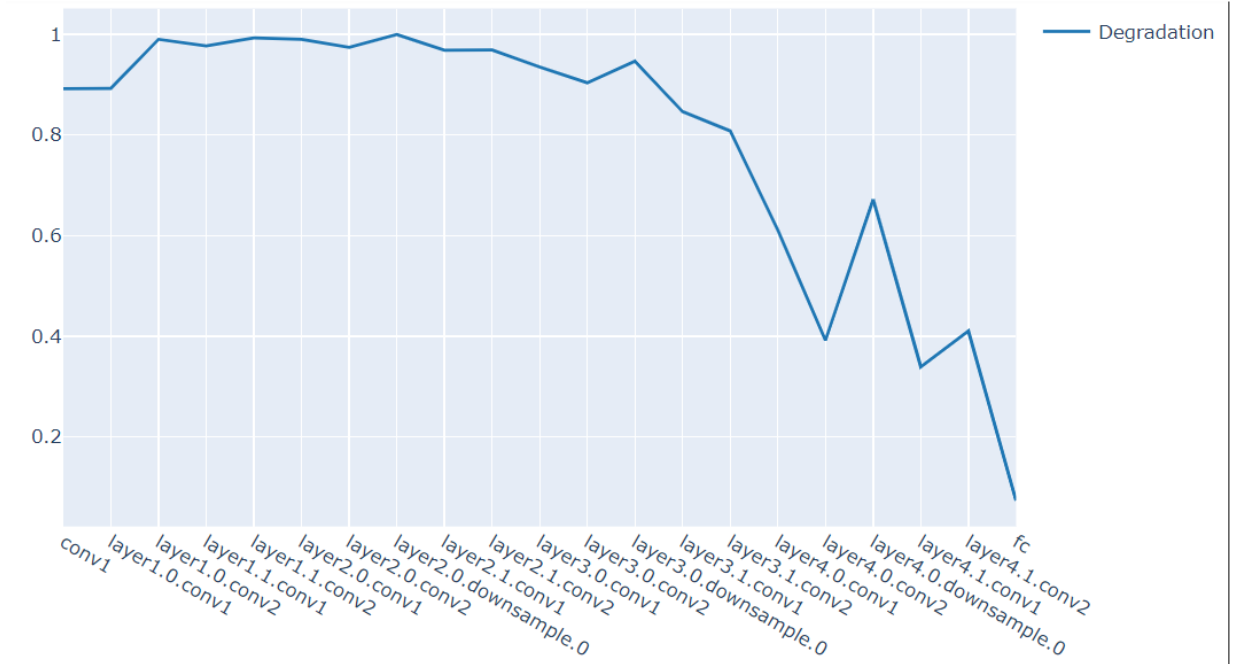


Figure 5.9: Impact of bit-flip on Resnet through the layers

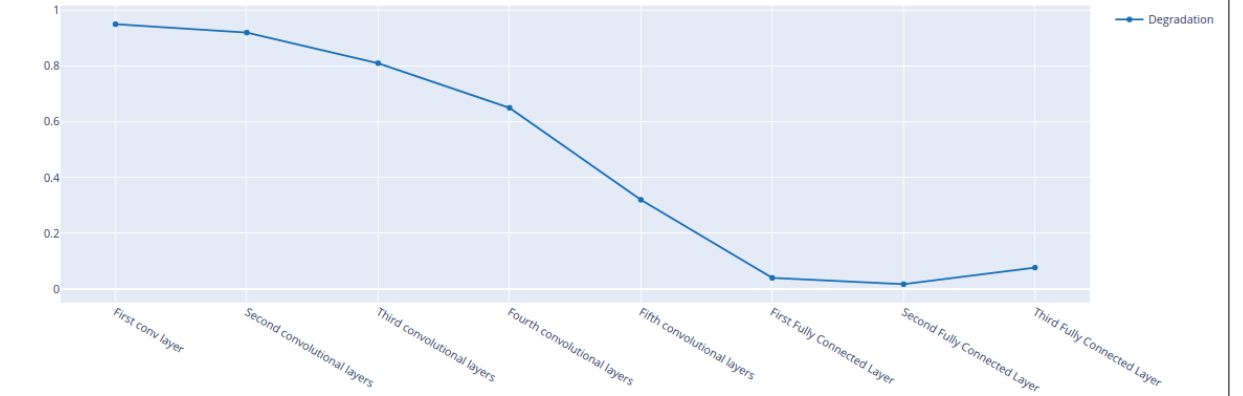


Figure 5.10: Impact of bit-flip on Alexnet through the layers (Degradation)

The results show in Fig. 5.12-5.15 a higher degree of tolerance inherent in some more compact DNN models designed for mobile environments such as MobileNet and MnasNet. These networks are leaner and more compact than AlexNet.

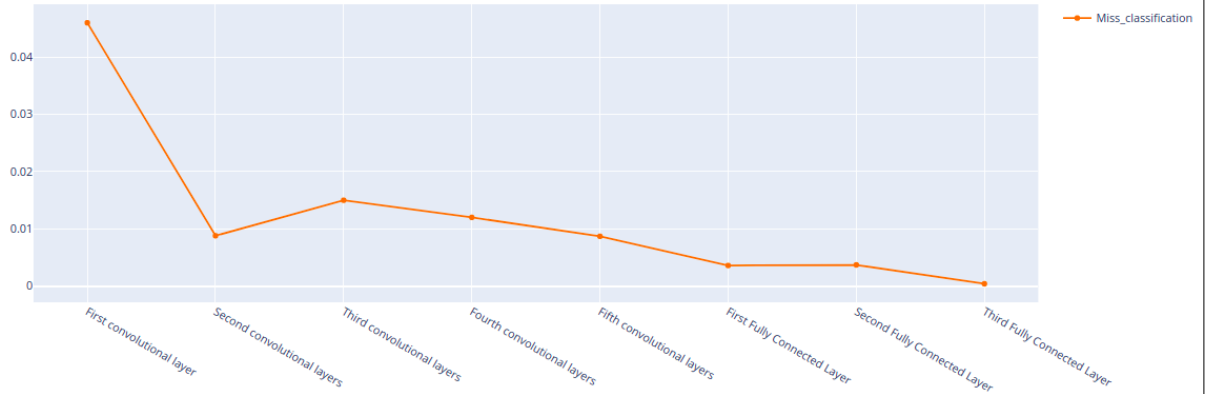


Figure 5.11: Impact of bit-flip on Alexnet through the layes (MissClassification)

Model	Parameters	Memory injectable
Alexnet	61.100.840	2G cells
ResNet	11.689.512	370M cells
MobileNet	3.504.872	110M cells
MnasNet	2.218.512	70M cells

Table 5.1: Model comparison: Total parameters in models

Models designed to be more functional in mobile environments are also inherently fault tolerant. The error rate in these models grows slowly compared to models such as AlexNet and ResNet, which have a much higher initial degree of accuracy.

5.3 Faults Tolerance results

The idea of pruning as a resilience technique is taken up by some paper that use pruning as a defense technique for very specific attacks. In [27], pruning techniques are used to mitigate the success of DNN backdoor attacks. In DNN backdoor attacks, the DNN learns to misbehave on backdoored inputs while still behaving on clean inputs because backdoored inputs trigger neurons that are otherwise dormant in the presence of clean inputs.

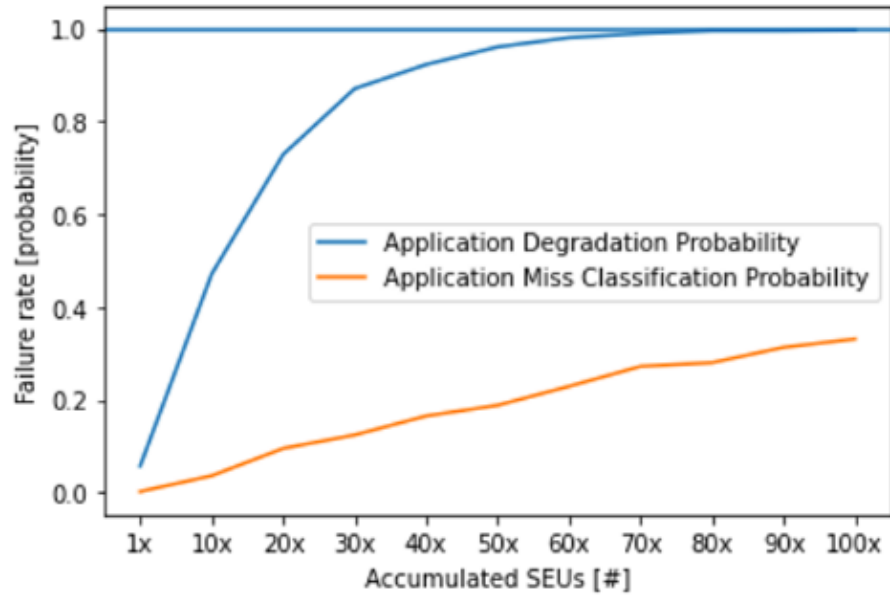


Figure 5.12: Failure rate function AlexNet

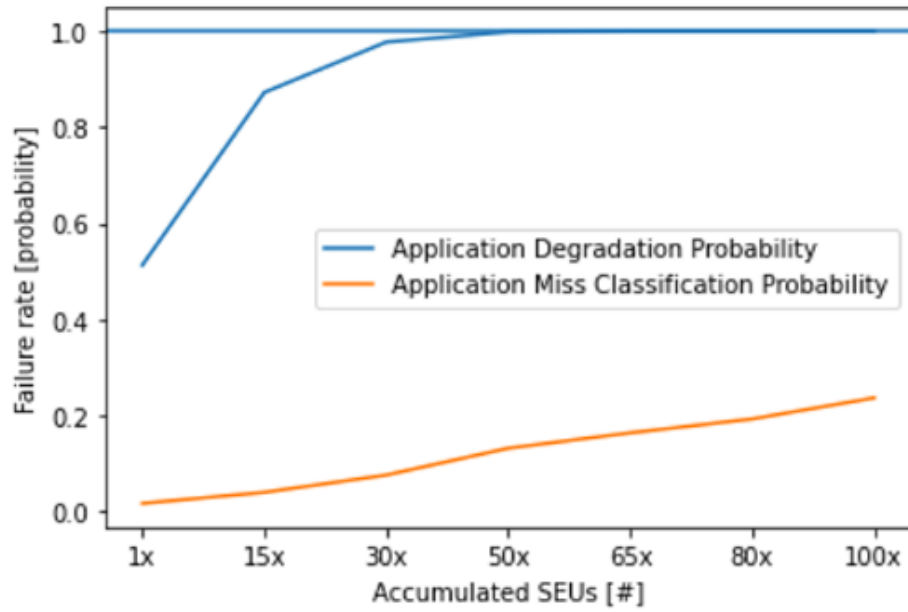


Figure 5.13: Failure rate function ResNet

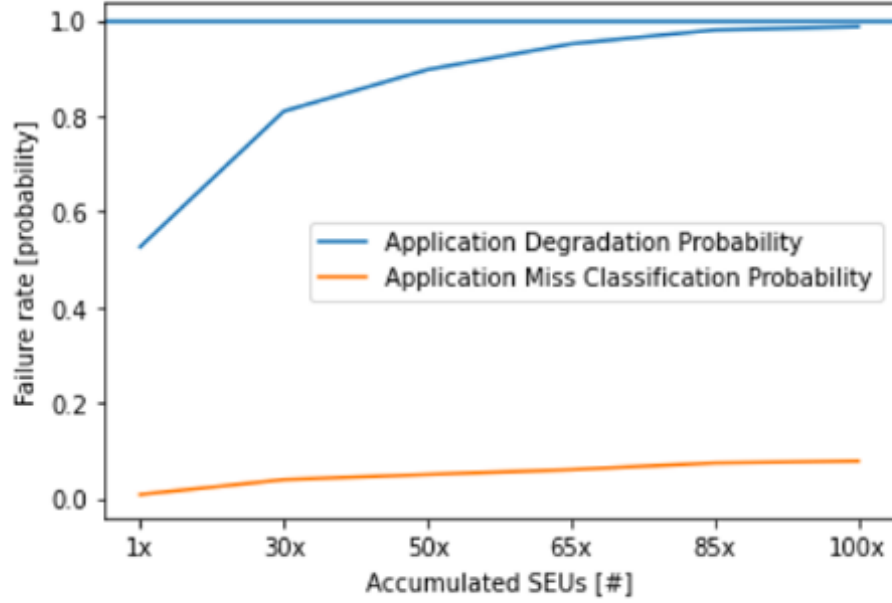


Figure 5.14: Failure rate function MobileNet

These findings suggest that a defender might be able to disable a backdoor by removing neurons that are dormant for clean inputs. They refer to this strategy as the *pruning defense*. They then evaluate fine-pruning, a combination of pruning and fine-tuning, and show that it successfully weakens or even eliminates the backdoors, in some cases reducing the attack success rate to 0% with only a small drop in accuracy for clean (original) inputs.

IN [34] shown that pruning not only improves the resource efficiency of neural networks, but also the resilience against poisoning attack. In particular, they found that the resilience depends on the level of pruning, a result similar to that obtained by me, the model retains higher accuracy where less trainable parameters remained after pruning.

The fields of application are obviously many, in my analysis I do not have a targeted attack. My analysis scenario is not deterministic, but having analyzed the critical problems I can implement mitigation strategies. As in [27], I apply a pruning strategy to mitigate the problems caused by e7. In particular, the alteration of e7 causes a large alteration in weight that goes from an extremely small weight to an

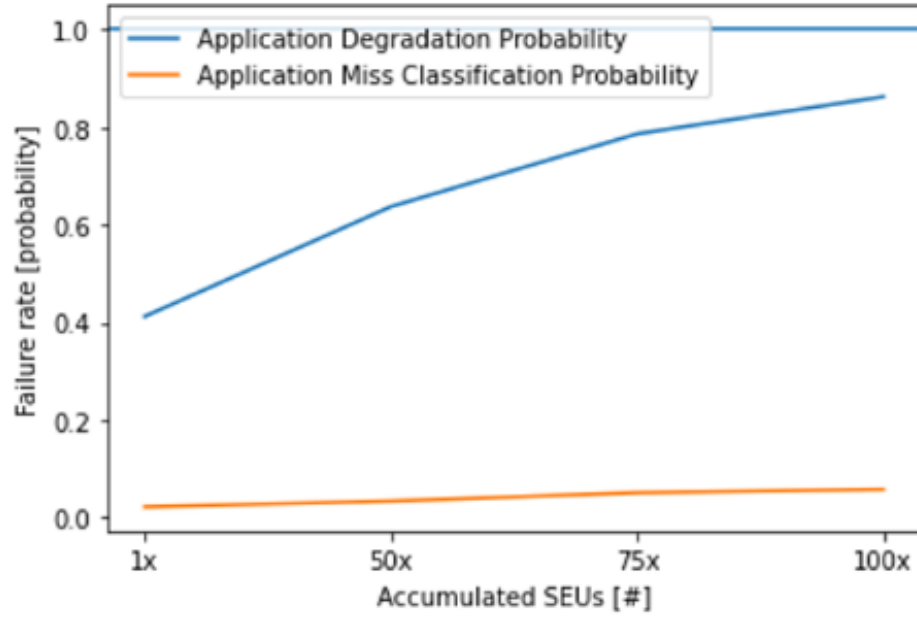


Figure 5.15: Failure rate function MnasNet

extremely large weight. Going to eliminate extremely small weights then helps my model perform better.

5.3.1 Prining results

The results obtained through pruning show remarkable achievements. Contrary to replication techniques, such as fault mitigation techniques, it turns out that pruning the network not only brings benefits such as reduced resources, but also proves to be a very good fault tolerance technique.

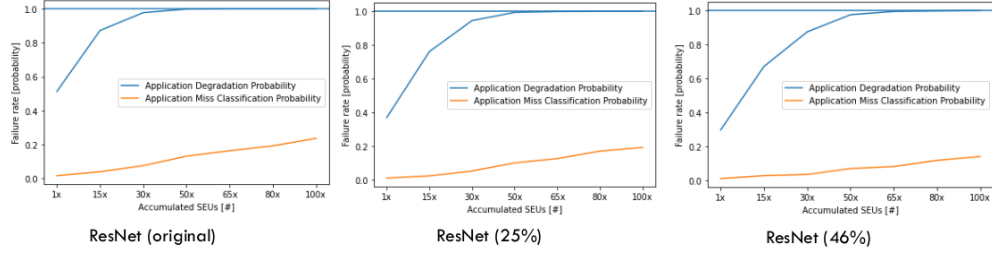


Figure 5.16: Failure rate function ResNet (a) ResNet (original) (b) ResNet (25%) (c) ResNet (46%)

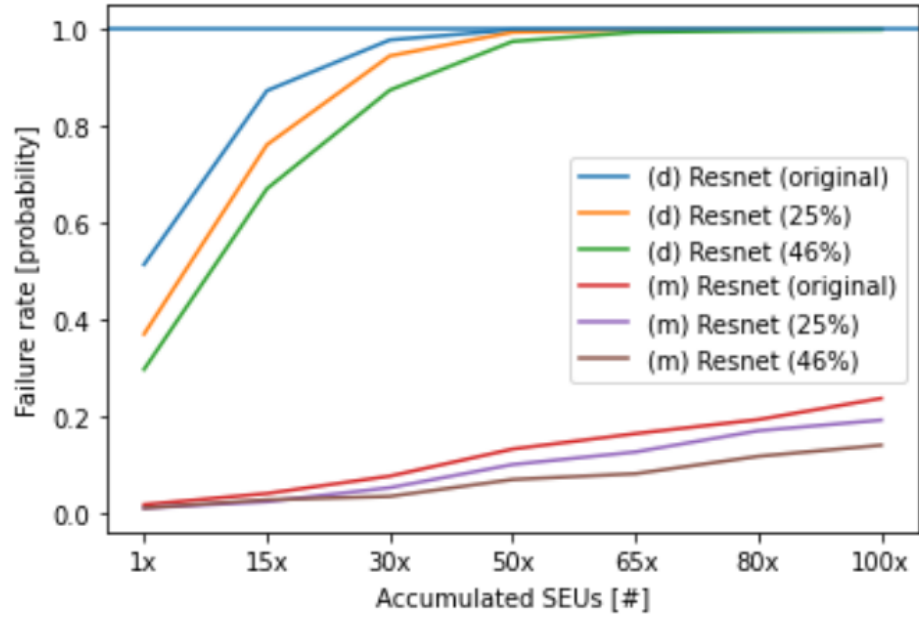


Figure 5.17: ResNet Pruning Results Summary

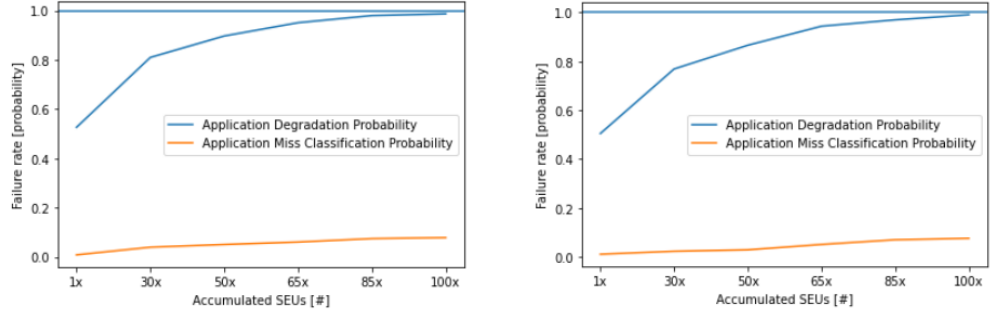


Figure 5.18: Failure rate function MobileNet (a) MobileNet (original) (b) MobileNet (0.72x)

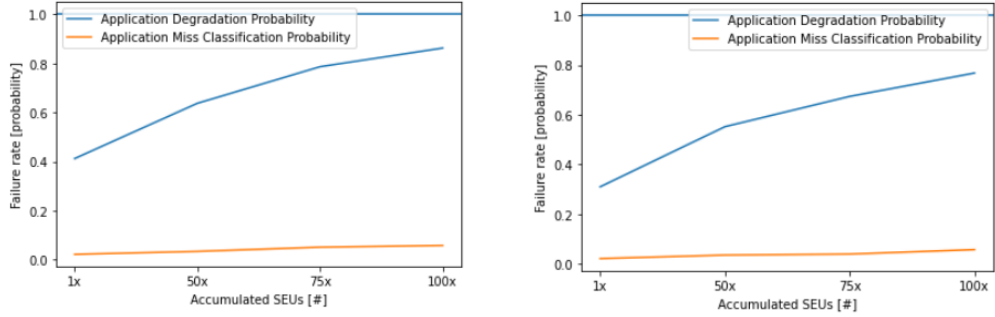


Figure 5.19: Failure rate function MnasNet (a) MnasNet (original) (b) MnasNet (0.7x)

Chapter 6

Conclusions and future work

Given the ever-increasing development of AI and in particular of deep learning in various contexts, we liked to explore its field of application even in a context such as that of the space industry. Space is a hostile environment for humans and devices. Knowing the obstacles met is the first step in implementing defence techniques and methods.

This thesis discussed the issues of a DNN operating in the space industry, caused by the accumulation of radiation on a device over time. In my thesis work, I will focus on the impact of SEUs on the device. Single event upset (SEU) or transient radiation effects in electronics are state changes of memory or register bits caused by a single ion interacting with the chip. In sensitive devices, a single ion can cause a multiple-bit upset (MBU) in several adjacent memory cells. SEUs can become Single-event functional interrupts (SEFI) when they upset control circuits.

In space, the SEUs can cause bit flips, which can affect the parameters of the trained model, causing performance degradation. Goal of this activity is to analyse the SEU effects on neural networks models. Analysis was carried out by simulating fault injections in the configuration memory, causing bit flips. Analysis attention is focused on the inference phase. The analysis was conducted at a high level, based on some tools, such as Pytorch that makes available model archives that are pre-trained and pre-packaged, ready to be served for inference. In particular, for

the analysis we referred to models that satisfy the image classification tasks. Data are represented using 32-bits floating-point representations, accordingly with the PyTorch model of the network.

The first part of my thesis focuses on analyzing and studying some neural network models (NN), knowing which DNN work best and what are the strengths to create an architecture that is best suited to an environment teeming with failures. The potential of the NN is also this, to study and design new models that optimize and improve the performance of some tasks, we will also study the tolerance limit of the network after which the model has completely broken its functioning. Usually to measure the robustness of a circuit there are radiation tests, which however are very expensive, fault injection techniques based on analysis tools or software programs represent a valid alternative to evaluate the reliability of a project, safety and fault coverage.

This activity is based on the design and development of the fault injection platform. The methodology is based on a Monte Carlo algorithm that generates distributions of SE-s within all neural network parameters and verifies if the analysed circuit is able to cope with the accumulation of their effects within the DNN. This methodology can be used as a measure to estimate resiliency of neural networks.

Thanks to this analysis, we could set a limit threshold of failures in order to re-program the device or apply refresh mechanisms to restore the initial weights and guarantee that the application always works accurately.

The second part of my thesis explores some fault tolerance techniques. Fault Tolerance is a crucial property for neural networks to ensure reliable computation for a long duration with graceful degradation over time. My method proposed to reduce the number of parameters in DNN to restrict the probability of small parameters being altered.

Results have shown that pruning not only improves the resource efficiency of neural networks, but also the resilience against faults. In particular, we found that the resilience depends on the level of pruning, the model retains higher resilience where less trainable parameters remained after pruning.

The choice to investigate the image classification task is due to the limitations of computational resources since the analysis has a very high computational cost. In a future scenario, we would like to explore how object detection models perform

with the accumulation of radioactive doses. Other future scenarios suggest other fault-tolerance techniques, we could explore different approaches to the problem. One scenario would be to train the model simultaneously with random faults. Another approach involves the study of a new architecture by applying more classical resilience techniques such as the replication of the most important neurons. An aspect that must certainly be taken into account concerns the generalization of the model to make the network more robust and intrinsically more resilient to errors. The second part of the network will allow the exploration of methods and technologies that make the model more tolerable and resilient, trying to increase the tolerability limit, this phase opens the way to a vastness of future work to create models of NN that could open the way for the use of this new science in a context such as space. Opening the way to infinite scenarios can lead devices to be more autonomous and improve the experience of space exploration by allowing us to know new boundaries and new discoveries to enrich our knowledge of the universe around us.

Bibliography

- [1] Vivek Kothari, Edgar Liberis, and Nicholas D. Lane. *The Final Frontier: Deep Learning in Space*. 2020. arXiv: 2001.10362 [eess.SP] (cit. on p. 13).
- [2] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. *DeepLaser: Practical Fault Attack on Deep Neural Networks*. 2018. arXiv: 1806.05859 [cs.CR] (cit. on pp. 14, 45).
- [3] Jakub Breier, Dirmanto Jap, Xiaolu Hou, Shivam Bhasin, and Yang Liu. *SNIFF: Reverse Engineering of Neural Networks with Fault Attacks*. 2020. arXiv: 2002.11021 [cs.CR] (cit. on p. 14).
- [4] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. «Fault injection attack on deep neural network». In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 131–138. DOI: 10.1109/ICCAD.2017.8203770 (cit. on pp. 14, 45).
- [5] Zheyu Yan, Yiyu Shi, Wang Liao, Masanori Hashimoto, Xichuan Zhou, and Cheng Zhuo. *When Single Event Upset Meets Deep Neural Networks: Observations, Explorations, and Remedies*. 2019. arXiv: 1909.04697 [cs.LG] (cit. on p. 15).
- [6] Fei-Fei Li, Andrej Karpathy, and Justin Johnson. «CS231n: Convolutional Neural Networks for Visual Recognition». In: (). URL: <http://cs231n.stanford.edu/> (cit. on p. 24).
- [7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: <https://doi.org/10.1145/3065386> (cit. on p. 25).

- [8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV] (cit. on p. 26).
- [9] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. *MnasNet: Platform-Aware Neural Architecture Search for Mobile*. 2019. arXiv: 1807.11626 [cs.CV] (cit. on p. 27).
- [10] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. DOI: 10.1109/JPROC.2017.2761740 (cit. on p. 40).
- [11] P.W. Protzel, D.L. Palumbo, and M.K. Arras. «Performance and fault-tolerance of neural networks for optimization». In: *IEEE Transactions on Neural Networks* 4.4 (1993), pp. 600–614. DOI: 10.1109/72.238315 (cit. on p. 42).
- [12] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. «Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures». In: *Proceedings of the IEEE* 100.11 (2012), pp. 3056–3076. DOI: 10.1109/JPROC.2012.2188769 (cit. on p. 45).
- [13] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. «Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors». In: *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 2014, pp. 361–372. DOI: 10.1109/ISCA.2014.6853210 (cit. on p. 45).
- [14] Muhammad Shafique, Theocharis Theocharides, Christos-Savvas Bouganis, Muhammad Abdullah Hanif, Faiq Khalid, Rehan Hafiz, and Semeen Rehman. «An overview of next-generation architectures for machine learning: Roadmap, opportunities and challenges in the IoT era». In: *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018, pp. 827–832. DOI: 10.23919/DATE.2018.8342120 (cit. on p. 45).
- [15] Haissam Ziade, Rafic Ayoubi, and R. Velazco. «A Survey on Fault Injection Techniques». In: *Int. Arab J. Inf. Technol.* 1 (Jan. 2004), pp. 171–186 (cit. on p. 47).

- [16] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, S. Adve, Christopher W. Fletcher, I. Frosio, and S. Hari. «PyTorchFI: A Runtime Perturbation Tool for DNNs». In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)* (2020), pp. 25–31 (cit. on p. 48).
- [17] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. «LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults». In: *2015 IEEE International Conference on Software Quality, Reliability and Security*. 2015, pp. 11–16. DOI: 10.1109/QRS.2015.13 (cit. on p. 48).
- [18] Chun-Kai Chang, Guanpeng Li, and Mattan Erez. «Evaluating Compiler IR-Level Selective Instruction Duplication with Realistic Hardware Errors». In: *2019 IEEE/ACM 9th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2019, pp. 41–49. DOI: 10.1109/FTXS49593.2019.00010 (cit. on p. 48).
- [19] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. *Intriguing properties of neural networks*. 2014. arXiv: 1312.6199 [cs.CV] (cit. on p. 57).
- [20] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. *Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks*. 2016. arXiv: 1511.04508 [cs.CR] (cit. on p. 57).
- [21] Shixiang Gu and Luca Rigazio. *Towards Deep Neural Network Architectures Robust to Adversarial Examples*. 2015. arXiv: 1412.5068 [cs.LG] (cit. on p. 57).
- [22] T. Ito and I. Takanami. «On fault injection approaches for fault tolerance of feedforward neural networks». In: *Proceedings Sixth Asian Test Symposium (ATS'97)*. 1997, pp. 88–93. DOI: 10.1109/ATS.1997.643927 (cit. on p. 58).
- [23] I. Takanami, M. Sato, and Yun Ping Yang. «A fault-value injection approach for multiple-weight-fault tolerance of MNNs». In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*. Vol. 3. 2000, 515–520 vol.3. DOI: 10.1109/IJCNN.2000.861360 (cit. on p. 58).

- [24] Naotake Kamiura, Tejiro Isokawa, and Nobuyuki Matsui. «Learning Based on Fault Injection and Weight Restriction for Fault-Tolerant Hopfield Neural Networks». In: *Proceedings of the Defect and Fault Tolerance in VLSI Systems, 19th IEEE International Symposium*. DFT '04. USA: IEEE Computer Society, 2004, pp. 339–346. ISBN: 0769522416 (cit. on p. 58).
- [25] N. Kamiura, Y. Taniguchi, T. Isokawa, and N. Matsui. «An improvement in weight-fault tolerance of feedforward neural networks». In: *Proceedings 10th Asian Test Symposium*. 2001, pp. 359–364. DOI: 10.1109/ATS.2001.990309 (cit. on p. 58).
- [26] B.E. Segee and M.J. Carter. «Fault tolerance of pruned multilayer networks». In: *IJCNN-91-Seattle International Joint Conference on Neural Networks*. Vol. ii. 1991, 447–452 vol.2. DOI: 10.1109/IJCNN.1991.155374 (cit. on p. 58).
- [27] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. *Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks*. 2018. arXiv: 1805.12185 [cs.CR] (cit. on pp. 58, 73, 75).
- [28] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: 1510.00149 [cs.CV] (cit. on p. 59).
- [29] Stéphane Mallat. «Understanding deep convolutional networks». In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 374.2065 (Apr. 2016), p. 20150203. ISSN: 1471-2962. DOI: 10.1098/rsta.2015.0203. URL: <http://dx.doi.org/10.1098/rsta.2015.0203> (cit. on p. 63).
- [30] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. arXiv: 1602.02830 [cs.LG] (cit. on p. 63).
- [31] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. *Bit-Flip Attack: Crushing Neural Network with Progressive Bit Search*. 2019. arXiv: 1903.12269 [cs.CV] (cit. on p. 66).

- [32] Muhammad Abdullah Hanif, Faiq Khalid, Rachmad Vidya Wicaksana Putra, Semeen Rehman, and Muhammad Shafique. «Robust Machine Learning Systems: Reliability and Security for Deep Neural Networks». In: *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*. 2018, pp. 257–260. DOI: 10.1109/IOLTS.2018.8474192 (cit. on p. 66).
- [33] Faiq Khalid, Muhammad Abdullah Hanif, and Muhammad Shafique. *Exploiting Vulnerabilities in Deep Neural Networks: Adversarial and Fault-Injection Attacks*. 2021. arXiv: 2105.03251 [cs.CR] (cit. on p. 68).
- [34] Bingyin Zhao and Yingjie Lao. «Resilience of Pruned Neural Network Against Poisoning Attack». In: *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. 2018, pp. 78–83. DOI: 10.1109/MALWARE.2018.8659362 (cit. on p. 75).