Architectures to Ensure the Functional Safety of Neural Network based Systems

Stéphane Burel

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Sciences pour l'Ingénieur**

Arrêtée ministériel : 25 mai 2016

Présentée par

Stéphane Burel (doctorant)

Thèse dirigée par **Lorena Anghel (directrice)**
et codirigéee par **Adrian Evans (co-directeur)**

préparée au sein du **Laboratoire CEA/DRT/LIST/DSCIN/LSTA**
dans **l'École Doctorale EEATS**

Architectures pour la sureté de fonctionnement des systèmes à base de réseaux de neurones

Architectures to ensure the functional safety of neural network based systems

Thèse soutenue publiquement le **9 décembre 2022**,
devant le jury composé de :

Panagiota Morfouli
Professeur des Universités, Université Grenoble Alpes, Grenoble INP, Présidente

Alberto Bosio
Professeur des Universités, CNRS, École Centrale de Lyon, Rapporteur

Haralampos-G. Stratigopoulos
Directeur de Recherche, CNRS, Sorbonne Universités, Rapporteur

Arnaud Virazel
Professeur des Universités, CNRS, Université de Montpellier, LIRMM, Examinateur

Kevin Martin
Maître de Conférences, Université Bretagne-Sud, Lab-STICC, Examinateur

Lorena Anghel
Professeur des Universités, Université Grenoble Alpes, CEA, CNRS, Grenoble INP, INAC-Spintec, Co-Encadrante de thèse

Acknowledgments

Je tiens à remercier l'Université française. C'est elle, plus qu'aucune autre, qui permit au fils de roturier que je suis de m'instruire. C'est grâce à elle que j'ai pu me soustraire des méandres du prolétariat le plus précaire pour rejoindre l'aliénation bourgeoise. Mais je suis un homme fidèle ; Jamais je ne l'oublierai.

Je tiens à remercier ma directrice de thèse. Mme Anghel laissera à mon esprit l'image de l'excellence technique, toujours en étant à l'écoute et accessible. Je tiens également à vivement remercier mon encadrant de thèse, M. Evans.

Je tiens à remercier l'ensemble du jury de la thèse. Votre déplacement fut un plaisir. L'ensemble de vos remarques le jour de la soutenance laisse comprendre que vous avez pris cette soutenance avec autant d'importance que je lui en ai accordé moi-même. Vos avis étaient pertinents et m'ont donné envie de continuer ces projets de recherche.

Je tiens à remercier les membres de mon laboratoire d'accueil et plus particulièrement les membres de l'équipe dédiée à l'accélération neuromorphique. En vous chassant du bâtiment dans lequel nous étions alors, notre département n'a fait que nous unir davantage.

Je tiens à remercier M. Jourlin et l'ensemble du Laboratoire Informatique d'Avignon qui m'ont par un été frais comme on n'en verra jamais plus, au détour d'un stage de recherche, donné l'envie de faire cette thèse.

Je tiens à remercier les rencontres faites sur le bord de la route, tous ceux avec qui j'ai sympathisé et que j'espère côtoyer longtemps encore. Je parle particulièrement de mon ami le zadiste maintenant accompli, de celui qui voulait être suisse mais n'en a pas le sou, de celui qui rêve d'avoir le bras long ou encore de celui qui a l'âme pure mais est resté bloqué dans une série TV.

Merci à vous.

List of Figures

List of Tables

ABFT Algorithm-Based Fault Tolerance. 31, 32, 105

AFM Activation Fault Model. 92, 94, 96, 98–100

AI Artificial Intelligence. 3, 6, 10

ASIC Application-Specific Integrated Circuit. 5, 40, 47

ASIL-D Automotive Safety Integrity Level D. 65, 91

AVF Architectural Vulnerability Factor. 46

AVG AVeraGe. 91, 97, 99, 102

BER Bit Error Rate. 13, 23, 31, 36

BERZAD maximum Bit Error Rate with Zero Accuracy Drop. 84–86

bf16 A 16-bit floating point numeric format used in AI accelerators that uses 8 exponent bits and 7 fractional bits. 79, 80, 84, 85, 89

BIST Built-In Self Test. 66

BN Batch Normalization. 7, 11, 21–23

BNN Binarized Neural Network. 19

CNN Convolutional Neural Network. 7, 10, 11, 16, 17, 22, 29, 33, 78–82, 89

CPU Central Processing Unit. 14

CSDC Critical Silent Data Corruption. 99, 100

CSRAM Configuration Static Random Access Memory, Used to store the configuration in an FPGA. 28, 46

DMR Dual Modular redundancy. 14, 51, 65

DNN Deep Neural Network. ii, viii, XVI, 3–42, 46–48, 51, 52, 56–58, 60–62, 64–69, 73, 75, 76, 79–81, 86, 87, 90–93, 95, 97, 101, 102, 105, 106

DRAM Dynamic Random Access Memory. 52

DUE Detected Unrecoverable Error. 13, 26, 27, 93

ECC Error Correction Code. 13, 14, 26, 86, 87, 106

FIT Failure In Time, 1 FIT corresponds to one failure in 1 billion hours. 65

fp32 A 32-bit floating point numeric format defined in IEEE-754. 80, 84–86

FPGA Field-Programmable Gate Array. 5, 12, 15, 19, 25, 28–30, 38, 40, 46, 47, 51, 65, 92

GPU Graphics Processing Unit. ii, 5, 12, 14, 15, 25–31, 33, 40, 46, 47, 51, 57, 65, 92, 99, 101

int8 8 bits integer numeric format. 84–87, 89

IoU Intersection over Union, A metric used to measure the accuracy in image segmentation and object detection. 8

LRN Local Response Normalization. 7, 11, 22, 23

LSB Least Significant Bit. 5, 17, 18, 38, 48, 55, 60, 78, 79, 81, 83–85, 87, 95, 105

MAC Multiply And Accumulate. 12, 31, 34, 51–55, 57, 68, 80

MAX MAXimum. 91, 97, 98, 100, 102

MBU Multiple Bit Upset. 38

MIN MINimum. 91, 97, 99, 102

mIoU mean Intersection over Union, An average of IoU metric for all the considered labels in image segmentation applications. 9

ML Machine Learning. 8

MOC-MOP Multiple Output Channel-Multiple Output Pixel. 53

MOC-SOP Multiple Output Channel-Single Output Pixel. 53

MSB Most Significant Bit. 17, 18, 38, 55, 68, 70, 82, 85, 86, 95

OP Opportunistic Parity. 78, 79, 81–87

OS Output Stationary. iii, iv, 52–56, 59–62, 64–68, 70–74, 105

PE Processing Element. iii, iv, 12, 39, 51–56, 58–60, 62, 64–76, 105

ReLU Rectified Linear Unit. ii, 17, 21, 23, 36, 101

RRAM Resistive Random Access Memory. 38

RS Row Stationary. 52, 53, 55, 56, 59, 61, 62, 68, 73

RTL Register-Transfer Level. XV

SA Stuck-At. 60, 62, 68, 70, 72, 75

SA0 Stuck-At-Zero. 60, 70, 71

SA1 Stuck-At-One. iv, 60, 70, 75, 76

SBED Syndrome-Based Error Detection. 32, 33, 35, 36, 48, 91, 97, 98, 101, 105

SDC Silent Data Corruption. ii, 13, 18–20, 26, 27, 32, 36, 46, 93, 97–100, 102

SEU Single Event Upset. 16

SIMD Single Instruction Multiple Data. 12, 39, 51, 106

SoC System on Chip. 5, 28

SRAM Static Random Access Memory. ii, 15, 25, 27, 28, 30, 37, 46, 51, 52, 92

STD STandard Deviation. 91, 97, 102

TMR Triple Modular redundancy. 14, 36, 38

TPU Tensor Processing Unit. 92

TSDC Tolerable Silent Data Corruption. 99, 100

WFM Weight Fault Model. 92–94, 96, 98, 99

WS Weight Stationary. 52–56, 59, 61, 62, 68, 73, 74

Contents

1 Introduction

Deep Neural Networks are the current cornerstone of various modern artificial intelligence applications [LeCun2015]. The growing number of these applications is due to the breakthroughs they have achieved in diverse fields [Otter2021, Feng2019], where Deep Neural Networks may exceed human accuracy. Safety-critical applications are no exception. In these applications, the system must be resilient to unexpected events, such as hardware faults. This chapter introduces the context of the thesis and the need for designing robust Deep Neural Networks.

Deep Neural Network (DNN)s are used in many modern Artificial Intelligence (AI) applications [LeCun2015]. Due to the breakthroughs that DNNs have achieved in natural language processing [Deng2013, Collobert2008] and image recognition [Krizhevsky2012], the number of applications employing DNNs has exploded.

The study of the tolerance of DNNs to hardware failures is important for many reasons. DNNs are already being deployed in critical tasks, where a failure of the system can result in significant physical harm or financial liability. It is estimated that 25% of passenger car miles will be driven by autonomous robotaxis by 2042 and their autonomous driving compute systems will employ multiple DNNs [IDTechEx2022]. It is imperative to ensure that hardware faults in these systems do not result in incorrect decisions, as human safety is at risk. We also note that DNNs are being employed in medical diagnostics [Ronneberger2015], another application where human lives could be impacted by erroneous computations. Even military equipment, such as missiles, are now employing DNN technology [He2021]; a domain where an incorrect result could be catastrophic.

Although neural networks continue to gain in accuracy for many applications, this comes at the cost of a huge number of computations, and thus high power consumption. One of the promising approaches to reduce power, is to operate the compute system at lower voltage [Wu2021a, Yang2017], however, this tends to introduce more faults as the circuit is operating with tighter margins. If DNNs can be made tolerant to hardware faults with minimal cost, low-voltage operation remains a promising path for reducing energy consumption, despite the higher rate of faults. Post-CMOS process technologies (e.g. [Hills2014]) are promising option for future NN accelerators, but as they are known to prone to faults, the system must be designed accordingly.

Sometimes, it is believed that DNNs are intrinsically fault tolerant – as the algorithms are bio-inspired. As will be seen in this thesis, the reality is more complex. Many hardware faults have no impact, but there exist critical faults which can cause a DNN to degenerate to random guessing. Understanding and identifying these critical faults is one of the focuses of this thesis.

In industry today, when DNN accelerators must be fault tolerant, such as in existing assisted and autonomous driving systems, manufacturers rely on hardware redundancy. For example, cars manufactured by Tesla incorporate two fully independent Fully Self Driving (FSD) chips along with their own power subsystem, DRAM, and flash memory for redundancy. Each chip boots up from its own storage memory and runs its own independent operating system. These chips are part of a safety system that incorporates dual-core lockstep [Talpes2020]. This is clearly a costly approach to achieve high-reliability - particularly when we consider that neural networks never achieve 100% accuracy, even in the absence of faults. Redundancy based approaches are clearly incompatible with low-power operation.

The goal of this thesis is to provide new insights into how faults alter the behavior of DNNs and to propose new, low-cost and innovative means to improve their robustness.

This thesis is structured as follows :

- **Chapter 2** provides an overview of the current scientific literature in the field of DNN fault tolerance, and also gives the necessary background to help understand this thesis.
- **Chapter 3** presents a study of the impact of faults in different DNN systolic accelerators.
- **Chapter 4** covers a new systolic accelerator specifically designed for fault tolerance.
- **Chapter 5** presents a method to protect the contents of weight storage memory with no storage overhead.
- **Chapter 6** describes a statistical technique to detect faults occurring in a DNN by monitoring its data-flow.
- **Chapter 7** concludes this manuscript and presents ideas for future work.

2

Analysis of DNN Robustness - State of the Art

Deep Neural Networks have been widely deployed as they are able to automate and to exceed human capabilities in many important tasks. They are frequently deployed in safety critical systems, where hardware fault tolerance is important. This chapter presents an overview of the existing works on the reliability of Deep Neural Networks hardware accelerators.

Contents

Deep Neural Network (DNN)s have specificities that must be taken into consideration when studying their fault tolerance. For example, the accuracy of DNNs is never perfect , even in the absence of faults, and a certain level of inaccuracy is usually acceptable [LeCun2015]. Furthermore, due to their structure, DNNs have intrinsic redundancy [Yvinec2021] and their activation functions, for example, are able to mask certain errors [Hoang2020]. These particularities of DNNs make them different from traditional computer systems, in terms of their robustness.

However, many other parameters associated with a DNN must be considered. Not only activation functions, but also pooling layers, normalization layers [Li2017] and also the nature of computing layers plays a role in the robustness of a DNN [Xu2019]. Even the training process can alter the fault tolerance capability of a DNN [Zhang2019]. Computational optimizations and the choice of data-type can impact the fault tolerance of a DNN. Another example is the quantization of a DNN, that can improve its robustness by several orders of magnitude [Gambardella2019]. Looking at all these parameters, it becomes obvious that evaluating the fault tolerance of DNNs is a complex task. In the same time, the overall reliability of a system depends on the hardware platforms and the executed application.

To avoid dealing with the underlying hardware platform, some authors considers abstract DNN, and study the fault tolerance of DNNs without considering the complexities of the hardware platform.

Publications on DNN fault tolerance focus on specific hardware platforms, such as Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), Application-Specific Integrated Circuits (ASICs) [Reuther2021], each of them having their specific failure modes when executing DNNs.

The special characteristics of DNNs can be exploited to provide low-cost fault tolerance. As an example, techniques such as redundancy currently used in state-of-the-art automotive System on Chip (SoC) solutions [Talpes2020, Matsubara2021] protect all data bits, whereas DNNs are highly tolerant of faults in their Least Significant Bits (LSBs) [Malekzadeh2021].

The scientific literature on the fault tolerance of DNNs is vast. Three survey articles have recently been published [Torres-Huitzil2017, Ibrahim2020, Mittal2020]. These surveys provide an overview of the works in this field of research.

This chapter presents a concise review of the existing state of the art. The remainder of this chapter is organized as follows :

- **Sec.2.1** presents the background for understanding this thesis.
- **Sec.2.2** presents existing works that analyze the robustness of DNNs.
- **Sec.2.3** presents existing techniques to improve the robustness of DNNs.
- **Sec.2.4** quantitatively identifies the current focus of the scientific literature in terms of case studies, hardware platforms, fault models, and fault detection and mitigation.
- **Sec.2.6** situates the contributions of this thesis relative to the existing works.
- **Sec.2.5** concludes this chapter.

2.1 Background

This section presents background material that is helpful for understanding this thesis, and is structured as follows : Sec.2.1.1 presents the background on DNNs. Sec.2.1.2 presents the background on DNN hardware accelerators. Sec.2.1.3 presents the background on hardware faults.

2.1.1 Background of DNNs

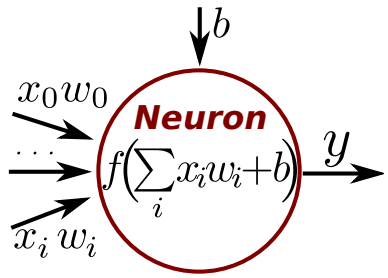
Artificial Intelligence and DNNs

The field of Artificial Intelligence (AI) is wide and covers many different fields of study. The term AI was coined by John McCarthy, as "the science of creating machines that have ability to achieve goals like human do" [McCarthy2006].

Machine Learning is a sub-field of AI that has recently stimulated scientific interest, and was coined by Arthur Samuel as "the ability for a computer to learn to perform better at a task than the person who programmed the computer" [Samuel1959]. Recent breakthroughs have been made possible by another sub-field of Machine Learning, DNNs.

Artificial Neuron

Artificial neural networks are mathematical models that are inspired from the study of biological neural networks. Biological neurons are excited by stimuli from other neurons via theirs synapses. When this excitation reaches a threshold, a biological neuron emits a spike that reaches other neurons via other synapses. In artificial neural networks, a neuron computes a weighted sum of its inputs synapses, performs a non-linear function on the result, and then this result, called an activation, is passed to other neurons, as illustrated in Fig. 2.1.



x,w,b,f,y are respectively inputs, weights, bias, activation function and activation value

Figure 2.1: Artificial Neuron

Activation function

A non-linear activation function at the output of neurons is essential for the learning process, and required to build multi-layer networks. Many activation functions exist, but in this thesis, we only use the ones summarized in Tab.2.1. In the subsequent sections, the output of the activation function will be referred as an activation.

Different layers

A DNN consists of layers, and all the neurons in a given layer are evaluated simultaneously. The layers can have different topologies and the most discussed in this thesis are the following :

- **Fully connected** topology implies that every neurons in the current layer propagates its activation to every neurons in the subsequent layer. This structure is typically found in the last layers of a classification DNN.

- **Convolutional layer** implies that a group of neurons in that layer (typically a small 3x3 or 5x5 region) is connected, via a shared set of weights, to the neurons in the next layer. This set of weights is called a kernel. In terms of implementation, convolutional layers require fewer weights, because the same kernel pattern is shared between neurons.

Name	Function	Derivative	Range
Sigmoid	$\frac{1}{1+e^{-x}}$	$g(x)(1-g(x))$	$(0,1)$
ReLU	$\begin{cases} 0 \text{ if } x \leq 0 \\ x \text{ if } x > 0 \end{cases}$	$\begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x > 0 \\ \text{undefined if } x = 0 \end{cases}$	$[0,\infty)$
ReLU6	$\begin{cases} 0 \text{ if } x \leq 0 \\ x \text{ if } x > 0 \\ 6 \text{ if } x \geq 0 \end{cases}$	$\begin{cases} 0 \text{ if } x < 0 \\ 1 \text{ if } x > 0 \\ \text{undefined if } x = 0 \end{cases}$	$[0,6]$

Table 2.1: Activation Functions Used in this Thesis

Pooling Layers

Convolutional Neural Networks(CNNs) can include pooling layers along with convolutional layers. Pooling layers reduce the dimensions (height and width) of the layer by combining a cluster of neurons in one layer into a single neuron in the next layer. Pooling layers are typically used to combine small windows of 2 x 2 neurons. Two pooling layer are frequently used :

- **Max** Pooling forwards the maximum value of each local cluster of neurons to the next layer.
- **Average** Pooling forwards the average value of the local cluster of neurons to the next layer.

Normalization layers

During training, the distribution of values in each layer changes when the weights of the previous layers are altered. This significantly slows the training by requiring lower learning rate and makes the training of networks with many layers difficult.

This phenomenon, coined as internal covariate shift by [Ioffe2015], is addressed with Batch Normalization, which is a trainable layer typically used after each convolutional layer which normalize the values of all neurons in a layer for each training mini-batch. As a result, Batch Normalization (BN) allows to use significantly higher learning rate thus improving the training time, and facilitating the training of deep networks. Consequently, BN is generally used in state of the art DNNs [Howard2019].

During the inference process, the trainable parameters of the BN layers are frozen. At this point, BN is simply a linear transformation of activations and hence can be merged with the previous convolutions.

Another normalization layer, the Local Response Normalization (LRN) is a layer that behaves like the Average Pooling layer by normalizing several locally neighbors neurons. Instead of average pooling layers, LRN layers are not used to change the shape of a layer and usually compute an average of neighboring pixels on the three axis (X, Y, Z) of the input layer. While no longer used in recent DNNs, some of the networks that are studied in this thesis have this type layer.

Training

As with all Machine Learning algorithms, DNNs are trained from input data. Various types of learning exist such as reinforcement learning that trains a DNN in an environment in order to maximize the cumulative reward, or unsupervised learning that learns patterns from unlabeled data. In this document, we mainly consider supervised learning.

In supervised learning, to each data is assigned a value (single or set of label or continuous value). The behavior of the network is evaluated through a cost function that compares the expected behavior for the input data and the actual behavior of the network. The value of the cost function is called the loss.

The training process alters the weights and biases of the network in order to reduce the loss.

During training, the weights are updated with the gradient descent process. After each iteration of the training process, each weight is updated with gradient descent depending on its partial derivative of the loss. As the iterations of training progress, the loss should be reduced and the network should converge.

After the network is trained, its weights are frozen and it can be deployed to analyze data that has not been used during training. If the training was effective, the DNN will behave as expected on these new data. This is called the inference.

Dropout

Dropout is a generalized technique performed during the training of state-of-the-art classification DNNs as it improves the overall accuracy by reducing over-fitting [Srivastava2014].

Dropout is applied on certain layers, typically the last layers of a DNN. For each training iteration, a predefined fraction of the neurons are randomly disconnected from the learning process. During the forward pass, the outputs of these neurons are set to 0. During the backward pass, the weights of these neurons are not modified. At the next training iteration, the neurons are reconnected, and new neurons are randomly selected for the Dropout process.

Data-sets

The data-sets used to train DNNs used in this thesis are presented in the following and cover three types of tasks :

- **Image classification** relies on data-sets where a label is attributed for each input image. For example, the ImageNet dataset contains images of different classes of objects, such as, for example, teapots. In a DNN trained for classification, a label is associated with each output neuron. For each prediction, each neuron outputs the probability for the input image to be related to its assigned label. The networks accuracy is evaluated based on its ability to assign labels to images that the DNN has not processed during training. Two metrics are used to compute the accuracy of the DNNs in this thesis.
  - **Top-1**, where the neuron with the highest probability is considered to be the prediction of the model.
  - **Top-5**, where the model is considered accurate if one of the five neurons with the highest ranked activation value corresponds to the correct label.
- **Object detection** relies on data-sets where the DNN is trained to detect, localize and assign a label to objects in the input images. The accuracy is computed with the Intersection over Union (IoU) metric, that compares the number of pixels that are in the region identified by the network and the actual object, divided by the total number of pixels. This metric give a value between 0 and 1. A threshold is then applied (typically 0.5 to 0.95). If the detected object is detected with an IoU above the threshold and is assigned the label correctly, then

this prediction is considered correct. The accuracy is then evaluated with the metric called precision and recall functions:

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive + False Positive}}$$

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive + False Negative}}$$

- **Image segmentation** is a task where the network attributes a label to each pixel in the image. The output of a DNN trained for image segmentation is an image comparable to the input image where each output pixel corresponds to a label associated with corresponding pixel in the input image. The metric for this task is the mean Intersection over Union (mIoU). mIoU is the average of the classic Precision/Recall for all the classes. For each class, individual Precision/Recall is computed where true positive is set as the number of pixels correctly labeled, true negative set as the number of pixels correctly not labeled, and false negative as the number of pixels that were not labeled when they should have.

Name	Task	Type	Size	Input	Output
MNIST	Classification	Handwritten digits	60 K	28x28 gray pixels	10 labels
CIFAR-10	Classification	Pictures	60 K	32x32 RGB	10 labels
CIFAR-100	Classification	Pictures	60 K	32x32 RGB	100 labels
ImageNet	Classification	Pictures	14 M	Variable	1000 labels
GTSRB	Classification	Road-signs	52 K	1360×1024 RGB	43 labels
PascalVOC	Classification, Detection	Pictures	500 K	Variable	Bounding Boxes
CalTech	Classification, Detection	Road Video	250 K frames	640x480	Variable
COCO	Detection, Segmentation	Pictures	328 K frames	640x480	Variable
WoodScape	Detection, Segmentation	Pictures	8 K frames	3266x2450	Variable

Table 2.2: Data Sets Discussed in this Thesis

Compression Techniques for DNNs

To achieve high accuracy on well-known data-sets, DNN topologies have grown deeper, requiring a huge number of computations for each inference operation.

Two well known techniques have been developed to reduce the number of computations required to evaluate a network.

- **Pruning** consists in removing unnecessary parameters and computations from a DNN during inference [Liu2018].

  The idea of pruning was made after the observation that, once trained, over-sized networks typically have a large number of weights whose values are close to zero or a large number of layers where the data-flow is sparse (many activations are close to zero).

  A pruning algorithm typically relies on a three-stage process : 1) Training the model. 2) Pruning the model by removing weights close to zero. 3) Retraining the network to adapt to the weight lose.

  There exist two techniques for pruning.

  - **Irregular Pruning** where weights are simply removed based on their value, without considering the target hardware.
  - **Structured Pruning** where the selection of weights to remove is made, taking into account the target hardware.

- **Quantization and Fixed Point Data-Types** improves the power efficiency of a DNN accelerator [Sze2017].

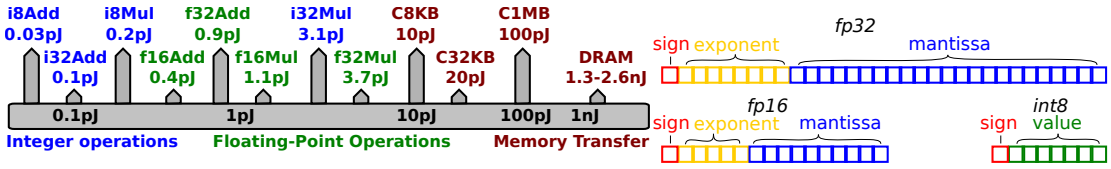
  The energy required to perform an arithmetic operation varies with the bit-width and the choice of operation (addition or multiplication on integers or floating-points). As illustrated in Fig.2.2, low bit-width operations on integer cost significantly less energy than equivalent operation on standard precision floating point values.

  Consequently, a DNN that is initially trained using floating-point values, can be converted to operate with integer weights and activations, resulting in a significant improvement in energy efficiency and memory footprint.

  Two strategies exists : The conversion of the DNN to fixed-point arithmetic and the quantization of the DNN.

  The conversion to fixed point changes the data representation without significantly altering the numerical values of the DNNs. Fixed-point computations are representing fractional values as integer multiples of some fixed small units. Fixed-point arithmetic operations are simpler and require less energy than floating point.

  Alternatively, the values can be converted to fixed-width integers, by a quantization process is more complex but allow significant bit-width reduction without altering the accuracy of DNNs [Yang2019]. To convert the original weights to integers, they must be re-scaled to the range of the integers. After re-scaling, it is necessary to perform additional training to correct the artifacts of the quantization. Like the fixed-point conversion, the quantization convert the weight or/and activations to integers. However, the quantization individually scales all the layers to use the maximal possible ranges of discretization allowed by the bit-width.



(a) Rough Energy Costs for Various Operations
Figure adapted from [Horowitz2014] with Logarithmic scale for measurement in 45nm 0.9V.

(b) Three Numerical Formats Commonly Used for DNNs

Figure 2.2: Improvement in Power Efficiency of DNNs with Reduced Precision

DNN Models

Many DNN models have been developed over the pact decade. This section briefly presents the most discussed models in this thesis.

- **LeNet** was one of the first CNNs and is widely used in many studies. It achieved outstanding progress in recognition of handwritten digits compared to the models of its time. The most famous version, LeNet-5, contains two convolution layers and three fully connected layers. As it is fast to train, LeNet is often used for an initial proof-of-concept in AI research. It is, however, orders of magnitude smaller than modern DNNs.

- **AlexNet** was the first CNN to participate in the ImageNet contest, and it outperformed all other non-DNN-based algorithms at the time. Although it introduced the LRN layers, AlexNet is not significantly different from LeNet-5 and can be seen as an eight-layer version of LeNet-5 adapted to a much harder dataset. CaffeNet is derived from AlexNet.
- **VGG** goes significantly deeper and proposes up to 19 layers. The most used VGG version is VGG-16, that is limited to 16 layers. VGG popularized the use of small kernels of size 3x3, which are still used in several modern DNNs. VGG is very big in term of model size and the number required computations. Researches found that to improve over VGG, it was not sufficient to simply add more layers, resulting in other strategies.
- **Inception** also referred as GoogleNet, added the concatenation of several layers to perform different size filters to the same layer. Each Inception layer is the concatenation of several convolutional layers on the same input, performing convolution filters of different sizes. Inception was one of the first DNNs to use the now globally accepted BN.
- **ResNet** introduced residual layers, that shortcut several layers to finally be concatenated with the results of parallel layers. This technique, combined with BN, made it possible to train much deeper networks, with depths up to 152 layers in the original paper.
- **MobileNet** is a DNN designed for mobile and embedded applications. Its success paved to way for other compressed models. MobileNet introduced depthwise convolutions, in which, instead of being the result of the convolution on all input channels, one output channel specifically focus on a single input channel. Three different versions of MobileNet currently exists but existing studies on fault tolerance currently only consider MobileNetV1 and MobileNetV2.
- **SqueezeNet** is another type of compressed DNN. SqueezeNet introduced the fire module, which is a mix of squeeze convolutional layer (that has 1x1 filters), followed by an expand layer (that has a mix of 1x1 and 3x3 filters), which can be seen as a light-weight version of Inception modules.
- **DeepLabV3** is a fully CNN designed for image segmentation. Built on top of a MobileNet or ResNet feature extractor, DeepLabV3 module uses dilated convolutions [1] (filters where the input pixels are spread out) to perform large filters without increasing the kernel size.
- **A few others DNNs** are referenced in this thesis, but the most frequently referenced networks are summarized in Tab.2.3

Name	# of layers	# of weights	Accuracy
LeNet-5	5	60 K	-
AlexNet	8	61 M	84.6%
VGG-16	16	138.4 M	90.1%
ResNet50	50	25.6 M	92.1%
InceptionV1	27	5 M	88.9%
MobileNetV1	28	4.2 M	89.9%
MobileNetV2	53	3.4 M	91.1%

Table 2.3: The Most Considered DNN Models in this Thesis
Accuracy is the Top-5 for ImageNet.

[1] Dilated convolutions can be equivalently referred as atrous convolutions.

2.1.2 Background of DNN Hardware Accelerators

The increasing usage of DNNs was historically made possible due to the development of GPUs that offer huge compute power [LeCun2015]. Other types of integrated circuits have also been used to execute DNNs and these are summarized below.

Graphics Processing Units

GPUs are parallel architectures that initially employed Single Instruction Multiple Data architectures for accelerating rendering of 2D and 3D graphics. The hardware architectures have evolved and they are used for many computational intensive tasks, including training DNNs.

Indeed, some modern GPUs are specialized for DNNs. NVIDIA has released new GPUs (e.g. Volta architectures) that embed special components for DNN acceleration, as well as software libraries (CuDNN).

Field-Programmable Gate Arrays

Modern FPGAs contain programmable logic (PL) resources consisting of logic blocks that can implement combinatorial functions and sequential elements. They also contain dedicated random access memory blocks and specialized, but configurable, compute units for arithmetic operations. All these components are wired together with a reconfigurable interconnects.

FPGAs are increasingly used as DNN accelerators due to their reprogrammability, their power efficiency and their ability to perform massive parallel computing [Blaiech2019].

Systolic Architectures

A systolic computer consists of an array of compute units where data flows between neighboring units, reducing the number of external memory accesses. The evaluation of DNNs requires regular computations, such as convolutions and matrix multiplications, which can be mapped to systolic hardware. In a systolic DNN accelerator, typically, each compute unit, called a Processing Element (PE), can perform a multiply-accumulate Multiply And Accumulate (MAC) operation. There are different ways to map DNNs to systolic architectures [Sze2017].

2.1.3 Background on Hardware Failures

In this section, we present a brief overview of hardware faults, how they can be modeled, and common techniques for detection and mitigation.

Hardware Failures

A hardware fault is the result of a physical phenomena which alters the normal operation of the hardware. A fault may be masked, if the output of the hardware unit is not used, at the time when the fault occurs. If the fault alters a variable or a state used in the circuit, then an error has occurred. If an error results in a system failing to achieve its objective, this is classified as a failure.

Fault can be loosely classified into three categories.

- **Permanent faults** are the result of a permanent change in the hardware and are persistent.
- **Transient faults** only impact the behavior of the hardware for a limited period of time. They may be caused by an external disturbance, such as an ionizing particle.
- **Intermittent faults** occur when the same fault occurs periodically, typically because, the operating conditions of the circuit are not stable - for example if a it is operating at an excessively low voltage.

Impact of Hardware Faults

A fault does not necessarily result in an error [Mukherjee2005a]. As the effect of a fault propagates, it may be masked. In compute based systems, the impact of faults is often classified as shown below :

- **Benign fault** is a fault that is masked and does not impact the program output.
- **Corrected fault** is a fault detected and corrected by a fault tolerant mechanism, typically by an Error Correction Code (ECC).
- **Detected Unrecoverable Errors(DUEs)** occur when a fault is detected, but the system can not correct the data. In this case, the system is aware of the fault.
- **Silent Data Corruptions(SDCs)** occurs when a fault is not detected, and the final result of the computation is incorrect. This is the most serious consequence of a fault.

Bit Error Rate (BER) is a metric to measure the ratio of erroneous bits in a data stream or in an unreliable memory. In the context of this thesis, BER is often used to quantify the rate of errors in a weight storage memory for a DNN model.

Fault Models

A fault is the result of a physical phenomena, however, to evaluate the impact of faults, their effects are typically abstracted using a fault model. For example, a fault induced by ionizing radiation can be simplified and modeled as a single bit-flip.

An abstract fault occurring during the evaluation of a DNN can be described by a tuple consisting of:

- **Which** component is impacted by the fault (e.g. weight memory, arithmetic unit, ...).
- **When** the fault occurs
- **What** effect has the fault (e.g. bit-flip, stuck-at-zero, ...).

In this thesis, the focus is on an architectural study of the impact of faults on DNN accelerators. We have thus adopted high-level fault models which can be easily evaluated. Although such models do not necessarily reflect the exact behavior of specific hardware faults, they are sufficient to compare the robustness of different accelerator architectures.

In our fault models, we have considered the following cases :

- **Bit-flips** model the effect of transient faults. Bit-flips can occurs on a single bit or on multiple, bits that are adjacent. The bit-flip is the most commonly studied model.
- **Stuck-at** models the effect of permanent faults. Stuck-at faults alters the value of one signal to a specific value that remains static (e.g. an output of an adder always being one).

Alternatively, another fault model is considered in existing works :

- **Noise** can be used to model different phenomena, such as inaccuracy in analog-based accelerators, adversarial attacks (human-based alteration of the DNN), and may even be used as very high level of abstraction of transient fault on quantized DNN. The phenomena modeled by noise are out of the scope of this thesis, but we briefly discuss about few articles that use this fault model.

Injection of Actual Faults

When there is an accurate model (hardware description, etc...) of the device under study, the effects of faults can be studied. When this is not the case, such as for commercial GPU or CPU, it may be necessary to inject real faults in the actual hardware.

This thesis examine the existing studies that have performed the following fault injections :

- **Radiation experiments** where an integrated circuit placed in a beam can be used to evaluate the effect of radiation in a complex integrated circuit.
- **Timing faults** are injected by increasing the clock frequency of a component above its maximal threshold.
- **Voltage faults** are injected by decreasing the voltage below the minimum value required for correct operation.

Real-world fault injection, for example through beam experiments, have the advantage that the entire design is subject to faults. As a drawback, such experiments offer limited visibility as there is no way to precisely know where specific faults occurred. Instead, only the high-level impact on the application can be observed.

Fault Detection, Fault Tolerance and Mitigation Strategies

The following strategies for improving the robustness of applications are discussed in this thesis.

- **Fault masking** removes the alteration of the behavior caused by a fault
- **Error mitigation** reduces the alteration of the behavior caused by an error
- **Redundancy** is a commonly employed fault detection and mitigation strategy, that relies on the duplication of components to detect a fault. Temporal and physical redundancies are considered. Dual Modular redundancy and Triple Modular redundancy (TMR) are often used as baseline when comparing the hardware overhead of fault mitigation techniques.
- **Fault tolerant training** aims to improve the fault tolerance of a DNN performing inference, by modifying the training procedure.
- **Razor flip-flop** is a flip-flop augmented with a shadow latch which can detect timing faults.
- **Error Correction Code (ECC)** is used for controlling errors in data over unreliable or noisy communication channels and memory storage. The central idea is to encodes the message with redundant information in the form of a code.

2.2 Understanding the Robustness of DNNs

This section presents works from existing studies that analyze the robustness of DNNs without proposing methods of improvement. This section is structured as follows :

- **Sec.2.2.1** presents the studies that focus on abstract DNNs that do not consider hardware DNN accelerators.
  - **Historical Works** briefly presents the early papers in the field of fault tolerance of DNNs.
  - **Bimodal Behavior of Floating-Point-based DNNs** presents the criticality of floating-point-based DNNs.
  - **Role of Data Types** presents the key role that numerical formats play in the robustness of DNNs.
  - **Impact of Pruning** presents the impact of this DNN compression method on fault tolerance.
  - **Layer-Level Analysis** presents the works that analyze which layers are most critical.
  - **Fault Tolerant DNN Mechanisms** presents the inherent error masking ability of DNNs.
  - **Sensitivity of Weights and Activations** presents studies considering faults in weights and activations.
  - **Estimation of the Robustness of DNNs** presents methodologies for evaluating the robustness of DNNs without using fault injection.
  - **Conclusions of the robustness of abstract DNNs** concludes this section.
- **Sec.2.2.2** presents the existing works that focus on the robustness of actual hardware DNN accelerators.
  - **Robustness of GPUs** presents the robustness of DNNs executed on GPUs.
  - **Robustness of Static Random Access Memory (SRAM) Cells** considers the potential for the power supply reduction of SRAM Cells that may be possible with DNNs.
  - **Robustness of FPGA-based DNN accelerators** presents the robustness of DNNs executed on FPGAs.
  - **Conclusions of the robustness of hardware DNN accelerators** concludes this section.

2.2.1 Robustness of Abstract DNNs

This section presents the existing studies that consider faults in mathematical models of DNNs.

As illustrated in Fig.2.3, the same DNN model can be executed on many different hardware platforms. This is one of the challenges when studying the fault tolerance of DNNs : Hardware faults are dependent on the architecture and implementation technology and may vary across different hardware DNN accelerators. To draw general conclusions on the fault tolerance of DNNs, a strategy used in the existing works consists of considering the DNNs without a specific hardware platform. Instead of performing realistic fault injection on models of actual hardware, faults are injected into the abstract DNN model, typically using weights or activations as target of faults.

We propose to call this approach the study of abstract DNN. This section presents the conclusions that can be drawn from this field of study.

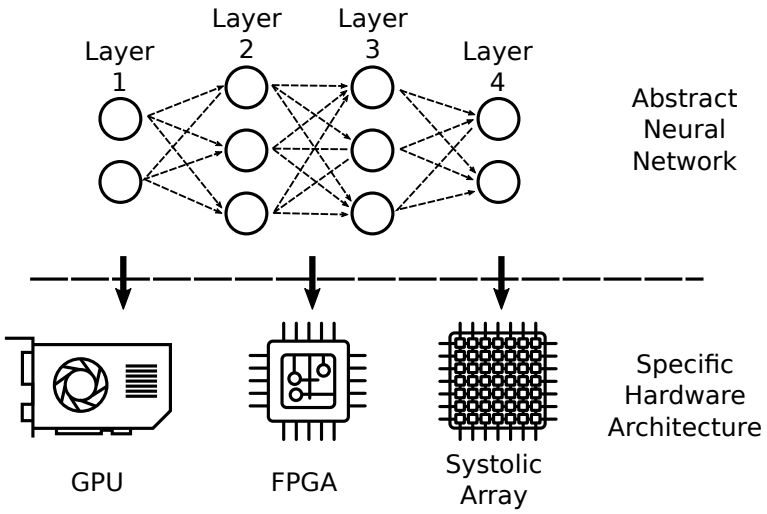


Figure 2.3: Difference between Abstract and Hardware-Specific Deep Neural Networks

Historical Works

Observation : The first scientific publications that focused on the fault tolerance of neural networks appeared in the 1990s, and most of them were based on abstract neural networks.

Notably : Clay and Sequin (UC Berkeley) published several papers on the fault tolerance of neural networks. In [Clay1992], the authors randomly removed from one to three hidden neurons during the training. They show that this technique not only makes the resulting network more tolerant to these faults during inference, but they show that it prevents over-fitting and thus improves generalization. This is comparable to the well-known drop-out technique that is still widely used in modern networks.

Furthermore : At this time, other authors explored different fault models. [Nijhuis1990] specifically studied the extent to which a network can be made tolerant to faults in the weight storage memory by including such faults during the training phase. At the same time, in Japan, researchers were looking at the impact of faults during training. [Alippi1995] studied a three layer neural network. They considered five sources of faults : input, weight, multiplier, adder and activation faults. Using a mathematical approach, they analyzed the effect of faults in a neuron and how they propagate across layers. [Ito1997] proposed a modification of the training process to improve the robustness of the model. These authors claimed that this approach requires fewer training iterations and provides better fault tolerance.

However : Although these studies constitute the cornerstone of the existing studies, the development of modern DNNs, new activation functions, normalization and pooling layers, and new hardware accelerators have triggered a renewed interest in the study of the fault tolerance of modern DNNs.

Bimodal Behavior of Floating-Point-based DNNs

Observation : Floating-point DNNs exhibit a bimodal behavior in the presence of faults : either the accuracy is negligibly impacted, or the accuracy drops to random guessing. Several articles have shown this behavior in the presence of memory faults (bit-flips in weights of DNNs). This section reviews the papers that have observed this phenomenon and presents an explanation for its cause.

Notably : This behavior is observed in multiple studies. [Arechiga2018] focused on the impact of SEUs in the weights of recent CNNs. VGG16, ResNet-50 and InceptionV3 are tested and for each network, the rate of faults causing a graceful degradation of accuracy is low. As shown in

Fig.2.4, faults cause either a catastrophic loss of accuracy or do not impact the accuracy. The fraction of runs that results in a graceful degradation (drop of accuracy between 10 and 70 percentage points) never exceeds 2%.

Furthermore : [Malekzadeh2021] performed an in-depth analysis of the effect of faults on weights on the simpler LeNet-5 CNN, while [Gao2020] injected faults on the weights, bias and batch normalization parameters on pruned networks and both observed this phenomena. [Li2019] also confirmed this observation when injecting timing errors induced by overclocking. In all these cited examples, with an increasing error rate, the accuracy suddenly drops to random guessing.

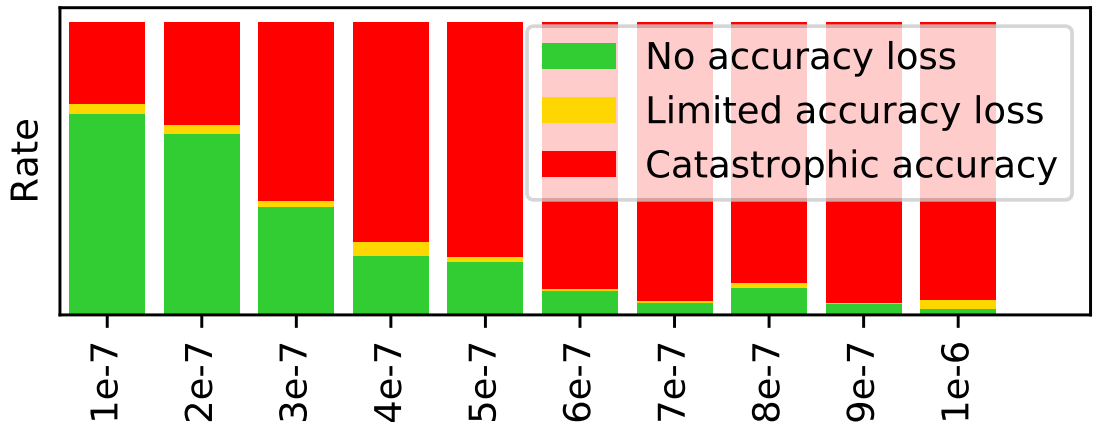


Figure 2.4: Bimodal Behavior on Floating Point based VGG-16
The stacked columns represent the percentage of runs that have a minor loss of accuracy (>80%), catastrophic accuracy (<10%), and a graceful degradation in accuracy (10 to 80%). The fault model is an increasing bit error rate in the weights. Fault free accuracy is 90%. 100 runs of 50K images were executed for each bit error rate. These data are taken from [Arechiga2018]. This publication has equivalent data for other topologies (Inception and ResNet-50).

However : Few papers have focused on the cause of this behavior. [Arechiga2018] compared the robustness of three CNNs. He concluded that the difference in robustness between them can be explained by batch normalization. This does not explain the bimodality. A better explanation is proposed in the study of [Malekzadeh2021]. This article provides insight into the comprehensive study of the robustness of CNNs to weight errors. While focusing on the simpler LeNet-5 network, their conclusion is that this observed behavior can be explained by the bit position affected by a fault. While the DNN appears to be robust to faults in LSBs, one single fault in the MSB of the exponent of a floating-point value can cause the accuracy of the DNN to collapse to random guessing.

To test the hypothesis of the criticality of MSBs causing the accuracy to drop to random guessing, we performed our own experiments and analyzed the results of [Arechiga2018]. Our results are summarized in Fig.2.5. There is a clear correlation between the probability of the MSB of a positive weight [2] being impacted and the rate of runs resulting in random-guessing accuracy. For example, for each of the tested networks, for approximately 10 bit-flips, both the rate of runs with random-guessing accuracy and the probability of a bit-flip altering the MSB of a positive weight is close to 20%. For approximately 45 bit-flips, the probability increases to 50% in both networks, while the rate of runs with random-guessing accuracy goes from 60 to 40%, depending on the network.

[2] We noticed that errors caused by a fault altering the MSB of a negative weight are likely to be mitigated by the widespread Rectified Linear Unit (ReLU) activation function, thus making them less critical.

Consequently, the hypothesis that a sudden loss of accuracy is triggered by a bit-flip in the MSB of the exponent to explain the bimodal criticality of floating-point-based DNN makes sense. Faults in the MSB of the exponent cause the accuracy of the DNN to collapse, while faults in the other bits appear robust to faults.

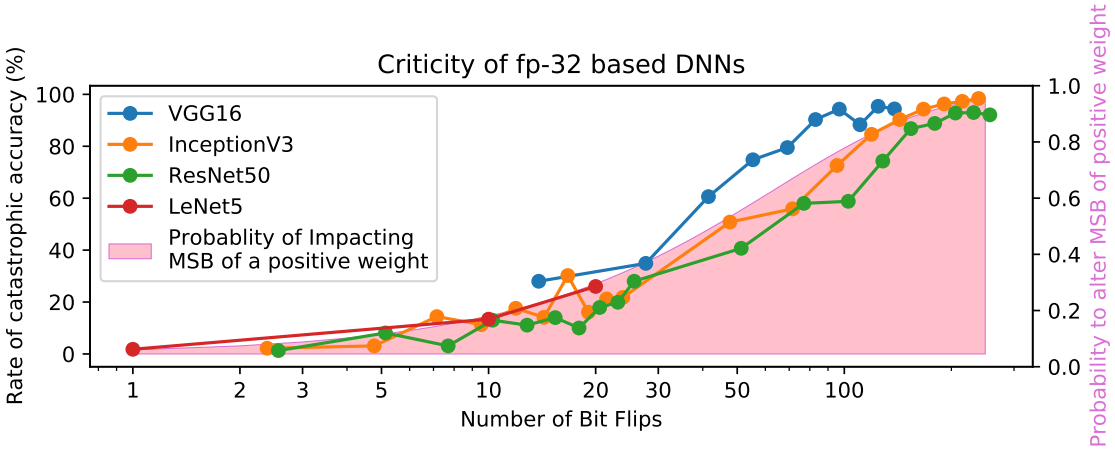


Figure 2.5: Behavior of Several DNNs at Increasing Number of Bit-Flips

The first four traces (blue, orange, green, red) are extracted from [Arechiga2018] and [Malekzadeh2021]. The pink curve is the calculated probability of the MSB of the exponent of a positive weight being hit; it is assumed that 50% of the sign bits are zero.

To conclude : The effect of faults on floating-point based DNNs is bimodal. The loss of accuracy is limited when faults impact LSBs. The accuracy is likely to collapse to random guessing when faults impact MSBs.

Neural networks calculated with other data types, however, appear to have an accuracy that gradually decreases in the presence of memory faults.

Role of Data Types

Observation : The numeric format used for weights and activations plays an important role in the robustness of DNNs. An error causing a large numerical deviation is likely to alter the behavior of a DNN [Li2017, Sabbagh2019]. Consequently, data types that reduce the probability of high numerical divergence are expected to be the most robust.

Notably : [Li2017] reports a difference in the robustness between different data-types that can differ by several orders of magnitudes, as seen in Tab.2.4. Networks using fixed-point values with a large integer bit-width are 9 times less robust than network using floating-point format. The tested fixed-point, a 32 bits fixed-point with 21 bits used to code integer has more of its bits that are likely to become SDC than 32 bits floating point when they are altered by an equivalent rate of transient faults. We note here that even if a floating-point format can cause a higher numerical divergence in the case of a bit-flip in the MSB of the exponent, they are more robust as they have a lower number of critical bits (number of bits that cause an SDC if they are altered by a fault).

However : Reducing the numbers of bits used to code the integer portion of the fixed-point value from 21 to 5 improves the robustness of the resulting DNN by 76×. The authors [Li2017] explain that the difference in robustness between data-types is explained by two factors, namely the number of critical bits, which is more important with 32b fixed-point (21b integer) and secondly by the numerical range, which is more important with 32b floating-point. As a consequence, data types that can represent an unnecessarily large numerical range should be avoided.

Data type	DNN	SDC Probability
32b floating-point	CaffeNet	0.46%
32b fixed-point (21b integer)	CaffeNet	7.01%
32b fixed-point (5b integer)	CaffeNet	0.0875%

Table 2.4: SDC Probability Based on Data Type
Results extracted from [Li2017]. SDC Probability for a single bit-flip in activations.

This problem is addressed by quantization [3]. The robustness of quantized DNNs has been extensively studied. [Sabbagh2019] shows that quantization can improve the robustness of LeNet-5 by three orders of magnitude and VGG-16 by one order of magnitude. [Syed2021] explores different quantization bit-widths to conclude that reducing the bit-width from 32 bits to 4 bits can improve the robustness by 20.7%. Quantization improves the robustness of DNNs, and the usage of low-bit-width quantized DNNs is preferred for fault tolerance.

Extreme quantization through binarization further improves the robustness of quantized DNNs. [Sabbagh2019] claims that binarization can improve the robustness of LeNet-5 and VGG-16 by 4 orders of magnitudes compared to their floating-point baseline. Furthermore [Gambardella2019] confirms that binarized DNNs record a worst case loss of accuracy of 10% with the unusual channel-error fault model (all bits of a given channel are stuck-at a faulty value). These studies confirm that Binarized Neural Networks are several of magnitude more robust than their floating-point-based counterparts. These theoretical observations regarding binarized DNNs have been confirmed with experimental tests on FPGAs by [Souvatzoglou2021, Libano2020]. We note however that binarization process significantly reduced the accuracy of a DNN, or requires the addition of neurons to maintain an equivalent accuracy compared to a larger numeric format.

To conclude : The data-format of a DNN plays a key role in its fault tolerance. DNNs based on numeric formats with a small numeric range are less sensitive to faults.

Impact of Pruning

Observation : Pruning [4] is another model compression technique for DNNs that has a minor impact on the robustness of DNNs.

Notably : Some authors have performed fault-injection studies on pruned networks. [Sabbagh2019] performed experiments on pruned versions of LeNet-5 and VGG16. The robustness of both networks is improved by one order of magnitude by pruning as seen in Fig.2.6.

However : With a pruning rate of nearly 90%, the previously cited paper also reduces the memory footprint of the DNNs by one order of magnitude. Consequently, pruning does not improve the robustness of the DNNs for a given absolute number of faults. Instead, pruning improves the robustness by reducing the probability of a fault to occur. To confirm this idea, [Gao2020] studied the effect of different pruning rates for VGG16, using multiple fault models (bit-flips in weights, bias, and batch normalization parameters). This study shows that the robustness of DNNs is hardly altered by the pruning rate, as the accuracy varies by +/-3% for 3 weight bit-flip regardless of the pruning rate, while the memory usage of the model is reduced with the higher rate of pruning.

[3] As seen in Sec.2.1, quantization is a process that converts both weights and/or activations of a DNN to a discrete space through the use of integers instead of floating-point computations. It allows a reduction in bit-width, which results in smaller model size and improves the energy efficiency of computations [Sze2017].

[4] As seen in Sec.2.1, pruning is a process that removes neurons and weights from the computation with a minimal degradation of the final accuracy. It allows a reduction in the compute and memory usage of a DNN. There exist several pruning methodologies that often rely on three steps : 1) The training of a baseline DNN, 2) The removal of nodes and/or weights of the network, and 3) A re-training for reducing the accuracy loss caused by step 2.

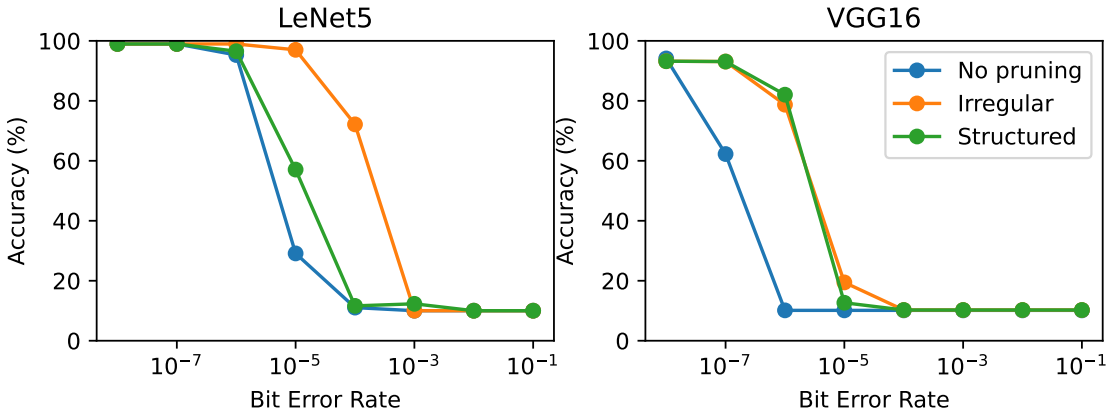


Figure 2.6: Accuracy of Various Pruned DNNs at Increasing Bit Error Rate
The data are extracted from [Sabbagh2019].

To conclude : Pruning reduces the computational and/or memory usage usage of DNNs. While pruning does not impact the per-bit sensitivity to faults, the memory reduction caused by pruning improves absolute sensitivity, as the memory footprint is reduced.

Layer-Level Analysis

Observation : The robustness of the layers of a DNN can vary.

Notably : [Kwon2016] studied the impact of faults in weights in different layers of LeNet-5. In this study, the results show that the first layer is three times more robust than the last convolutional one. [Li2017] studied the robustness of different layers in three different DNNs (AlexNet, CaffeNet and NiN). The output layers of the first two DNNs are more sensitive than the earlier ones (the last layer has a SDC rate that is twice as high as the third layer). [Xu2019] performed similar experiments on AlexNet, VGG16 and VGG19, and these results are represented in Fig.2.7. We see that for each tested DNN, the last fully-connected layer is the most sensitive one. [Wan2021] also performs a similar analysis using a custom network comparable to LeNet-5, and reaches similar conclusions.

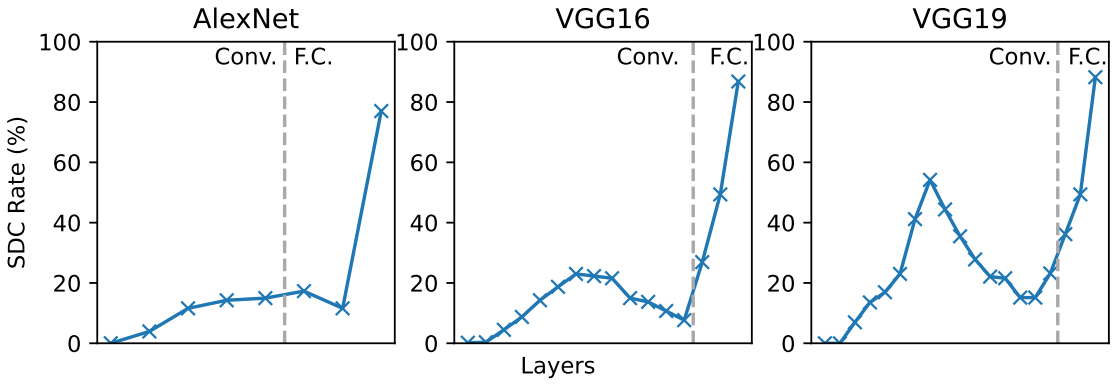


Figure 2.7: Probability of SDCs of various Layers of Three DNNs at Constant Error Rate
These data are extracted from [Xu2019].

However : Much work remains to be done to understand the robustness of individual layers. In [Kwon2016], the distribution of the values of the weights is proposed to explain this difference, as the first layer has a range that is 20x larger than the third layer, being three times more robust (only two layers are considered in this study). In [Li2017], the distribution of the activations are considered as a possible explanation. As we will see in chapter 3, this explanation does not consider the topology of DNNs. Modern networks use BN, which results in normalized weights and activations among all layers, while the difference of robustness between layers remain. Therefore, there must be another explanation for understanding the robustness of layer.

To conclude : The first layers are generally more robust than the last ones. Typically, the last fully-connected layer is likely to be the most sensitive in a DNN. However, much works remain to be done to understand all the factors that contribute to the sensitivity of different layers.

Inherent Robustness of DNNs

Observation : Certain architectural characteristics of DNNs such as activation functions, pooling layers and normalization layers contribute to the overall fault tolerance of the whole model. This section explores the role played by these mechanisms.

Activation functions are able to mitigate the effect of certain faults. [Li2017] cited the error mitigation of certain activation functions but did not quantitatively analyze them. As illustrated in Fig.2.8, the ReLU activation function is able to mask faults producing negative values of an activation without changing its sign, as any negative value is set to 0. [Malekzadeh2021] compares the robustness of two activations functions : ReLU and Sigmoid and concludes that the latter is more robust. The positive output of ReLU is not bounded. The Sigmoid function, on the other hand, is bounded between 0 and 1 and thus is expected to mask more faults. This expectation is confirmed at the cost of a reduced fault-free accuracy. The results show that with comparable error rate, the Sigmoid prevents the bimodal problem seen in Sec.2.2.1. However, real world applications may not benefit from these findings, as the ReLU activation function performs significantly better in many applications (including this study) and is computationally much simpler [Datta2020]. Activation functions are able to mask some errors. The ReLU function which is currently one of the most widely used in modern DNNs can mask errors resulting in negative values.

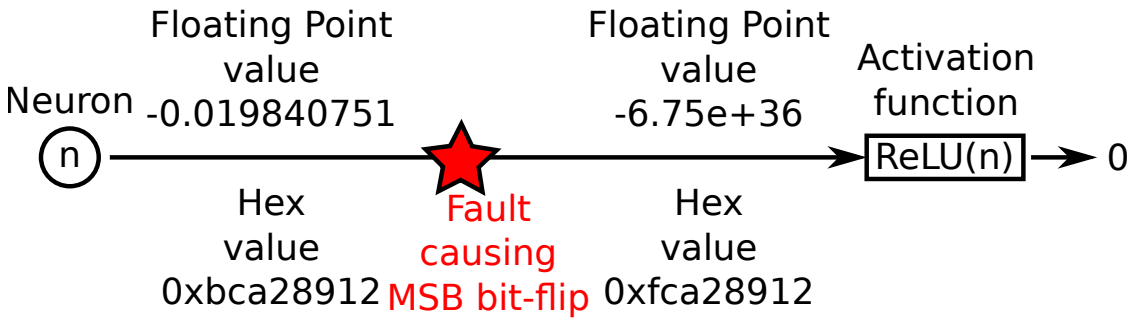


Figure 2.8: Example of ReLU Masking an Error
ReLU is an activation function defined as $ReLU(x) = max(0,x)$ This activation function is used in state of the art DNNs trained for computer vision.

Pooling layers play a minor role in robustness. [Li2017] discusses the error-masking ability of pooling layers without providing quantitative analysis. [Malekzadeh2021] analyzes the error masking ability of max and average pooling layers. The results of this study show that compared to max pooling, the usage of average pooling layer reduces the number of catastrophic errors by approximately 60% but does not mitigate all of them. Consequently, pooling layers are theoretically able to mitigate certain faults, but are not effective for masking all of them.

LRN is a layer that normalizes several neighboring neurons. [Li2017] compares three DNNs (AlexNet, CaffeNet and NiN) and observes that the networks which uses LRN are more robust than those which do not. Without a quantitative analysis of comparable networks, the authors conclude that normalization layers play a role in the robustness. We think that such conclusions are premature, since the three networks are fundamentally different (e.g. NiN is a fully convolutional DNN, while AlexNet and CaffeNet are more traditional CNNs with a mix of convolutional and fully-connected layers). In fact, this article only considers LRN, which is rarely used in modern networks. The authors do not consider BN, which is currently accepted as the best normalization technique [Moradi2020].

BN is studied by [Arechiga2018]. This author considers three different networks (VGG16, ResNet-50 and Inception) and observes an order of magnitude difference in robustness between these DNNs. They conclude that this is caused by the normalization layers. VGG16 which does not use normalization is ten times more sensitive to faults than the others networks. Based on our own experiments, we question the conclusion of the authors. The three tested DNNs are fundamentally different, and a better explanation is proposed by [Mittal2020], who notes that these DNNs are equivalently robust to the same absolute number of errors. In addition, during inference, BN performs a simple linear transformation, consequently may be fused with the previous layer without any impact on the computation [Ioffe2015]. The ability of BN to mask errors therefore is questionable, since there is no real normalization of the inputs during inference.

Experiments were performed to further analyze the effect of BN on the robustness of a DNN. The results of our experiments are shown in Fig.2.9. We recorded the average accuracy on 100 runs for increasing bit error rates on VGG-16, with and without BN. Considering the same fault model (bit-flips in weights) as [Arechiga2018] we see that the usage of BN slightly reduces the robustness. With the activation fault model however, BN appears to slightly improve the robustness. However, in all cases the effect of BN on robustness is minor. While previous studies claimed that BN improves the robustness of DNNs, we think that when analyzed carefully, the effect of BN on robustness is negligible.

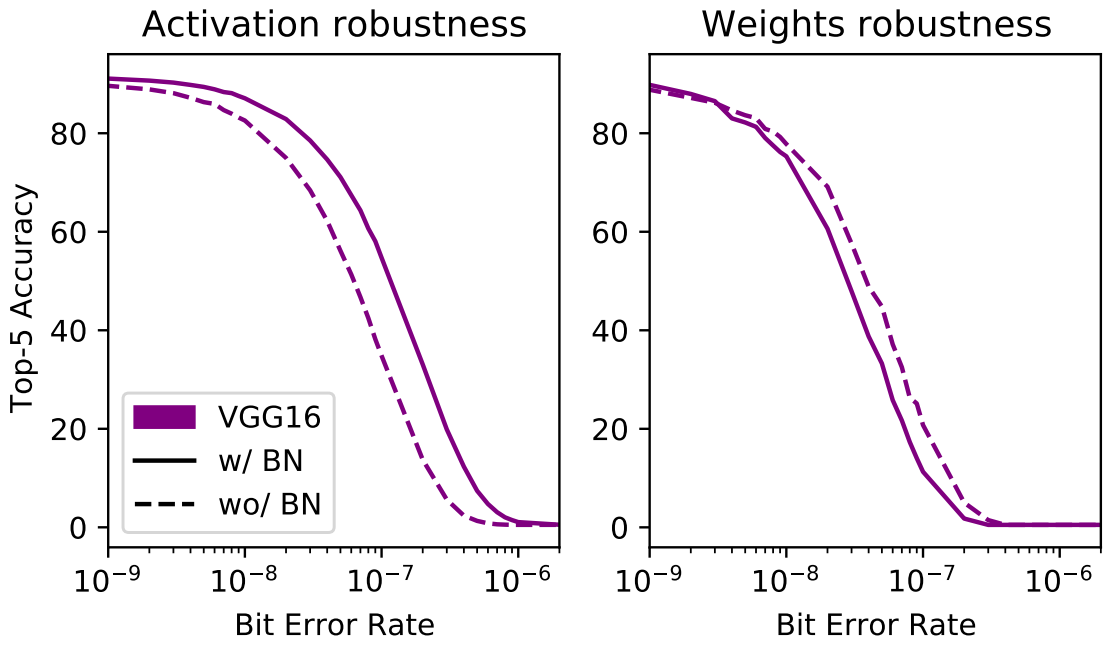


Figure 2.9: Impact of Batch Normalization on Robustness

To conclude : Various components of DNNs can individually mitigate faults. Both activation functions, normalization layers and pooling layers can theoretically reduce the impact of faults. However, in practice, these observations can not easily be applied to modern DNNs. The widely used ReLU activation function is less efficient at mitigating faults than the historical Sigmoid function. Equivalently, the currently globally accepted BN is less efficient to mitigate faults than the LRN, which is not used anymore in modern DNNs. The ability of pooling layers to mask errors which provoke a large, erroneous values, is limited.

Sensitivity of Weights and Activations

Observation : For abstract DNNs, the weights and activations are frequently used as the target of faults injection experiments and studies.

Notably : Being stored in memories, weights typically represent an important fraction of the memory footprint of a DNN, as seen in Sec.2.1.1. Thus a fault occurring in the memory of a DNN is likely to impact its weights. Consequently, studying faults in the weights is a way to evaluate the impact of faults in memory. Equivalently, the activations are the computational result of a weighted sum between activations from the previous layer (or input) and weights. Thus a fault occurring in the computational units executing a DNN is likely to impact the activations. Thus, faults occurring in the activations can be seen as a proxy for faults in the computational data-path of a DNN.

However : While many studies separately consider these two fault models, there are few works that extensively compare the robustness of both, and they all report that faults occurring in activations are less likely to alter the behavior of the application than faults occurring in weights. [Reagen2018] tested the robustness of several models (LeNetFC, LeNetCNN, CIFAR10, VGG16, ResNet50 and Tigru) to faults occurring in activations and weights. In every tested cases, the robustness of activations is better than that of the weights, for equivalent BER. On bigger networks (ResNet50 and VGG16), activations are more than 50x more robust to faults than weights. [Neggaz2018], using an equivalent methodology, reaches the same conclusions. This work focused on LeNet-5, but the results show that faults injected in activations have a smaller impact as faults injected in weights. To confirm the results of these studies, we performed our own experiments to assess the robustness of both these components and our results are summarized in Fig.2.10.

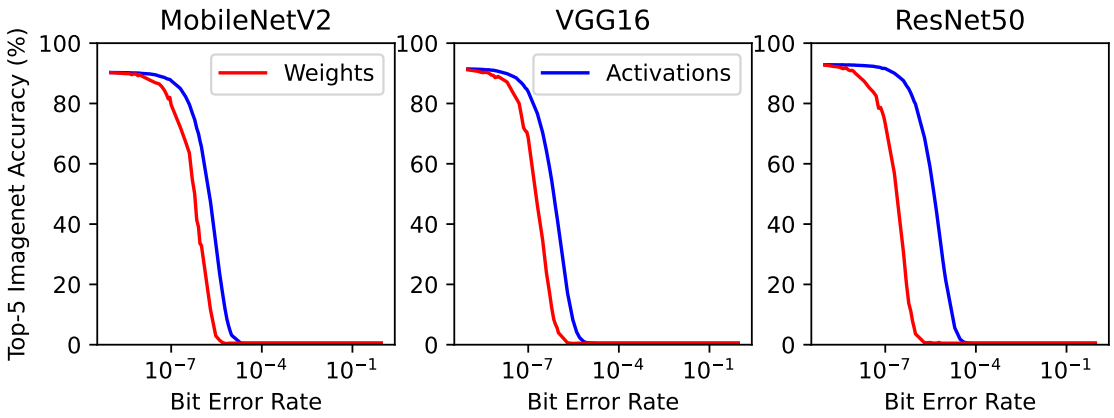


Figure 2.10: Comparison of the Robustness of Weights and Activations
These data are the results of our own experiments. To reduce the bimodal criticality of DNNs seen in Sec.2.2.1 , we used the ReLU6, a clipped version of ReLU. $ReLU6(x) = min(6, max(0, x))$ Equivalent experiments on ReLU were performed, with comparable results. We used the same fault injection methodology as [Reagen2018] but using floating-point DNNs.

We applied an equivalent bit error rate in both weights and activations for MobileNetV2, VGG16 and ResNet50. For each network, at equivalent bit error rate, the accuracy is more impacted by faults occurring in weights than faults in activations. This can be explained by the fact that a fault occurring in a weight will spread to a whole channel, while a fault occurring in an activation only leads to a limited local propagation. Thus, our experiments confirm the conclusions of the existing works.

To conclude : Weights and activations are two widely used fault targets that exhibit different levels of robustness. Intuitively, faults in weights can be viewed as a proxy for faults in memory and faults in activation as a proxy for computational faults. The few articles that quantitatively compare their robustness conclude that activations are more robust than weights and our own experiments confirm this observation.

Estimation of the Robustness of DNNs

Observation : A portion of the existing works focuses on estimating the robustness of an abstract DNN without performing time consuming fault injection campaigns.

Notably : DNNs are complex and the simulation of their hardware accelerations is time consuming. Consequently, light-weight techniques to estimate the robustness of a DNN are of scientific interest. For example, [Xiang2019] proposed a statistical analysis of the weights in a DNN and their importance to the output decision of the network. With the sensitivity of each weight, the computation of the robustness of the whole application can be deduced. This theoretical approach relies on the assumption that the input values to the DNN follow a normal distribution.

The study of human generated faults with the intent of causing a DNN to produce an incorrect result, known as adversarial attacks, is vast. Although this is not the topic of this thesis, certain ideas from this field of study may be applicable to DNN hardware fault tolerance. [Webb2018, Yu2019] both propose new metrics to statistically estimate the robustness of a DNN to adversarial noise at the inputs. [Couellan2021] proposes a probabilistic estimation of the robustness of the output of a DNN to local variations in its inputs, using a process similar to gradient descent.

However : The cited publications make assumptions which may limit their applicability in real-world applications. The statistical-based estimation proposed by [Xiang2019] is limited to weight perturbations (noise) as a fault model. Furthermore, the assumption that the inputs of a DNN follow a normal distribution may not always be true. The adversarial attacks studied by [Webb2018, Yu2019, Couellan2021] do not exhaustively cover the fault models studied in the works that consider hardware faults.

To conclude : Techniques to estimate the robustness of a DNN and avoid costly fault injection campaigns are of great importance. However, much work remains to be able to estimate the robustness estimation of a DNN to hardware faults. The existing studies focus on statistical or probabilistic estimation of the robustness of DNNs and rely on assumptions (e.g. normal distribution of the inputs) or theoretical fault models (noise) and more work is required to study whether these results can be applied to the analysis of hardware fault tolerance.

Summary of the Robustness of Abstract DNNs

This section concludes the analysis of the State Of The Art that considers abstract DNN, and summarizes the conclusions.

Critical faults reducing the accuracy to random guessing is a problem in floating-point based DNNs. We have seen that on floating-point-based DNNs, faults have virtually no impact on the DNNs output until a certain threshold is crossed, after which the behavior of the DNNs is reduced to random-guessing. There even exist rare cases where one single bit-flip in such DNNs results in complete failure.

Fault Tolerant DNN Mechanisms can contribute to mask faults or mitigate errors. Activations functions may clip high variations caused by a fault, and the usage of average pooling layers

over max pooling improves the robustness of DNNs. These components are not specifically designed to mask faults, and many of these mechanisms are no longer used in modern DNNs, such as average pooling or Sigmoid activation functions.

Data types play a key role in the robustness of DNNs as globally explained in existing works. When the data type can limit the numerical range available within a DNN, it reduces the worst-case numerical divergence caused by a fault. Binarized DNN thus appears to be the most robust.

Pruning plays a role in the fault tolerance of DNNs and can further improve the robustness, by reducing the computational and memory usage of a DNN.

The layers of a DNN are not equally robust. It is generally accepted that the final, fully-connected layers are more sensitives than the first layers, typically convolutional layers used for feature extraction

Weights and activations are two common fault targets as they serve as simple proxies for memory and computational faults, respectively. Errors altering weights are more likely to degrade the accuracy of a DNN than activation faults. Weights faults propagate through a whole channel, whereas an activation fault propagates only to downstream neurons.

The estimation of the robustness of DNNs using fast techniques is of interests as fault-injection campaigns are computationally expensive. Approximative techniques based on statistics rely on assumptions and more work is required to see whether they can be applied to the analysis of hardware fault tolerance.

Conclusions drawn from the study of abstract DNN provide useful insight in how to improve the robustness of DNNs, however abstract DNNs remain an abstraction and do not always reflect the behavior of faults in actual hardware accelerators.

2.2.2 Robustness of DNNs Accelerated on Hardware Platforms

This section presents the existing works that focus on the robustness of DNNs considering hardware accelerators specificities. For exhaustively estimating the hardware fault tolerance of a DNN, architectural and technological aspect must be taken into consideration. As a consequence, an important part of these studies consider the robustness of DNNs when they are executed by hardware accelerators. This section presents works that analyze the robustness of DNNs when they are executed on the most used hardware accelerators and is structured as follows :

- **Sec.2.2.2** presents the robustness of DNNs executed on GPUs.
- **Sec.2.2.2** considers the power consumption reduction that tolerance of DNNs could allow in case of voltage-down-scaling of SRAM Cells.
- **Sec.2.2.2** presents the robustness of DNNs executed on FPGAs.
- **Sec.2.2.2** concludes this section.

Robustness of GPUs

Observation : The robustness of GPUs as DNNs accelerators is an active field of research. As GPUs are commonly used accelerators for DNNs due to their high computing powers and their accessibility [Strigl2010], study on their robustness are of scientific interest. GPUs are becoming increasingly subjects to hardware faults due to the shrink of transistors size [Fratin2018] and the emergence of new computing paradigm for reducing energy consumption [Ganapathy2019]. Noticeably, the fault tolerance of GPUs is assessed with software based fault injections frameworks.

Notably : NVIDIA has recently published SASSIFI/NVBitFI, a fault-injection framework for their GPUs [Hari2017, Tsai2021]. SASSIFI is a low-level assembly-language tool to perform fault injection during run-time of a GPU application. Faults can be performed on several types of instructions that operate on value and address and alter their outputs. This frameworks is used by

many studies as [Adam2021b], which performs fault injection on AlexNet. The results are presented in Fig.2.11. Authors conclude that the tested model is more subject to SDC than DUE. All SASSIFI instructions are tested, and the Floating-points Addition (FADD) and Load (LD) instructions are by far the most vulnerable to bit-flips. They result in SDC for at least 84% of injected faults. The instructions STORE, Set Register value (SETP), Floating-point Fused Multiply-Add instructions (FFMA) are also vulnerable at a lower degree. Condition-Code registers (CC) instructions, on the other hand, appears to be resilient to errors.

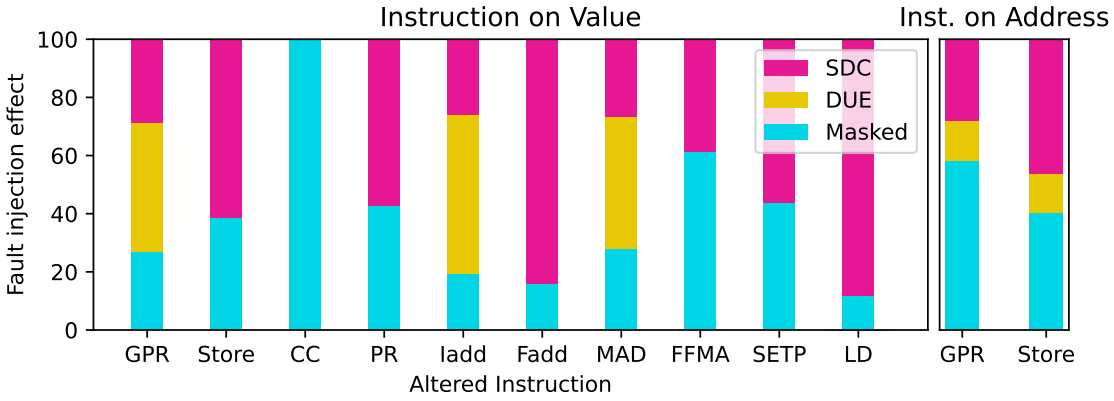


Figure 2.11: Criticality of Different Instructions of AlexNet on a GPU
The impact of fault on each tested instruction is showed. Fault injection executed with SASSIFI framework, on NVIDIA GeForce GTX 750 Ti. Instructions are split in the Instruction operating an Address on the one side, and Instructions operating other values on the other. For each instruction, rates of fault injection with masked effects, SDC and DUE are reported. Data extracted from [Adam2021b].

However : Some studies are cautious about the NVIDIA framework, considering that these software-based fault injections may not efficiently simulate the effect of real faults. [Ito2021] supposes that the control flow (order of instructions) is a weakness of the GPU-based applications, but considers that the CC control instructions of SASSIFI is not representative of the control flow of an applications. Instead, authors perform fault injection on the program counter to disturb the control of the studied DNN. With one single bit-flip, experiments show that the instructions used for loading arguments, configuring threads and blocks leads to DUE in 53.4% of cases. Instruction that write data to memory leads to DUE in 66.9% of cases. Arithmetic-related instructions leads to 3.4% of DUE. Authors thus conclude that the arithmetic-related instructions are the most robust. [dos Santos2021] compares the software-based framework with beam experiments, and concludes that the fault injection framework of NVIDIA successfully estimate the rate of SDC. DUE rate, however, is from 60X (ECC-on) to 46.700X (ECC-off) underestimated by software-based faults injections, compared to beam experiments measurements. Authors explain that components unreachable by the programmer seem responsible for an important rate of DUE. [Ito2021] reach the same conclusions. In both of these studies, an important rate of DUEs remains caused by unknown agents, which could be explained by the fact that hardware components inaccessible to the programmers (thus to the fault injection framework) are sensitive and lead to an important number of DUEs.

Furthermore : The role of the resource utilization of GPUs on fault tolerance is considered by [Badia2021]. The robustness of three different scheduling strategies to radiation-based experiments are compared. The main conclusion of this work is that most of the errors result in DUE, and they can be reduced by lowering the application latency. In others words, the fastest algorithm is the strongest, even if it uses intensively the cores of the GPU.

To conclude : As the robustness of GPUs for DNNs is an active field of study, NVIDIA has published tools that allow to perform software-based fault injections on applications using their GPUs. This framework highlight the criticality of instructions related to data-transfer, while arithmetic operations are robust. Such conclusions are confirmed by other fault injections executed on the program counter of GPUs. These software-based fault injections successfully simulate the rate of SDC compared to physical measurements, but fail to efficiently simulate the rate of DUE. This is caused by the fact that components unreachable by programmers may cause an important number of undetected DUEs. The resource utilization of GPU also plays a role on the robustness. The fastest algorithm is the strongest, and full usage of resources of GPUs is advised by the cited works for improving the robustness.

Robustness of SRAM Cells Used in DNN Accelerators

Observation : SRAM is a type of random-access memory (RAM) that uses latching like circuitry to store each bit. SRAM cells are being a growing concern for the energy consumption of modern applications [Kumar2021]. This has triggered the focus of researchers and many strategies have been found to reduce the power consumption of SRAM [5] at a cost of a reduced reliability. This section presents the existing works that focus on the energy-consumption vs robustness trade-off of SRAM cells.

Notably : The power consumption is reduced by lowering the voltage supply used by cells. [Yang2017] proposes quantitative experiments on a real world 28nm CMOS 8KB SRAM test chip. On the tested chip (28nm FD-SOI), authors show that supply voltage can efficiently be reduced to a range from 0.8V to (0.49-0.52V) with limited impact on the classification accuracy of a tested network. This lead to leakage savings of 4.8x to 5.4x and memory access power of 2.6x to 2.9x. Other authors explore the fault tolerance of DNNs to reduce their power consumption. [Wu2021a] use existing data in the scientific literature to use a targeted voltage (0.5 V), that induces an important rate of bit-cell failure rate (0.1%) and make a DNN adapted to it. They experiment a low-bit-width, floating-point data type. They show that this 7-to-8 bit floating point drives to a robust DNN up to important bit-cell failure rate (0.1%). The usage of low-bit-width floating point and the strategy of developing a DNN computational paradigm designed for robustness is of scientific interest.

However : The conclusions of previously cited studies should be considered with prudence. In [Wu2021a], the fault model used (bit stuck-at) may not be representative of read-error faults in SRAM caused by low voltage, and all this work rely on assumptions taken from data from another study (namely 6T SRAM Cell on 65nm technology in [Wilkerson2008]) that may be not adapted to modern accelerators. In [Yang2017], the considered case-study is a two-layers DNN, which is not representative of recent applications. This issue is addressed by [Wu2021b], which use a more recent case study (VGG-16). Authors also simply reduces the reduction of the supply voltage of SRAM cells to reduce the power consumption. Beyond a tolerance threshold, this methods induces read-errors, which are simulated as bit-flips. As we have seen in Sec.2.2.1 and Sec.2.2.1, it can have severe effects on floating-point-based DNNs, but can be more efficiently endured by quantized DNNs. To tackle this issue, authors use quantized DNNs and simulate errors in the weights of a quantized neural network. They conclude that quantized DNNs can tolerate an important number of bit error rate, and thus that the power-supply lowering can reduce the consumption of both dynamic and leakage power by 30% without effects on the classification accuracy of the network. These results are illustrated in Fig.2.12. The robustness of quantized DNNs can be used to reduce the power consumption of DNN without great efforts.

[5] The development of new technologies and new SRAM Cells is beyond the scope of this thesis, but the reader is referred to [Rao2022, Kim2021, Abbasi2022] to learn more about emerging SRAM technologies.

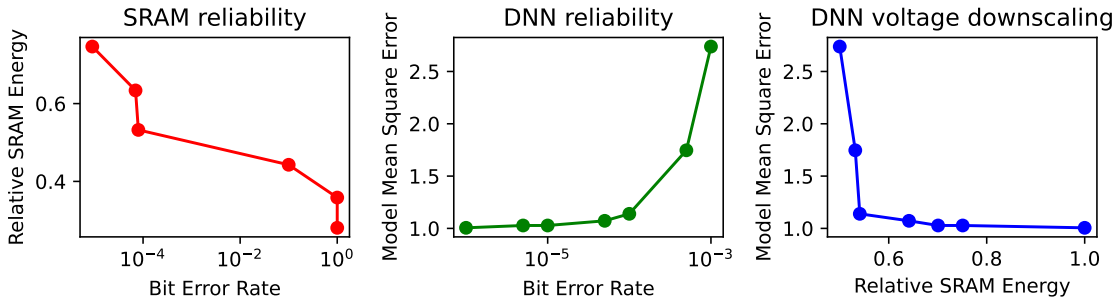


Figure 2.12: Example of SRAM Cell Voltage Down-Scaling for DNNs
The intrinsic robustness of quantized DNNs can be used to reduce the power consumption of an application without great efforts. Voltage can be reduced by 30% without any effect on accuracy, and 45% with a minimal loss on accuracy. Model is a ResNet50 trained for CIFAR10 and quantized to 8 bit computations. The data are extracted from [Wu2021b].

To conclude : Due to the large role played by SRAM in energy consumption of modern SoC, several papers consider the lowering of the power supply as an efficient power-reduction strategy. It induces a raise of bit error rate, which makes the floating-point data unadapted due to their bimodal behavior in presence of faults, as seen in Sec.2.2.1. Existing works consider the quantized DNNs, and show that a power reduction can lead to significantly improvement of energy efficiency without significant accuracy loss.

Robustness of FPGA-based DNN accelerators

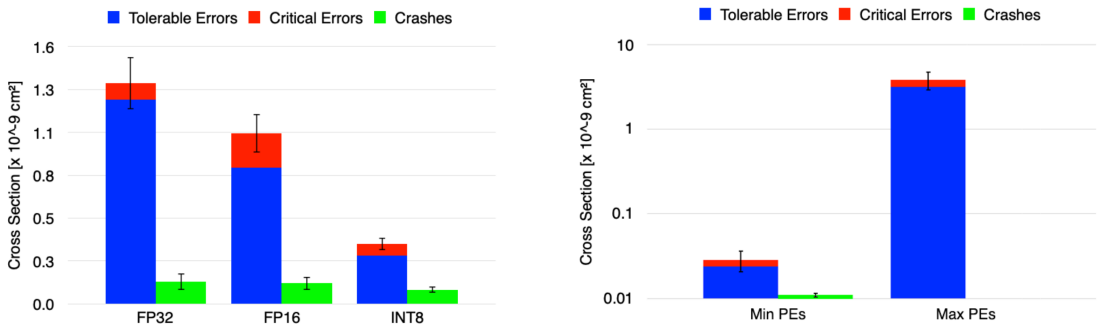
Observation : FPGAs offer an important hardware reprogrammability, and a better power consumption than GPUs [Reuther2021], which makes them a target of choice for the design of DNNs accelerators. As a drawback, it has been proven that FPGAs may be sensitive to radiation and other errors, depending on the mechanism they use to store the configuration data [Wirthlin2015].

Notably : Configuration units of FPGAs are known to be a weak spot of FPGA robustness. As seen in Sec.2.2.1, binarized DNNs, which have been proven to be resilient to arithmetic and weight memory errors. For example, [Sabbagh2019] report that binarized DNNs are robust to up to 10% accumulated faulty bits in their weight. In [Souvatzoglou2021], authors consider reliability issues that are specifically induced by Configuration Static Random Access Memory (CSRAM) on FPGA accelerating binarized DNN. Authors perform a vulnerability analysis considering configuration memory of FPGA, user flip-flops and registers. This work concludes that single bit-flip fault altering CSRAM of such accelerator leads to critical errors with a probability of 0.3% and crashes with 1.2%. Single bit fault in user flip-flops leads to critical errors with a probability of 0.12% and crashes with 2.25%. The gap of these results with the robustness of binarized DNNs to computation faults reported by other studies can be explained by the fact that the configuration memory of FPGAs remains a weak spot of such architectures for the acceleration of DNNs.

However : The FPGA remains a target of choice for studying the fault robustness of DNN accelerators, as faults can be easily injected in the control units of executed DNN accelerators. [Xu2021] assess the global robustness of a DNN accelerator, including faults in Direct Memory Access modules, control unit (that control the execution of the accelerator), instruction memory and their related exceptions. DNN accelerators are less robust than globally expected when the fault tolerance of control is taken into considerations. The first components that force the system to stop considering an increasing bit error rate are the control units, while equivalent level of faults only slightly reduce the accuracy in the computational units of the DNN. This work concludes that control units should be considered to compute the robustness of an accelerator. Fortunately, the

fault tolerance of such units have already been extensively studied [6]. Control units play a major role in the robustness of a DNN. They seems to be a weak spot of DNNs but can be improved with methods that are out of the scope of this thesis.

Furthermore : The role of resource usage on the robustness of FPGAs was studied by [Libano2021]. Authors use a CNN trained on the MNIST dataset and test the robustness of different data types and parallelism strategies to fault injected by neutron beam experiments. The results are reported in Fig.2.13, that represents the rate of critical faults (altering the output of DNN), crashes and tolerable faults depending on the resources usages (rate of used configurations bits) and used data-types. The reduction of critical errors depends on the resources usages of data-types. Compared to FP32, FP16 uses 40% less resources and has 22% less errors. INT8 uses 64% less resources and reduce 72% of errors. The parallelism strategies also impact the robustness. Two strategies are used : Maximal and minimal resources usage. Maximal uses 130x more resources and is 133x more likely to be affected by errors. While resource utilization directly impact the robustness of the model, the latency also plays a major role as well. Considering that the maximal resource usage is 1000x faster, authors conclude that the reliability of a DNN depends more on its latency than the amount of resource it uses. Given the data types, the conclusion is that quantized DNNs are more robust that floating-point-based implementation. However, we notice that the quantized DNNs are only 6x more robust than other tested data type, whereas many others studies conclude that this difference of robustness should be measured in several orders of magnitude, as seen in Sec.2.2.1.



(a) Neutron Cross Section for Different Data-Types

(b) Neutron Cross Section for Different Usage of Resources of FPGA

Figure 2.13: Results of Neutron Beam Experiments on a FPGA
Results extracted from [Libano2021] from experiments on 28nm Zynq7000 (XC7Z020) running a LeNet-5-like network. Outputs altered by faults are classified as errors. Critical errors change the classification results, while tolerable does not. Fig.2.13a compare data types. Fig.2.13b plots the neutron cross section of different levels of parallelism. Min PEs is the minimal version of FPGA acceleration and use a limited number of resources. Max PEs uses every possible resources of the FPGA. The latency of the tested DNN is not considered in these figures.

To conclude : FPGAs, which offer unique opportunity as DNN accelerators, suffers from their complexity. The configuration memory of the FPGAs appears to be a weak spot, and experiments have proven that DNN accuracy is subject to be altered by faults in such component before being altered by faults in the weight memory, or the computational units of DNNs. The studies considering FPGAs confirm conclusions reached by works related to the robustness of GPUs. The implementation of the DNNs that has the lowest latency and the lowest bit-width quantized data-type is the most robust.

[6] This is out of the scope of this thesis, and reader is referred to [Mukherjee2005b] to further study the architectural fault tolerance of modern computing architectures.

Conclusions on the robustness of hardware DNN accelerators

Observation : Considering hardware specificities is necessary to precisely assess the robustness of the execution of a DNN.

Notably : The studies that considers only abstract DNNs generally does not considers faults occurring in control units or configuration of DNN accelerators. Yet theses are a weak spots of DNN accelerators, as reported by studies that considers faults occurring on GPUs and FPGAs.

However : Fault tolerance of DNN hardware accelerators benefits from discoveries in the studies that considers abstract DNNs. For example studies consider to benefit the fault robustness of quantized DNN to reduce the voltage of SRAM Cells of DNN hardware accelerators below the minimum threshold. This dramatically reduces the power consumption of the accelerator, but induce errors that robust DNN can absorb in a certain proportion.

To conclude : When deploying a DNN to a safety critical application, the overall hardware DNN accelerator must be taken into consideration, which induces that architectural and technological specific weak points such as configuration and control units must be taken into consideration.

2.3 Improving the robustness of DNNs

This sections presents works from the existing studies that focus on improving of the robustness of DNNs. This section is structured as follows :

- **Sec.2.3.1** presents techniques used for detection of faults in DNNs.
  - **Arithmetic-Based Fault Detection** presents code-based arithmetic techniques.
  - **Symptom-Based Error Detection** presents techniques that monitor the DNN for abnormal behavior.
  - **Error Detection During Training** presents fault detection techniques that can be applied during the training process.
  - **Hardware-based Fault Detection** presents hardware-oriented techniques.
  - **Conclusions on Fault Detection Techniques** concludes this section.
- **Sec.2.3.2** presents techniques used to mitigate faults occurring in DNNs.
  - **Zero Masking Technique** presents articles that mask erroneous value with zeros.
  - **Numerical Range Restriction** presents articles that limit the range of all the values to minimize the impact of erroneous values.
  - **Fault Tolerant Training** presents techniques that modify the training procedure to make the DNN more tolerant to faults during inference.
  - **Selective Modular Redundancy** presents techniques based on the temporal or spatial redundancy.
  - **Fault Tolerant Scheduling** presents techniques that remove faulty components from the execution of a DNN.
  - **Conclusions on Fault Mitigation Techniques** concludes this section.

2.3.1 Fault Detection Techniques for DNNs

As fault detection plays a key role in fault tolerance, in the existing works, we find many fault detection techniques which focus or have been adapted on DNNs. This section presents the most well-known fault detection techniques for DNNs

Arithmetic-Based Fault Detection

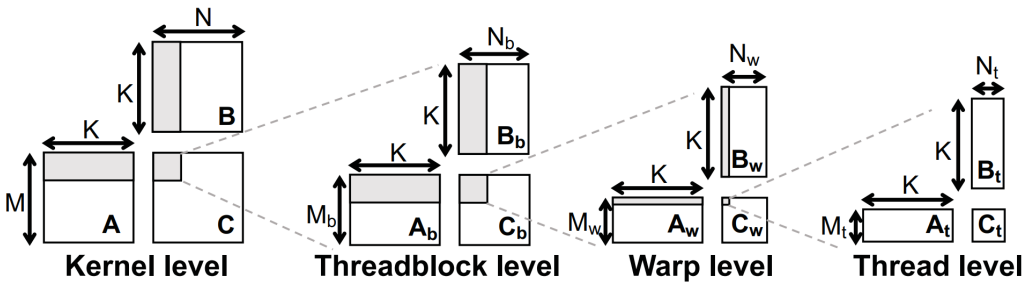
Observation : AN codes and Algorithm-Based Fault Tolerance (ABFT) are two methods used in arithmetic operations to detect faults. The AN codes adds redundancy in the data (typically by multiplying an integer with a prime number and by adding a constant) to detect faults occurring during arithmetic applications [Brown1960]. ABFT is adapted to matrix multiplications. It adds an extra column or row to matrices in order to create a check-sum that can be verified in the product [Huang1984]. The computational and storage overhead for arithmetic techniques is lower than full duplication.

Notably : A subset of the AN codes is explored in the studies on fault tolerance of DNNs. [Goldstein2021] applied a 5-bit arithmetic AN codes to the MAC units of conventional DNN accelerators, and tested the resulting strategy on several convolutional networks. Authors test two different prime codes, 7 and 31, and this technique can detect over 99.5% of faults with a BER of up to 10e-2, since the detection fails if the fault modifies the MAC result such that it becomes a multiple of the code (unlikely to happen with prime numbers). [Fu2021] use Residue Number System AN codes to perform error detection during arithmetic operations. Residue Number System codes an integer with its modulo, and the initial value became the modulo of the coded information. Both of theses AN codes can be implemented in software without relying on specific hardware, but they imply an increase of the bit-width of used integer (for example it imply a 5 bits overhead for [Goldstein2021]). This make them less suitable for resources constrained embedded applications.

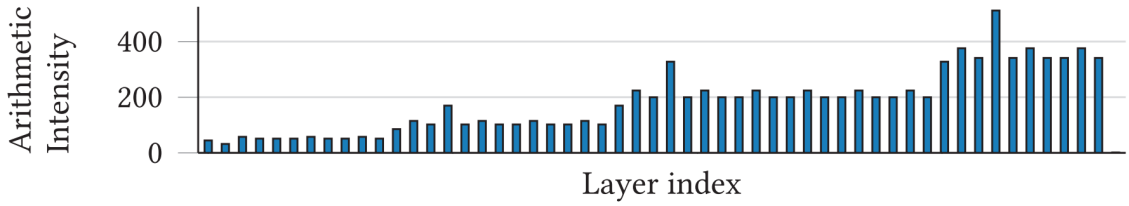
ABFT appears more adapted to DNNs since it applies to matrix multiplication accelerators, which are used in many accelerators of DNNs. [Hari2021] uses ABFT for detecting faults occurring in the convolutions of a DNN. By taking into account the existing optimizations operated during DNNs inference (e.g. fusing several operations on each neuron of certain layers), authors target to reduce the overhead induced by the usage of ABFT by 6 to 26%. [Kosaian2021] goes further in the optimization of ABFT, as illustrated in Fig.2.14. The concept of ABFT is illustrated in Fig.2.14c. Modern GPUs decompose the computations of matrix multiplications at different hierarchical levels, as illustrated in Fig.2.14a and ABFT can be performed at different level, as illustrated in Fig.2.14d. Thread level ABFT adds redundant computations, but uses data already loaded in the accelerator and hence reduces the amount of data transfer, and is adapted to layers with low arithmetic or with high memory usage. Conversely, global ABFT increases the rate of data transfer and is adapted to layers with high arithmetic intensity (significantly more arithmetic operations than memory transfers). The computing and memory usage of layers vary on modern DNNs, as shown in Fig.2.14b. Taking account of these arithmetic intensities can reduce the overhead of ABFT. [Ozen2020c] also proposes Sanity-Check,the use of linear algorithmic check sums to detect errors in computational layers of DNNs. This idea adapts ABFT to DNNs by adding neurons to the computing layers that will perform check sum and that are compared to neurons of the next layers to detect errors. The idea, very similar to traditional ABFT, is integrated in the DNN in a clever way, so it can be performed on any hardware platform.

However : Both of these approaches relying on AN codes and ABFT are derived from the scientific literature on fault tolerance for scientific computing. They imply significant overhead and are efficient to detect arithmetic faults. They detect faults inducing negligible numerical differences, whereas DNNs are tolerant to faults producing small numerical divergence, as seen in Sec.2.2.1. Consequently, we think that such techniques do not fully exploit the specificities of DNNs.

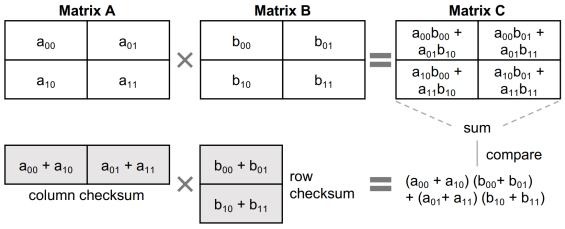
To conclude : AN codes and ABFT have been studied and adapted to DNNs however these approaches do not consider the fact that DNNs are already robust to a certain level of erroneous computations. Their deployment induce high cost that might be unnecessary.



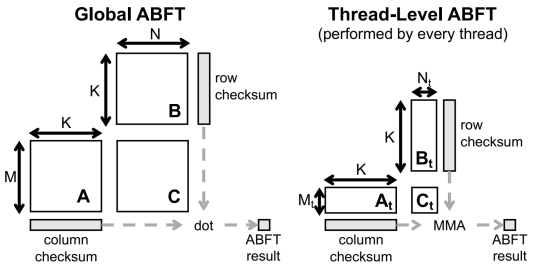
(a) Hierarchical Scopes of Matrix Multiplication on GPUs



(b) Arithmetic Intensity of Layers of a ResNet50 Model (img = 1920x1080, batch size=1)



(c) Example of ABFT



(d) ABFT at two Different Execution Scopes

Figure 2.14: Usage of ABFT on GPU-based DNNs
Figures extracted from [Kosaian2021], who proposes an algorithm to choose the best scope for ABFT for each layer.

Symptom-Based Error Detection

Observation : Syndrome-Based Error Detection (SBED) is based on watching application-specific symptoms (e.g. unusual values of variables, memory addresses or loop iterations) as an indicator of the presence of a faults. SBED has been adapted to the specificities of DNNs.

Notably : In existing works, the output value of neurons is frequently monitored. [Li2017] analyzes the numerical effect of a fault to conclude that an error leading to a SDC typically makes the magnitude of activations very large. They test a two step methodology to detect SDC. 1) During training, the maximal numerical value of neuron of each layer is recorded. 2) During inference, this value is set as threshold to detect abnormal neuron values. With an arbitrary 10% tolerance threshold (threshold = 1.1× max neuron), 90.2% of faults leading to SDCs are detected. An equivalent methodology is presented by [dos Santos2018], with an arbitrary ×10 threshold (threshold = 10× max neuron). With a different DNN, they report a 98% fault detection rate. [Chen2021] goes further and adds a new layer type to TensorFlow framework [7]. The named layer, Ranger, is trained to detect the numerical range of each layer during training, and can, during inference, reduce the range of erroneous neurons in the normal range of its layer but the authors do not clearly presents the rate of detected faults. [Schorn2018] uses the sum of every neurons in a layer as a fault detection criteria and detects 97.29% of faults.

[7] TensorFlow is an open-source Python software framework for machine learning.

SBED can also be adapted to detect faults in DNNs working on video frame. [Draghetti2019] analyses the fault tolerance of DNNs trained for video stream (e.g. autonomous driving). As temporal adjacent frames are likely to be similar, the outputs of a DNN performing task on theses frame are expected to be comparable. A high divergence between the output of DNN in two temporally adjacent frames is considered as effect of a fault. This method achieves the lowest overhead of all tested symptom-based error detection techniques as it only compares the inputs and outputs of the DNN and achieves a fault detection rate of 80%.

However : SBED is a promising technique, although much work remains to considered. All of these studies suffer from false positives (indicate the presence of a fault while no fault is present). [Li2017] considers the maximum neuron value of a layer as a fault detection threshold and reports a fault positive rate of 7.5%. With the same methodology, [dos Santos2018] reports experimental beam-testing results for GPUs implementing a similar methodology, but do not explicitly report false positive rates. [Schorn2018], that considers the sum of every neurons as fault detection threshold, achieves 5.17% false positives. The existing techniques for SBED applied to DNNs suffer from high false-positive rates, and for this reason, are not ready to be deployed in industrial applications.

To conclude : SBED can detect faults by monitoring the behavior of a DNN. Abnormal neuron activation values can be used as symptom of faults, as faults which provoke critical errors typically result in large activation values. This approach can efficiently detect more than 90% of critical faults, but induces a false positive rate that is problematic for real-world applications. Furthermore, this approach is not adapted to highly quantized DNN where the maximal value of a neuron is limited by the bidwitdh of its integer representation. Other methods consider the correlation of adjacent outputs of a CNN, but can not be applied to all types of tasks.

Error Detection During Training

Observation : The majority of existing studies focus on fault tolerance during inference, however, some authors have studied the training process. As seen in Sec.2.1.1, one metric used during training is the loss. One example of a function to compute the loss is the Multi-Class Cross Entropy as shown in Tab.2.5 The training process aim to reduce the loss by altering each weight with the back-propagation process. Unusual loss variation during training can be used to detect faults.

Formula	$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$
M	Number of classes
y	binary indicator (0 or 1) if class label c is the correct classification
p	predicted probability observation o is of class c

Table 2.5: Multi-Class Cross Entropy Loss Function

Notably : [Mahmoud2021] monitors the loss during training and detect abnormal variation of the loss dues to faults, as illustrated in Fig.2.15. Fig.2.15(a) shows an error-free inference that classify the input correctly. Fig.2.15(b) shows an example of error causing a mismatch. Fig.2.15(c) shows an example of error altering the loss without causing classification mismatch. This study considers to use localized fault injection on weights and to monitor the loss variation induced by the fault to measure the sensitivity of the faulty weight, which can then be used to strengthen the sensitive weight.

Errors during reinforcement learning can be detected with the monitoring of loss as well. [Wan2021] considers the rewards during reinforcement learning. Authors found that transient faults lead to a sudden drop in reward and permanent faults result in continuous low reward during training.

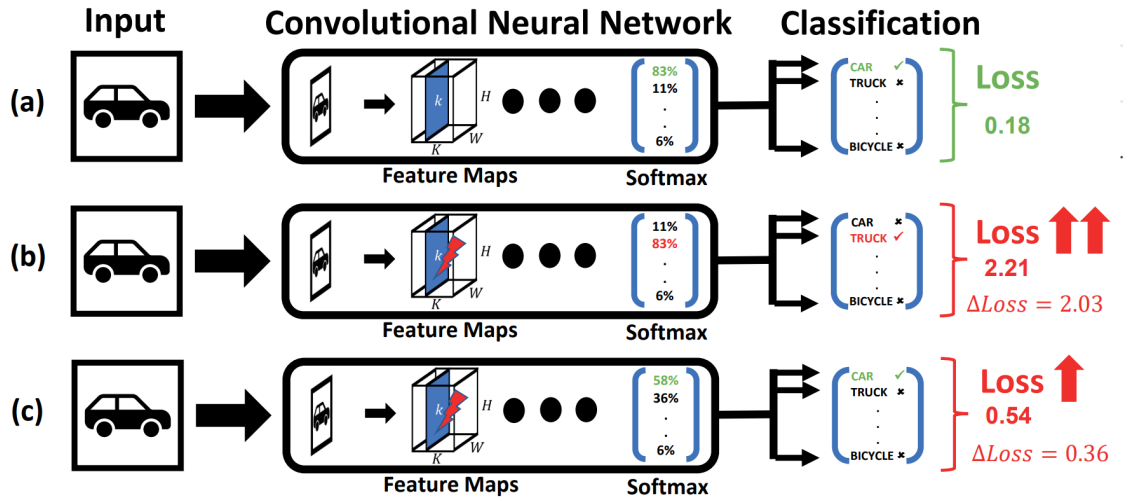


Figure 2.15: Fault Detection Based on Loss Monitoring
Example of fault detection based on loss monitoring. These figures are extracted from [Mahmoud2021].

However : The monitoring of loss during training has a limited interest for two reasons. 1) The number of cases of critical training are limited. 2) Errors during training can be skipped by coming back to a previous version of the parameters.

To conclude : The detection of errors during training can be easily detected with the monitoring of loss criterion, which can also be used to quantitatively measure the sensitivity of a part of the DNN. Transient errors can result in a sudden increase of the loss, while permanent errors may keep the loss continually high, thus preventing the convergence of the DNN. However, the real world use case of safety critical training is limited.

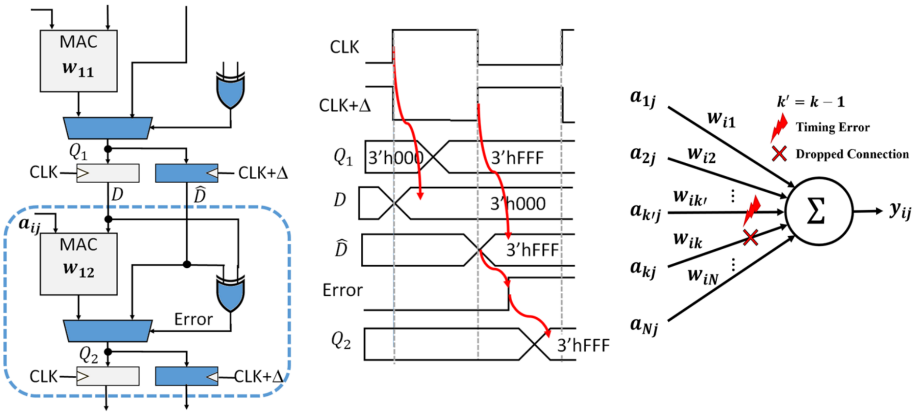
Other Hardware-based Fault Detection

Observation : Faults can also be detected by the use of dedicated hardware components.

Notably : Several authors propose hardware techniques to detect transient faults. Razor flip-flops [Ernst2004] are commonly proposed in existing works that consider the fault tolerance of DNN to detect transient faults. [Reagen2016a] and [Zhang2018a] use Razor flip-flops, as illustrated in the Fig.2.16. For example, in [Zhang2018a], in presence of 1.56% of timing errors, AlexNet on Imagenet has a Top-5 accuracy of 75% on a Razor protected accelerator, while the accuracy is random guessing without Razor protection (fault-free accuracy is 80%). The detection of timing faults is possible due to the redundant registers illustrated in Fig.2.16a. When a timing error is detected, delayed, correctly sampled value can be forwarded, short-circuiting the next MAC. This effectively results in a synapse being lost, as seen in Fig.2.16c. Razor flip-flops can efficiently detect transient or timing faults, but introduce a high hardware overhead so Razor flip-flops should be applied selectively.

However : Specialized hardware to detect permanent faults has been covered in existing studies. [Li2020a] proposes to take advantage of the fact the hardware units within a hardware accelerator are not always fully utilized and uses the cycles during which processing elements are idle to perform on-line testing on them. The idea is adapted to DNNs since it has been shown (see [Kosaian2021]) that the arithmetic intensity of layers vary significantly. This method is architecture specific, and it is difficult to guarantee the test coverage as it depends on the workload. [Motaman2019] proposes an off-line fault detection technique that considers a scratchpad-based DNN accelerator, and demonstrates that most of the hardware units can be tested.

To conclude : Dedicated hardware can be used for testing the components of DNN accelerators, covering both transient and permanent faults. Pre-existing techniques as Razor flip-flops or off-line test patterns can be used on DNN accelerators.



(a) Block-Level Diagram *(b) Impact of Timing Errors on the Waveform of Components* *(c) Impact of Timing Errors on the Computation of a Neuron*

Figure 2.16: Illustration of Razor Flip-Flop to Detect and Mitigate Timing Errors
Figure extracted from [Zhang2018a].

Conclusions on Fault Detection Techniques

Observation : Many of the fault detection techniques published are adaptions of pre-existing test techniques.

Arithmetic check-sums, can efficiently detect errors, but no not exploit the tolerance of DNNs to faults that have a small numerical impact.

Symptom-Based Error Detection techniques consider the output value of neurons, and detect faults which produce abnormally high activation values. This technique is simple and effective but is not adapted to quantized neural networks. In some cases, the analysis focuses on the output values of the final layer. Inter-frame correlation is adapted to DNNs which take video streams as input. SBED is a promising approach as the compute overhead is often small, but existing techniques suffer from a high false positive rate.

Detection of errors during training can be achieved with the monitoring of loss criterion (or reward for reinforcement learning). Both permanent and transient faults can be detected during training with this methodology.

Hardware approaches for the detection of faults proposed rely on the adaptation of techniques developed for other usages, as the Razor flip-flops, or the usage of off-line functional test, but they can require custom hardware and can not always be applied on-line.

To conclude : Some approaches to DNN fault detection consist of applying pre-existing techniques (e.g. [Li2017, Goldstein2021, Fu2021, Zhang2018a, Motaman2019]) and work remains to be done to further adapt these techniques to the specificities of DNNs and to reduce the rate of false positives, which is essential for industrial adoption. Other authors have developed techniques that are specific for DNNs (e.g. [Mahmoud2021, Chen2021, Schorn2018, Draghetti2019]). Many of these approaches are heuristics and require tuning. This is a problem when it is necessary to prove the safety of a system.

2.3.2 Fault Mitigation Techniques for DNNs

Improvement of fault tolerance of DNNs is an important field of study for critical systems using DNNs. It consists either of reducing the probability of an error to occur or to reducing the impact of errors.

Zero Masking Technique

Observation : In the previous section, many techniques for detecting faults were presented. One a specific value such as an activation is found to be erroneous, an effective technique to continue the computation with a minimal impact on accuracy, is to replace the corrupted value with zero. This simple method is illustrated in Fig.2.17 when a fault detection threshold is exceeded by an activation.

$$f(x) = \begin{cases} x, & if\ 0 \leq \mathrm{x} \leq T \\ 0, & otherwise \end{cases}$$

Figure 2.17: Masking of Neuron that Exceed a Fault Detection Threshold with Zero
Figure extracted from [Hoang2020].

Weights altered by faults can be masked using this technique. [Reagen2016a] proposes several error masking techniques, including the masking of weight faults detected with Razor flip-flops which can improve the robustness of the tested network by one to two orders of magnitude in regard to the BER, depending on the masking scope(bit or word).

Neuron faults can also be mitigated. [Hoang2020] proposes to mask the erroneous value of a neuron by zeros, and consequently increase the robustness of tested DNNs by five order of magnitude in regard to the BER for faults occurring in activations, as illustrated in Fig.2.17. This paper proposes an activation function that masks to zero the value that exceed a fault detection threshold, following methodology described in Sec.2.3.1.

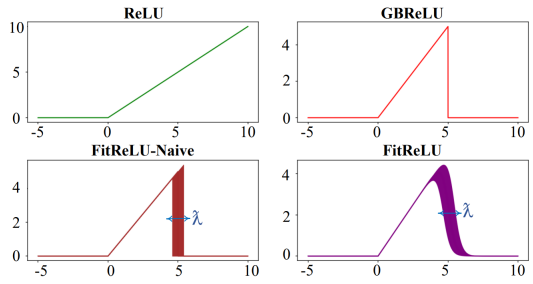
To conclude : The zero-masking is a simple technique to mitigate the faults occurring in both weights and activations. DNNs have proven to be able to absorb a certain amount of zero masked values, thus making this error-mitigating method adapted to DNNs. It can be explained by the fact that with the usage of activation functions as ReLU or Sigmoid, the zero value is common in the data-flow of DNN even without faults.

Numerical Range Restriction

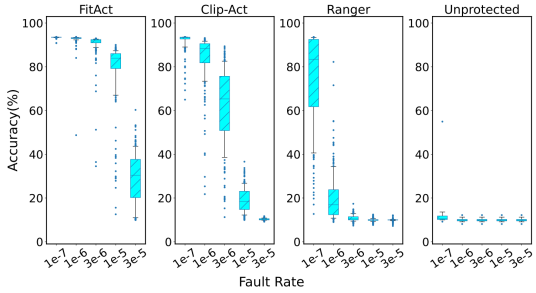
Observation : The faults that produce critical errors in DNNs typically result in high numerical values being propagated in the neurons. Consequently, some researchers consider limiting the numerical range of neurons to a predefined range to reduce the impact of faults.

Notably : [dos Santos2018] proposed the robust Max Pool layer, that detects faults with the SBED technique seen in Sec.2.3.1. Once a neuron is detected as being faulty, the robust Max Pool discards the erroneous value and returns the second highest value, assuming the latter is below the fault detection threshold. These authors report being able to correct 87% of critical SDCs. Activation functions can mask errors, as seen in Sec.2.2.1 and the use of clipping activation function, illustrated in Fig.2.18a, is an effective way to restrict the numerical range of neurons. [Hoang2020] proposes Clip-Act. It relies on costly fault injection campaigns to adjust the activation function's range to obtain optimal robustness. These authors improve the robustness of AlexNet by one order of magnitude, and VGG-16 by three orders of magnitudes. [Ozen2020a] presents a filter after the activation function that replaces erroneous values with the median of neighboring neurons. Authors report a robustness improvement comparable to the TMR.

[Ghavami2021] proposes Fit-act. With this approach, the maximal value of the activation function to the maximal value for that layer observed during training. Fit-act improves the robustness by one order of magnitude compared to Clip-Act. Finally, [Chen2021] proposes Ranger to clip erroneous values below a fault detection threshold. Results of these studies are presented in Fig.2.18b.



(a) Representation of Various Activations Functions Using the Range Restriction



(b) Model Accuracy vs Increasing Bit Error Rate for Different Numerical Range Restriction Proposed in the Literature

Figure 2.18: Numerical Range Restriction of Neurons Output to Improve Fault Tolerance
Figures extracted from [Ghavami2021].

However : The robust Max Pool Layer proposed by [dos Santos2018] is not adapted to modern DNNs. as it only works on models that use a important number of pooling layer, while modern DNNs such as MobileNetV3 [Howard2019] and RegNet [Radosavovic2020] no longer uses Max Pooling layer. Others works on clipping of activation functions are not as well adapted to quantized DNN due to their reduced numerical range.

To conclude : The numerical range restriction reduces the impact of errors. Combined with activation functions, these methods make it possible to replace erroneous values with zeros, a local median value, or a clipping threshold. Such methods have proven to be efficient for Floating-Point DNNs and solve the problem of bimodal robustness seen in Sec.2.2.1, but are less adapted to quantized DNNs.

Fault Tolerant Training

Observation : Modifications to the training process can help improve the robustness of a DNN during inference.

Training techniques for specific faults are common in existing studies and they consists of training a DNN in the presence of localized faults. [Kim2018] considers Memory Adaptive Training, a two-step process. First one considers a voltage down-scaled accelerator. A memory failure map of the SRAMs that are faulty under Vmin is generated. Then, a training process is executed, that simulate the faults during training. The trained DNN learns to adapt and recover from specific faults likely to happen during inference. With this method, the author report a limited loss of accuracy when performing under Vmin (4% accuracy loss with this training methods versus 70% for the baseline, for a energy reduction from 0.9V to 0.5V). [Zhang2019] disables computation of faulty processing elements and performs simulations during training to reproduce this behavior. The authors report that DNNs can be trained in the presence of permanent faults (pruned computations) even when half of the processing elements are disabled. These results were obtained on LeNet-5 and AlexNet models and may not scale as well to compressed DNNs. [Abdullah Hanif2020] proposes a similar technique to train a DNN for faulty hardware. The presented methodology, Salvage-DNN, allows retraining a DNN to perform in the presence of permanent faults, based on fault-aware mapping to avoid the usage of faulty components.

Training technique where the actual faults are not known have also been proposed. As seen in Sec.2.1.1, Dropout [Gal2016] randomly masks a given fraction of neurons in the last layers of a DNN to improve generalization, resulting in an improvement in the accuracy. Typically used on the last fully-connect layers, [Solovyev2019] generalized dropout to all layers to efficiently improve the robustness of DNN to random errors in weights. With this fault model applied to one single weight of the first convolutional layer of a VGG-16 model, the rate of SDC goes from 6.83% without Dropout to 1.09% with a 10% Dropout. Errors caused by emerging hardware are examined by [Li2020b], who uses fault injection to modify the loss criterion by addition of a random noise, in order to train a DNN to reliably perform using unreliable component (namely Resistive Random Access Memory). Finally, the quantization process is studied by [Stutz2021] who proposes the use of a random bit error rate during quantization process to improve the robustness of quantized DNNs to various errors.

To conclude : Many authors have shown that modifying the training procedure can improve the robustness of a DNN. Existing works covers both the fault-aware training, where the focus is to work around known faulty hardware units and training techniques where the actual faults are not known during training. Fault aware-training is highly effective ([Zhang2019] claims that DNNs perform well even when 50% of the processing elements are faulty). Training a DNN is highly compute intensive ,thus it is unrealistic to imagine that customized training can be performed for every faulty components. Training techniques that do not require knowledge of specific faults are less efficient, but do not impose such strong assumptions.

Selective Modular Redundancy

Observation : Selective modular redundancy can improve robustness and reduce the overhead compared to full TMR. Selective modular redundancy is adapted to the specificities of DNNs as it protects only the most sensitive elements. Existing studies consider the selective modular redundancy of bits, channels, layers or even of the whole model.

Bit level redundancy has been explored in these works. As seen in Sec.2.2.1, all bits are not equivalently robust and MSBs are more sensitive to faults than LSBs. [Mahdiani2012] protects the MSB of DNN accelerators with hardware redundancy. The idea is simple and can be customized as the ratio of overhead to robustness can be adjusted.

Convolutional channel redundancy protects only the most sensitive channels. Authors of [Gambardella2019] propose a two step method for increasing the robustness of a DNN. 1) A fault injection campaign is executed to detect the most sensitive channels of a DNN. 2) These channels can be triplicated with TMR. For the tested 4-bits quantized DNN this method achieves results equivalent to TMR but require a 179.6% overhead (compared to 200 % for TMR).

Layer redundancy is proposed by [Khoshavi2020]. This study uses FPGA and FINN synthesizer [Umuroglu2017] to add redundancy to the layers that have the most sensitive weights and neurons. For 100 simultaneous Multiple Bit Upsets, the error resiliency of the tested binarized model is doubled.

Entire model redundancy is proposed as well. [Gao2022] studies the robustness of ensemble networks compared to a single larger network. By replacing a ResNet 101 with several smaller ResNets, the authors can improve the robustness of the DNN, while maintaining equivalent accuracy. Compared to TMR applied to the baseline DNN, authors report equivalent robustness while reducing the hardware overhead by 33%.

To conclude : Selective modular redundancy can improve the fault tolerance of a DNN, and redundancy can be added at different level of granularity as bit, layer, channel or the whole model. This approach can reduce the overhead compared to naive redundancy as TMR by 10 to 33% while obtaining equivalent robustness.

Fault Tolerant Scheduling

Observation : Some DNN hardware accelerators rely on Single Instruction Multiple Data parallel execution. On such machines, a means mechanism to mitigate errors consists into of avoiding use of faulty computing units by altering the scheduling.

Notably : [Abdullah Hanif2020] present a fault aware mapping to a systolic accelerator that avoids to schedule important weights on processing elements with permanent faults. The proposed systolic accelerator is derived from [Zhang2019] and can prune the computations of faulty processing elements (skipping/masking them to zero). The fault aware mapping is a two-step process. 1) Computation of the sensitivity of each DNN weight. 2) The weight with the least importance are mapped to components that will get its computations skipped during inference. This is made possible by the fact that one weight is scheduled on specific processing elements on this architecture. The usage of this specific scheduling can improve the robustness of the resulting DNN accelerator. For example, with this scheduling on 8-bit quantized VGG-11 trained for CIFAR-10, the baseline scheduling does not know any accuracy loss with up to 2% of faulty PEs, while the same accelerator with fault tolerant scheduling can operate with 6% of faulty PEs without accuracy loss.

Furthermore : [Liu2021] proposes to simply remap the faulty/skipped operations on reliable component, thus adding temporal redundancy to avoid any loss of accuracy caused by permanent faults. Authors report that on average the tested DNN systolic accelerator consisting of an array of 32x32 PE is fully functional with less of 3.13 % of faulty PE, but does not explicitly show the latency overhead of this technique.

To conclude : Some authors have considered altering the scheduling of computations of a DNN accelerators to improve its reliability. Most studies focus on permanent faults on systolic accelerators. The specificity of DNN allows mapping unimportant weights to faulty PEs. The simple re-scheduling of operation to non-faulty hardware has been studied, but does not exploit specificities of DNNs.

Conclusions on Fault Mitigation Techniques

Existing works covers several techniques to improve the robustness of DNNs, that consider the specificities of DNNs.

The **Zero masking technique** is a simple but effective method to reduce the impact of a fault. Due to the inherent redundancy of DNNs, models can endure many values being masked to zero either in their weights, neurons, or computational value, without exhibiting a significant loss of accuracy.

Numerical range restriction uses the fault detection threshold seen in Sec.2.3.1 to prevent excessive large numeric values. These approaches are simple and effective for floating-point-based DNNs, but their applicability to quantized DNNs remains to be proven.

Fault tolerant training can play a role in the robustness of the final model as well. Existing studies present different fault tolerant training techniques, that can be divided into two groups. 1) Those aware of the specific faults 2) Those not aware of the actual faults and which use random faults during training.

Modular redundancy adds redundancy to the most critical parts of a DNN. Redundancy at different levels of granularity (bit, kernel, channel, layer) have been also proposed. With ensemble networks, the redundancy is performed at the full network level.

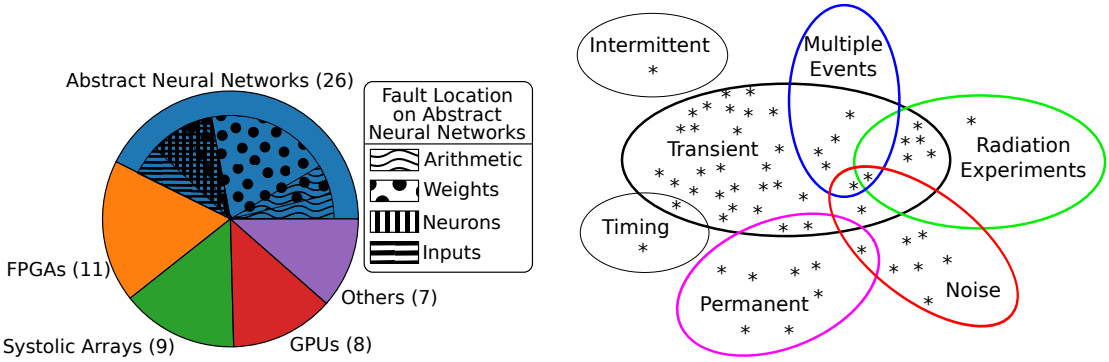
Scheduling strategies were discussed in Sec.2.3.2. Due to the highly parallel nature of modern DNNs accelerators, scheduling strategies can avoid the use of faulty units for important computations. Temporal redundancy is also considered but does not benefit from the specificities of DNNs.

2.4 Summary of Existing Studies

This section quantitatively analyzes topics analyzed by the existing works by studying the number of articles that address specific topics so the ideas presented in the previous sections can be put in perspective to highlight the current focus of this field of research. The topics cover the case study, the hardware platform and the fault model.

This section is organized as follows :

- **Sec.2.4.1** presents how the cited articles focus on faults.
- **Sec.2.4.2** covers how considered works focus on faults analysis, detection and mitigation.
- **Sec.2.4.3** analyzes the use cases (networks and data sets).



(a) Part of Hardware Target in the Literature
For each considered hardware target is displayed the number of articles that uses them.

(b) Studied Faults in the Literature
Each * represent an article, showing the most studied faults

Figure 2.19: Summary of the Fault Models in the Literature

2.4.1 Hardware Targets and Fault Models

Several hardware targets are analyzed in the considered works, as seen in Fig. 2.19a. 57% of the works consider a specific hardware target, and the others 43% study the effect of faults on Abstract Deep Neural Networks, seeing the DNN as a mathematical model.

Hardware-focused articles are equally distributed between *FPGAs* (31.4%), *Systolic*-ASIC-based accelerators (25.7%). and *GPUs* (22.9%). The remainder, labeled *Others*, (20%) cover various hardware targets that consider a fault in one hardware element (e.g. specific SRAM Cells [Azizimazreah2018] or issues with emerging technologies such as memristors [Li2020b]).

A large fraction of the existing studies does not consider specific hardware platforms. 50% of them focus on faults in the weights. Others analyze theoretical components such as the neuron, activation, or feature map (7.7%), the inputs (7.7%), and arithmetic faults (7.7%). Several articles consider the cumulative effect of multiple fault models (19.2%). Finally a few articles inject faults in abstract DNN without specifically defining the fault target (7.7%).

Many types of faults are currently studied. The most represented fault models are *Transient* fault models (40 studies). Among studies that use this fault models, 34 considers single event upsets, and 6 multiple event upsets. The next most used fault models are *Permanent* faults, and the *Noise* [8] (10 studies each). The rarest fault models are *Timing* faults (2 article) and *Intermittent* faults (1 article). Finally, 7 studies consider real-world beam experiments to assess the fault tolerance of DNNs instead of using simulated fault models.

[8] Noise fault model is a linear transformation applied to a target.

2.4.2 Fault Tolerance of DNNs

The most analyzed components or characteristic of DNNs for fault tolerance are the layers analyzed to compare their individual sensitivity (16 studies) and the role of data-type on the robustness of the whole model (11 studies). Other studies consider the sensitivity of neurons (5 studies) and bit positions (4 studies). The estimation of the robustness of the model without performing fault injection campaign is considered (4 studies), as well as the role of pruning (2 studies) and the effect of timing faults (2 studies) on the robustness of the studied model.

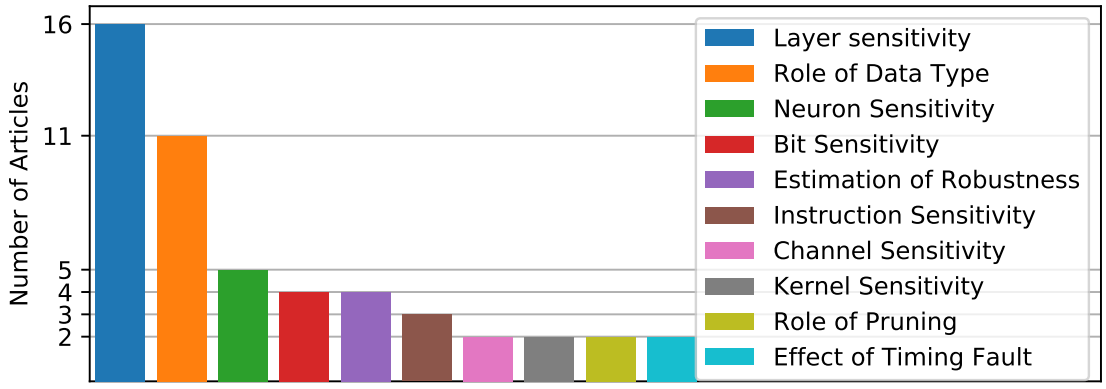


Figure 2.20: Report of the Number of Articles that Study Specific Parts of DNNs

The most commonly considered fault detection techniques are the detection of faults based on abnormal range of neurons (9 articles), the usage of redundancy for fault detection (7 articles) or the usage of check-sums to detect arithmetic errors (5 articles). Well-known techniques based on test patterns are used to detect faults (2 articles), as well as the monitoring of loss variation for detecting training fault during training (2 articles). Finally, 1 article proposes Inter-Frame Correlation to detect faults occurring on a video-based dataset.

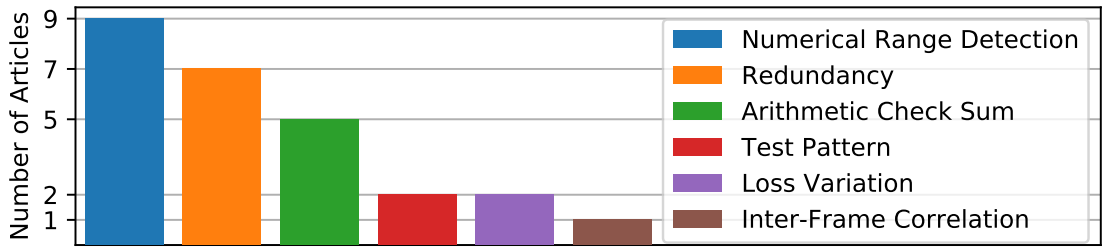


Figure 2.21: Report of the Number of Articles that Employ Specific Fault Detection

The most popular techniques for improving the robustness of a DNN in cited studies are the usage of modular redundancy (9 articles), fault tolerant training (9 articles), the usage of zero-masking to mitigate errors (8 articles) and scheduling strategies to avoid using faulty components (6 articles). The numerical range restriction (4 articles) as well as the the masking of erroneous value with median of neighbors (1 article).

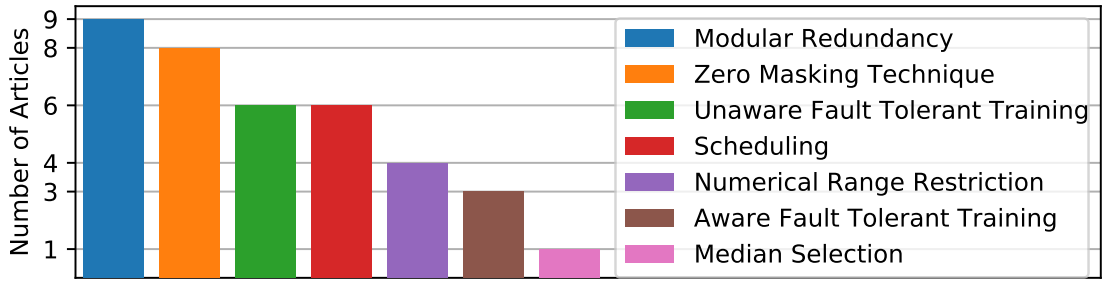


Figure 2.22: Report of the Number of Articles that Use Specific Methods to Improve Robustness

All the techniques in the state of the art are not equally used. The most common analyses of fault effects in this field of research consider the role of layers and data-types in the robustness of the model.

The most used method to detect faults are the detection of abnormal numerical range of neurons, the usage of redundancy, and the usage of arithmetic check sums to detect faults occurring during computations.

The most adopted technique to improve the robustness of a DNN rely on modular redundancy, scheduling strategies and fault tolerant training. Once errors have been detected, the most common methods to mitigate it consists into masking the error with zero.

2.4.3 Data-sets and DNN Topologies

This section presents the case study in the works that consider the fault tolerance of DNNs. It considers the data sets and the topologies of DNNs. The cited works are listed in in Tab.2.7 and the usage of case studies is represented in Fig.2.23 and 2.24.

A strong tendency that can be observed is that nearly all the studies uses computer vision data sets, and image classification is by far the most used task (65 articles). A smaller number of studies consider the object detection (5 articles). To the best of our knowledge, no articles consider semantic or instance segmentation [9]. The three most used data sets are CIFAR (21 articles), MNIST (20 articles) and ImageNet (19 articles). Others data sets such as GTSRB, Pascal VOC, Caltech and COCO are less used. The *Other* category includes various other data sets (Reuters, flora, etc.) and various tasks (regression, reinforcement learning, classification, etc.) The data sets used only once are regrouped in this category [10].

A broad number of DNN topologies are considered. Four of them are largely represented : *ResNet* (19 articles), *VGG* (19 articles), *LeNet* (18 articles) and *AlexNet* (17 articles). Small convolutional neural networks with fewer than six layers are grouped under *LeNet*, even if they are not explicitly named so. Aside from these DNNs, several others topologies are sparsely represented. We note that topologies designed for embedded systems (*MobileNet*, *SqueezeNet* and *ShuffleNet*) are sparsely represented. Finally, the most modern and efficient topologies (*MobileNetV3*, *MNasNet*, *RegNet*) are absent from the studied works.

2.4.4 Summary Table

Tab.2.6 and Tab.2.19a summarizes the best papers in the scientific literature of fault tolerance of DNN. For each of them, the relation with used fault models and case studies is established.

[9] COCO is a data set that handles object detection and semantic segmentation tasks. The article that uses the COCO dataset uses the object detection labels of this data set.

[10] With an exception of the COCO dataset. We believe that COCO dataset is closer to real-world critical system than the others data sets merged into the Other category. Consequently, we did not include COCO dataset in the Other category

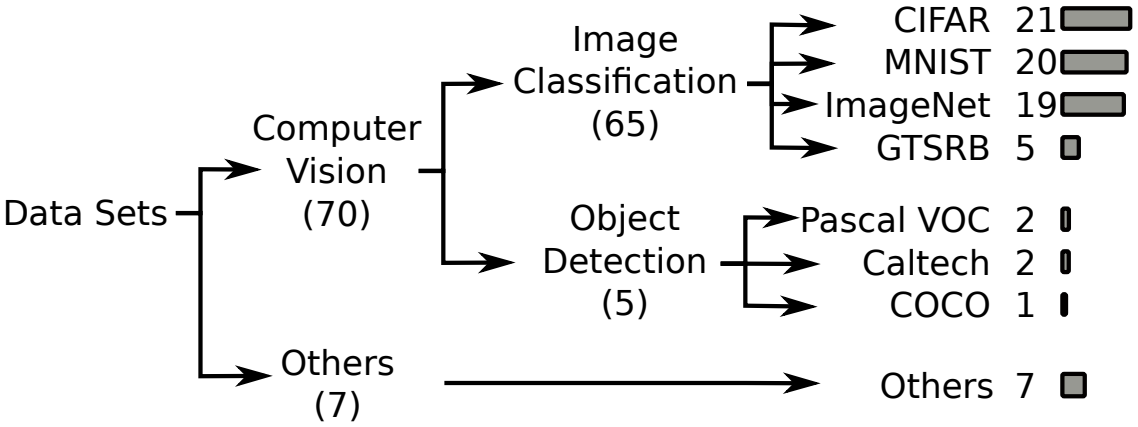


Figure 2.23: Data Sets Used in the Scientific Literature

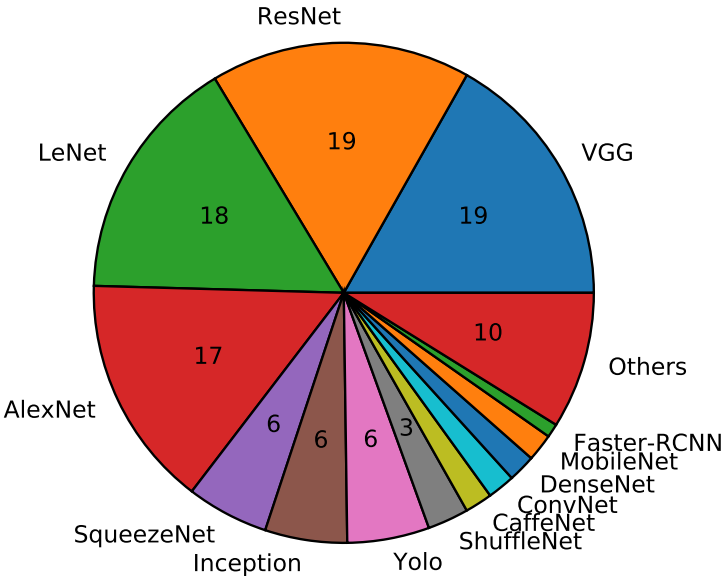


Figure 2.24: DNNs Topologies Used in the Scientific Literature

Article		Hardware target					Fault analyzed								Fault target					Fault impact analysis												Fault tolerance improvement													
																																Fault detection							Error mitigation						
Reference	Citations	Abstract neural networks	GPU	Systolic arrays	FPGA	Others	Intermittent fault	Timing Fault	Permanent faults	Transient faults	Multiple Events Upsets	Radiation experiments	Noise	Arithmetic	Weights / Synapse	Neurons / Activations	Inputs	Layer vulnerability analysis	Channel vulnerability analysis	Kernel vulnerability analysis	Neuron vulnerability analysis	Instruction level analysis	Estimation of robustness	Data-type impact	Pruned DNNs	Fault timing analysis	Bit sensitivity analysis	Redundancy	Omniscient	Arithmetic check sum	Numerical range detection	Inter-frame correlation	Test pattern	Loss variation	Zero masking technique	Scheduling strategy	Modular redundancy	Numerical range restriction	Aware fault tolerant training	Unaware fault tolerant training	Median selection
[Abdullah Hanif2020]	13			X					X										X		X													X	X			X			
[Adam2021a]	3		X							X								X		X		X						X								X					
[Adam2021b]	3		X							X												X																			
[Arechiga2018]	18	X								X					X																										
[Azizimazreah2018]	34					X				X	X							X									X											X				
[Badia2021]	0		X									X																								X					
[Chen2021]	9	X								X	X					X					X									X								X			
[Couellan2021]	4	X											X				X					X																			
[Draghetti2019]	3		X							X																				X	X				X					
[Fu2021]	0	X											X	X															X											
[Gambardella2019]	18				X				X										X				X														X				
[Gao2022]	0				X					X																											X				
[Gao2020]	3	X								X	X				X									X		X															
[Ghavami2021]	0	X								X					X						X									X							X			
[Goldstein2021]	1	X								X				X														X												
[Hari2021]	13	X								X		X			X	X	X											X												
[Hoang2020]	35	X								X					X			X			X																X			
[Ito2021]	2		X							X		X																												
[Khoshavi2020]	4				X					X	X							X																			X			
[Kim2018]	39					X	X																				X									X				
[Kosaian2021]	1		X							X				X														X									X			
[Kwon2016]	11	X											X		X			X																			X			
[Li2020a]	7				X					X				X		X										X						X		X						
[Li2020b]	14					X			X				X	X	X																									X	
[Li2019]	5				X			X																	X																
[Li2017]	260	X		X						X					X			X			X			X		X				X							X			
[Libano2021]	7				X					X		X												X													X			
[Libano2020]	15				X					X		X												X													X			
[Liu2021]	1			X					X																		X												
[Mahdiani2012]	43					X																					X									X			
[Mahmoud2021]	0	X								X	X		X			X		X		X													X						
[Malekzadeh2021]	0	X								X					X			X								X														
[Motaman2019]	6					X			X																						X									
[Neggaz2019]	19	X								X					X			X																						
[Ozen2020a]	6	X		X						X					X	X																								X
[Ozen2020b]	8	X								X																				X				X						
[Ozen2020c]	3			X						X																		X												
[Reagen2018]	174	X								X					X	X							X																	
[Reagen2016a]	545					X				X													X				X							X						
[Sabbagh2019]	15	X								X					X								X	X																
[Salami2018]	48				X				X	X								X									X							X						
[dos Santos2021]	2		X							X			X									X																		
[dos Santos2018]	75		X							X	X	X	X					X												X				X			X			
[Schorn2018]	38			X						X																				X										
[Solovyev2019]	8	X											X		X																								X	
[Souvatzoglou2021]	0				X					X								X																						
[Stutz2021]	10	X								X					X									X															X	
[Syed2021]	0	X								X					X			X						X																
[Wan2021]	2	X							X	X				X	X			X						X							X			X	X			X		
[Webb2018]	38	X																					X																	
[Wu2021b]	0			X						X																													X	
[Wu2021a]	0					X			X															X							X									
[Xiang2019]	3	X											X		X								X																	
[Xu2021]	8				X				X													X																		
[Xu2019]	8	X			X					X					X	X	X	X																						
[Yang2017]	49									X			X				X																						X	
[Yu2019]	26	X											X				X						X																	
[Zhang2018a]	96			X				X		X								X								X	X									X				
[Zhang2019]	25			X					X																			X							X				X	

Table 2.6: Summary of Hardware Considerations in the Literature

Article		Data set									Network Topology													
Reference	Citations	Caltech	Pascal VOC	ImageNet	CIFAR	COCO	MNIST	GTSRB	Others	AlexNet	CaffeNet	ConvNet	DenseNet	Faster-RCNN	Inception	LeNet	MobileNet	ResNet	SqueezeNet	ShuffleNet	VGG	Yolo	Others
[Abdullah Hanif2020]	13			X	X													X			X		
[Adam2021a]	3			X	X								X										
[Adam2021b]	3			X						X													
[Arechiga2018]	18			X											X			X			X		
[Azizimazreah2018]	34			X						X	X										X		
[Badia2021]	0																						
[Chen2021]	9			X	X		X	X	X	X						X		X	X		X		X
[Couellan2021]	4																						
[Draghetti2019]	3	X												X								X	
[Fu2021]	0																						
[Gambardella2019]	18				X		X																X
[Gao2022]	0				X													X					
[Gao2020]	3				X																X		
[Ghavami2021]	0				X					X								X			X		
[Goldstein2021]	1			X	X		X			X		X				X		X			X		
[Hari2021]	13			X														X			X		
[Hoang2020]	35				X					X											X		
[Ito2021]	2					X																X	
[Khoshavi2020]	4				X																X		X
[Kim2018]	39						X		X														
[Kosaian2021]	1			X						X			X					X	X	X	X		X
[Kwon2016]	11						X									X							
[Li2020a]	7				X										X		X	X					
[Li2020b]	14				X												X	X			X		
[Li2019]	5									X						X					X		
[Li2017]	260			X	X					X	X	X											X
[Libano2021]	7						X									X							
[Libano2020]	15						X									X							
[Liu2021]	1			X						X								X			X	X	
[Mahdiani2012]	43																						
[Mahmoud2021]	0			X						X					X			X	X	X			
[Malekzadeh2021]	0						X									X							
[Motaman2019]	6																						
[Neggaz2019]	19			X						X					X				X		X		
[Ozen2020a]	6				X		X	X								X		X	X				
[Ozen2020b]	8				X		X	X		X						X		X	X				
[Ozen2020c]	3							X		X													
[Reagen2018]	174			X	X		X		X							X		X			X		X
[Reagen2016a]	545						X									X							
[Sabbagh2019]	15				X		X									X					X		
[Salami2018]	48						X		X							X							
[dos Santos2021]	2																					X	
[dos Santos2018]	75	X	X												X			X				X	
[Schorn2018]	38				X			X															X
[Solovyev2019]	8				X																X		
[Souvatzoglou2021]	0						X									X							
[Stutz2021]	10				X		X																X
[Syed2021]	0						X									X							
[Wan2021]	2															X							
[Webb2018]	38																						
[Wu2021b]	0				X													X					
[Wu2021a]	0			X											X			X		X			
[Xiang2019]	3																						
[Xu2021]	8		X	X					X									X				X	
[Xu2019]	8			X						X											X		
[Yang2017]	49						X																
[Yu2019]	26																						
[Zhang2018a]	96			X			X		X	X						X							X
[Zhang2019]	25			X			X		X	X						X							X

Table 2.7: Summary of used DNNs and Data-Sets in the Literature

2.5 Conclusions on the State Of The Art of Robustness of DNNs to Hardware Faults

This section concludes the presentation of existing works that focus on the fault tolerance of DNNs. The field of research of fault tolerance of DNNs is complex and covers many aspects. Because the hardware robustness of an application is intimately connected to the hardware platform dedicated to its execution, it appears difficult to draw general conclusions on the robustness of DNNs. To tackle this issue, part of the existing studies considers abstract DNN, without considering specific hardware implementation, to draw general conclusions.

Abstract DNNs : Key observations on the articles related to abstract DNN are that data type used by DNNs plays a major role on their robustness. Quantized DNNs improve the fault tolerance of a DNNs by several orders of magnitude, mainly because the maximum numerical impact of a fault is limited by the data type, while floating-point-based DNNs, without a specific fault mitigation strategy, are subject to critical faults even at a low bit error rate. Pruning, another compression method for DNNs, also improves the robustness, but less significantly than quantization. DNNs benefit from an inherent error masking ability due to their activation function and their pooling layers, but their efficiency to mask faults is limited. All parts of DNNs are not equally robust to faults. The evaluation of the robustness of specific kernels or channels relies on computationally expensive fault injection campaigns, and no alternative method to estimate their robustness has yet proven to be effective. Concerning layers, the last fully-connected layers are more sensitive than the first convolutional ones. Likewise, a consensus exists on the fact that a fault altering an activation is less likely to triggers an SDC than a fault occurring in a weight, which could be explained by the important rate of reuse of weights. However, these conclusions, drawn from the study of abstract DNNs, can not completely predict a real-world Architectural Vulnerability Factor of a DNN, as they do not exhaustively cover all possible hardware failures. Yet, they give useful insight to improve the robustness of topologies of DNNs and their accelerators.

Analysis of DNN hardware accelerators : Specific works that are much closer to real hardware implementation do exist. They mainly focus on GPUs and FPGAs on one side, because these are common DNN accelerators, and on SRAM cells on the other side, because these cells, generalized among modern accelerators can be voltage-down-scaled to significantly reduce the power consumption of an accelerator at the cost of degraded reliability. The studies on fault tolerance of GPU-accelerated-DNN mainly focus on the analysis of the robustness of DNN on fault injection with general-purpose fault injection tools. The observation was made that such software-based tools are not enough to model the robustness of GPU-accelerated-DNN, probably because configuration memory are sensitive to faults and faults can not be simulated in these components by software developers. The studies made for FPGAs come to equivalent conclusions. FPGAs seems to be sensitive to faults altering their CSRAM, which prevent the FPGA-based DNN accelerators to benefit from the DNN fault tolerance, because their configuration memory is less robust to the same bit error rate than DNNs. The studies that consider the globally used SRAM Cells mainly focus on the power efficiency that can be obtained from the voltage-down-scaling of such cells without great effort due to the inherent robustness of DNNs.

Fault detection plays a major role in the robustness of DNNs. The studies that focus on fault detection of DNNs have limited scientific innovation, as most of the related articles rely on the adaptation of already existing techniques for DNNs. The usage of arithmetic check sums was tested and can efficiently detect faults, but does not benefit from the specificities of DNNs (e.g. sparsity or tolerance to errors that have a limited numerical impact). The Symptom-Based Error Detection techniques, traditionally used in general-purpose computing to watch the value of variables, memory space addressing or loop iterations were adapted to DNNs. The most widespread technique consists in watching the value of neurons to detect abnormal magnitude of activations, classified as effect of errors. This approach is low-cost and effective, but induces false positives and is not adapted to quantized DNNs, which have a limited range of possible values even without

faults. Both transient and permanent faults occurring during training can be easily detected by the watching of loss criterion, but the interest of this method is limited, as the training is stochastic-step-based and a fault occurring in one epoch can be ignored at next one. Furthermore, there appears to be less need for fault tolerance during training. Finally, several studies use hardware approaches to detect faults, such as Razor flip-flops or the usage of test patterns. These methods can be adapted to DNNs but their applications to DNNs is not different than to general purpose computers.

Fault mitigation of detected faults has also been studied. Most of these techniques rely on the specificities of DNNs. The most widespread technique to mask the effect of faults is the zero masking technique. Due to their inherent redundancy, DNNs can absorb a certain number of zero masked values either in their weights, neurons, or computational value, without altering their behaviors. In addition, the intrinsic error masking ability of DNNs seen previously can be improved. Many works propose to restrict the numerical range of neurons to limit the effect of faults. This approach is effective for floating-point-based DNNs, but is unadapted to quantized DNNs. Fault injection during training can be used to improve the robustness of DNNs. The training methods can be divided into two categories : fault aware training, that known the exact location and effect of a fault and that is typically used to recover from permanent faults, and fault unaware training, that considers random faults during training and is typically used to improve the robustness of DNNs to transient faults. Selective modular redundancy is used to add redundancy to the most critical part of DNNs. Redundancy can be added at several scopes : bit, channel, layer. Finally a different approach considers ensemble networks to add redundancy at the topology-level to improve both robustness and accuracy of models. Finally, as the acceleration of DNNs relies on massively parallel execution, scheduling strategies avoid the execution of important computation on unreliable components. Temporal redundancy is considered as well.

To conclude, the scientific literature on fault tolerance of DNNs covers many aspects. The major aspect to improve the robustness of DNNs relies on the usage of quantized DNNs to limit the numerical impact of an error. Techniques used to detect faults mainly rely on the usage of well-known techniques traditionally used in general purpose computing that are adapted to DNNs. However, techniques for improving the robustness of DNNs intimately rely on specificities of DNNs, as the fault tolerant training or the usage of modular redundancy to protect the most sensitive parts of DNNs. We note however, that the existing works struggle to significantly improve the robustness of quantized DNNs, and that most of fault detection techniques are not adapted to quantized DNNs.

2.6 Contributions of the Thesis

This section put the contributions of this thesis into perspective with respect to the existing research on the robustness of DNNs.

Chapter 3 includes a study comparing existing architectures of Systolic-Arrays and analyzes their fault robustness to permanent faults in their data-paths. As seen in Sec.2.4.1, the three most studied hardware platform in existing studies are : the GPUs, the FPGAs, and the ASIC integrating on Systolic-Arrays. While these works study the hardware robustness of specific architectures, few studies compare existing architecture based on Systolic-Array.

Chapter 4 presents a systolic-array-based architecture designed for Fault Tolerance, based on the conclusions of the previous chapter. This architecture integrates on-line testing to identify permanent faults, something that has not often been applied to DNN accelerators. The proposal is also able to mask faults with minimal impact on accuracy. This architecture has led to two scientific publications [Burel2021b, Burel2022c] and one patent [Burel2021a].

Chapter 5 proposes a fault detection and mitigation mechanism for transient faults in the weight memory of a DNN. The fault detection of transient faults, as seen in Sec.2.3.1 does not

always benefit from the specificities of DNNs. Their LSBs, as seen in Sec.2.2.1 are robust to fault. We provides a technique that codes a bit-parity in the LSB of weights to detect soft errors. This methodology has led to one scientific publication in an international conference [Burel2021c].

Chapter 6 proposes a new SBED technique that improves fault detection and eliminates almost all the false positives. We have seen in Sec.2.3.1 that SBED is a low-cost effective method but cab produce false positives that are be undesirable in commercial systems. We propose an improvement to existing SBED techniques that reduces the rate of false positives and increases the false detection rate. While the existing studies focus on the maximal value of neuron to set an arbitrary fault detection threshold, we propose to analyze the standard deviation of the dataflow to set an adaptive fault detection threshold, and to monitor multiple statistics (maximum, average, minimum and standard deviation) to detect faults. This technique has led to one scientific publications in an international conference [Burel2022b].

Articles

[Burel2021b]	Stéphane Burel, Adrian Evans et Lorena Anghel. MOZART: Masking Outputs with Zeros for Architectural Robustness and Testing of DNN Accelerators. In 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS), pages 1–6. IEEE, 2021.
[Burel2021c]	Stéphane Burel, Adrian Evans et Lorena Anghel. Zero-Overhead Protection for CNN Weights. In 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pages 1–6. IEEE, 2021.
[Burel2022c]	Stéphane Burel, Adrian Evans et Lorena Anghel. MOZART+: Masking Outputs with Zeros for Improved Architectural Robustness and Testing of DNN Accelerators. IEEE Transactions on Device and Materials Reliability, 2022. 2022.
[Burel2022b]	Stéphane Burel, Adrian Evans et Lorena Anghel. Improving DNN Fault Tolerance in Semantic Segmentation Applications. In 2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), pages 1–6. IEEE, 2022.

Patents

[Burel2021a]	Stéphane Burel, Adrian Evans et Lorena Anghel. Accélérateur systolique de réseau de neurones et système électronique et procédé de test associés. In Patent FR2105808, 2021.
[Burel2022a]	Stephane Burel et Adrian Evans. Calendrier de fautes pour accélération de l'analyse de fiabilité des circuits intégrés. In Patent FR2205817, 2022.

Table 2.8: Contributions of this Thesis Published in Articles and Patents.

3

Robustness of Deep Neural Network Systolic Accelerators

Deep Neural Networks are being increasingly used in mission critical applications making it important to have a clear understanding of how hardware faults impact their operations and accuracies. In the past, DNN fault tolerance studies focused on faults in the abstract network, evaluated with floating point numbers. However, for inference, DNNs are often executed on custom hardware where an array of Processing Elements (PEs) is used to execute each layer, typically performing computations using a fixed-point numeric format. Due to the re-use of the PEs, a single hardware fault translates to multiple faults in the abstract network. In this chapter, we study three systolic hardware architectures and present experimental results from a fault injection study based on large DNN models.

As we have seen in chapter 1, Deep Neural Network (DNN)s can be deployed in safety critical applications, thus, it is essential to understand the impact from hardware faults on accelerator likely to be used in such applications.

The state of the art regarding the fault tolerance of DNN hardware accelerators has been presented in Sec.2.2.2 and focuses on three types of platforms : Graphics Processing Unit (GPU)s, Field-Programmable Gate Arrays and the power-efficiency improvement offered by voltage downscaling of Static Random Access Memory (SRAM).

Two well-known industrial DNN accelerators that consider the safety have been publicly presented. The first is a circuit from Tesla [Talpes2020] incorporating a SIMD systolic DNN accelerator and it achieves robustness by using DMR at the chip-level. The other [Matsubara2021], a commercial automotive SoC from Renesas, utilizes on-chip redundancy to meet safety requirements. In both cases, the safety goal is achieved by DMR and neither exploits the intrinsic fault robustness of DNNs.

In this chapter, we analyze the robustness of existing systolic DNN accelerators to faults occurring in their arithmetic units and we draw conclusions for the design of robust accelerators.

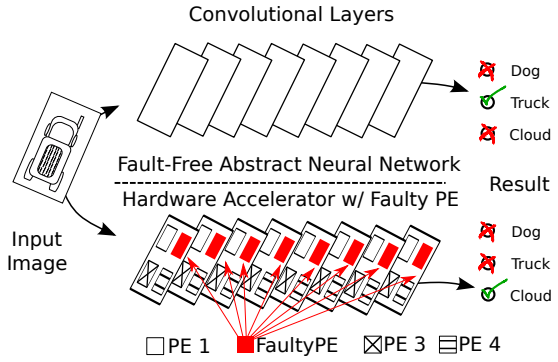


Figure 3.1: Projection of a Hardware Fault onto the Abstract Network

In practice, DNNs require an large number of identical operations, namely Multiply And Accumulates (MACs). Thus, accelerators with a systolic data-path are well suited to accelerate DNNs. These accelerators are composed of a large number of Processing Element (PE)s that process and transfer data to neighboring PEs. However, modern DNNs are composed of millions of neurons, thus it is not possible to dedicate one hardware PE for each neuron. Consequently, a DNN must be mapped to the systolic array in such a way that a given PE is re-used for the computation of multiple neurons. A single hardware fault in a PE can produce multiple errors in the abstract network and thus potentially affect the behavior of the executed DNN. This concept is illustrated in figure 3.1, where a fault in one PE affects the computations for multiple activations in the abstract model. Since the same PEs are used for all the layers, every layers will be affected by a hardware fault. A failure analysis is therefore necessary to understand the behavior of DNN executed on a systolic architecture.

The contributions of this chapter are : 1) A comparison of the impact of equivalent arithmetic faults on three different architectures for a DNN accelerator, 2) An analysis methodology based on considering *worst case* classification accuracy for batches of images which is important for understanding the impact of a fault, 3) An analysis of the impact of *folding* on fault tolerance, given several type of arithmetic faults.

This chapter is organized as follows : Sec.3.1 presents the hardware architectures that were analyzed. Sec.3.2 presents the fault injection methodology used in our experiments. Sec.3.3 presents our experimental results. Sec.3.4 compares our results with the scientific literature. Sec.3.5 presents idea for future work. Finally, Sec.3.6 concludes this chapter.

3.1 Hardware Architectures

In this section, the architecture and data-flow of systolic DNN accelerators is presented, as they are the focus of our fault injection campaign.

3.1.1 Systolic DNN Accelerator Architectures

Systolic architectures reduce memory transfers by locally reusing data. They consist of a fixed size array of PEs which perform specific operations. In the case of DNNs, the PEs typically perform MAC operations and transfer data to their direct neighbors. The calculations of the abstract network are mapped to the PE array. Energy efficiency is the key concern and this is achieved by minimizing the number of external memory accesses, as an access to off-chip Dynamic Random Access Memory typically costs orders of magnitude more energy than to an on-chip SRAM.

In [Sze2017], three broad spatial architectures are identified : *weight*, *output* and *row stationary*. They rely on re-use of either weights, activations or both. We briefly present these architectures, however, the reader is referred to [Sze2020] for details on the implementations.

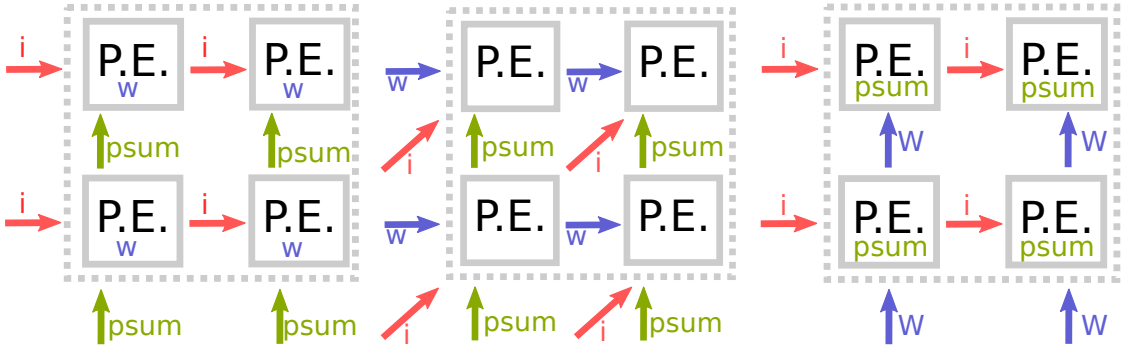


Figure 3.2: Data-flow in Systolic DNN Accelerator Architectures

a) Weight Stationary Data-flow

The Weight Stationary (WS) data-flow is illustrated in Fig.3.2a. Within a WS data-flow (such as [Park2015]), each column contains weights for a given output channel and performs the calculations for this channel. Weights are pre-loaded into the PEs and remain stationary. Activations flow horizontally and partial sums flow upwards. After several cycles, the sums arrive at the top of the column.

b) Row Stationary Data-flow

The Row Stationary (RS) data-flow is illustrated in Fig.3.2b. In a RS architecture (such as Eyeriss [Chen2016]), 2-D convolutions are broken into 1-D convolutions, which are processed in a single PE. PEs store multiple weights for the same row and perform simultaneous MAC operations but only need storage of one partial sum. This architecture achieves re-use of both weights and partial sums. We consider a PE which stores 4 weights.

c) Output Stationary Data-flow

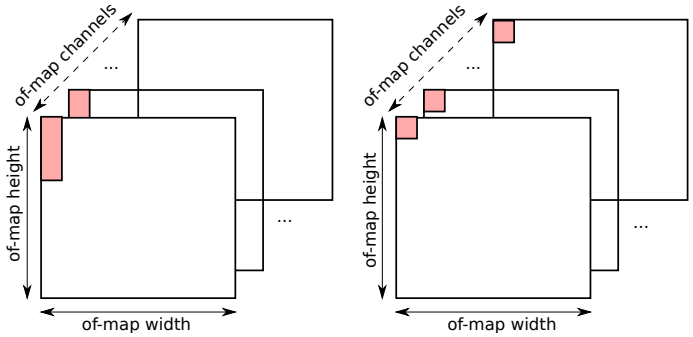
The Output Stationary (OS) data-flow is illustrated in Fig.3.2a. Alternatively, in an OS architecture (such as Shidiannao [Du2015]), the partial sums stay fixed in the PE. The columns share weights, and rows share input features. Each PE is dedicated to a single output pixel. Weights and input

features flow between the PEs. Detail on the scheduling of this implementation will be seen in the following section.

d) Mapping of PEs on OS data-flow

As discussed in [Chen2016], in an OS architecture, there are different ways to map PEs to the output feature map channels and pixels. In a Multiple Output Channel-Multiple Output Pixel (MOC-MOP) mapping, the PE array operates on multiple output channels at a time and the PE array is scanned across the input feature map. This approach is well adapted for convolutional layers with a large number of channels because the input features are re-used for multiple channels. In a Multiple Output Channel-Single Output Pixel (MOC-SOP) mapping, different input images from the same batch are simultaneously processed in different rows of the array. In other words, each row process a different image of the input batch. It makes the MOC-SOP adapted to fully convolutional layers.

In our analysis, we have selected the MOC-MOP mapping for convolutional layers with large kernels. For the fully connected layers in VGG-16 and in the depth-wise (1x1 kernel) layers in MobileNet, due to the fact that there is a single output channel, or a single weight in the kernel, the MOC-MOP is not efficient, thus we used the MOC-SOP mapping for these layers.



(a) MOC-MOP *(b) MOC-SOP*

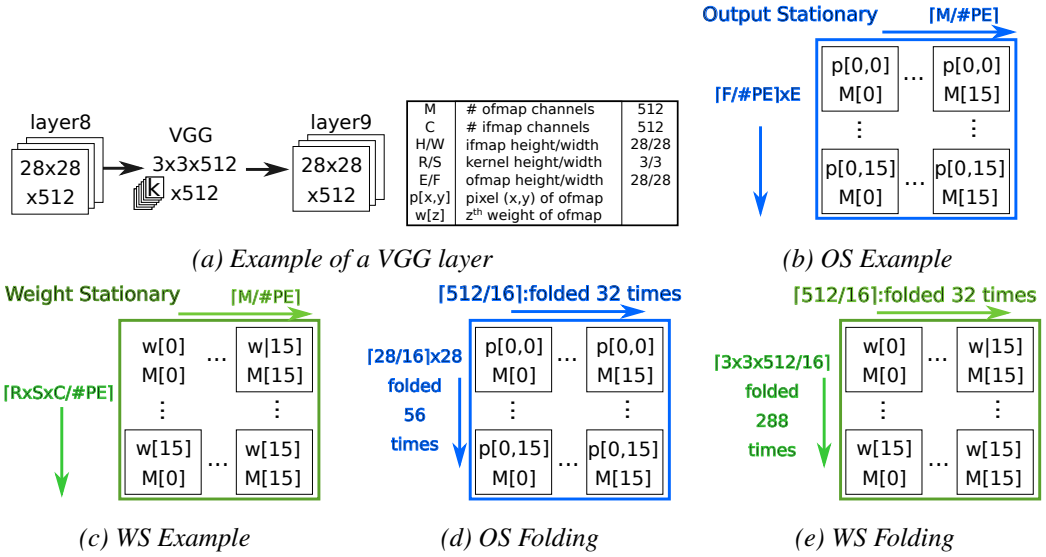
Figure 3.3: Two Variants of the OS Architecture

3.1.2 Folding

Generally, the PE array is too small to simultaneously compute all the operations in a given layer. The layer must then be *folded* into smaller parts that fit and can be computed sequentially in the array. The *folding* depends on the architecture. For all architectures under study, each column is dedicated to a specific output channel. In the OS architecture, the rows of PEs correspond to adjacent pixels of the same row of an output channel. For the WS and the RS architectures, the rows correspond to the weights of the same input channel. In all architectures, when PEs have executed all of their MAC operations, the whole array is reloaded with the next folded part of the layer. After a certain number of iterations, the layer is fully computed. The size of the input image (OS) or the size of the kernel (WS/RS) may not be a multiple of the number of rows, thus at the end, some rows of the array will remain unused. Similarly, if the number of output channels is not a multiple of the number of columns of PEs, some columns will be idle.

Let us consider the case of the computation of the 9^{th} layer of VGG-16 being executed on a 16x16 PE array (16x(4x4) for RS architectures). The mapping of this example on a PE array for OS and WS architectures is shown in Fig.3.4 (RS architecture is not presented for readability).

At this layer there are 512 input channels of 28×28, the kernels are 3x3 and there are 512 output channels also of dimension 28×28. Since the number of output channels (512) is a multiple of the number of columns, this layer must be folded horizontally 32 times ($= 512 \div 16$).



(a) Example of a VGG layer *(b) OS Example*

(c) WS Example *(d) OS Folding* *(e) WS Folding*

Figure 3.4: PE Mapping Depending on Architecture

For the OS architecture, only one row of the *of-maps* can be computed at a time. Since the *of-map* is 28 pixels wide, then the PE array must be folded twice, and 12 rows of PEs are idle during the second pass. This must be repeated 28 times, once for each of the *of-map* rows. Thus, for the OS architecture, the layer must be folded 56 times vertically.

For the WS architecture, each PE row is dedicated to a certain weight of a shared input channel. As kernels are 3x3, with 512 input channels, each row must process a total of $4608 = 512 \times 9$ weights. Thus, in this case, the layer is folded 288 ($= 4608 \div 16$) times vertically.

To summarize, in this example, to compute this layer (9^{th} layer) in VGG-16, the array will be reloaded 1792 ($= 32 \times 56$) times with the OS architecture, and 9216 ($= 288 \times 32$) times in the WS architecture. It is important to note that the total number of MAC operations to be computed for a layer is independent of the architecture. It is the number of times that the array must be reloaded and the operations mapped to each PE that changes.

The number of times a layer is folded obviously depends on the size of the PE array. To understand the impact of folding, we studied two PE array sizes (16x16 and 64x64).

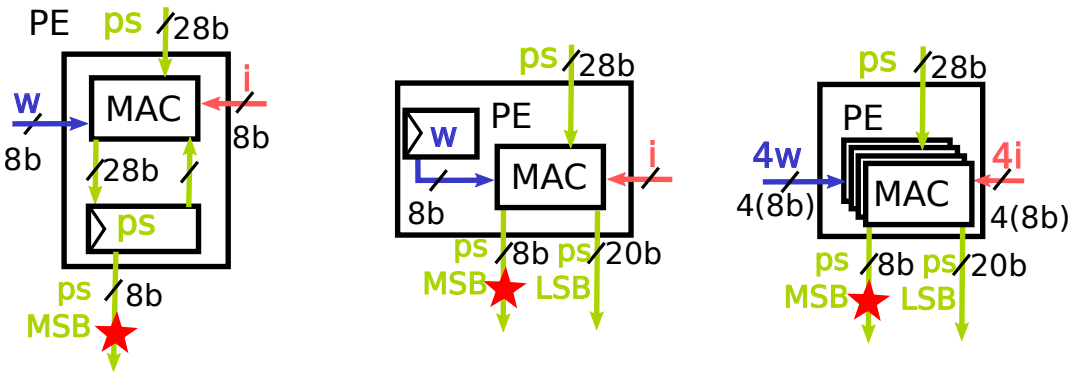
3.2 Fault Injection Methodology

3.2.1 Hardware Fault Abstraction

This chapter considers computational faults in the PE of the tested architectures. To simulate permanents errors in the data-path of a PE, we use a high-level fault-model illustrated in Fig.3.5.

Our focus is at the architectural level, thus we abstract the various technology dependent permanent faults (i.e. defects that manifest as single and multiple stuck-at faults, bridging faults) as single and multiple stuck-at logic errors.

We inject stuck-at faults on 1 to 8 bits at the output of the tested PEs and these faults are injected using a mask. We do not claim that this model reflects all possible logic errors occurring in PEs. However, we believe this fault model is sufficient to compare the effect of fault propagation in these architectures, and thus to draw conclusions on the relative fault tolerance of these architectures.



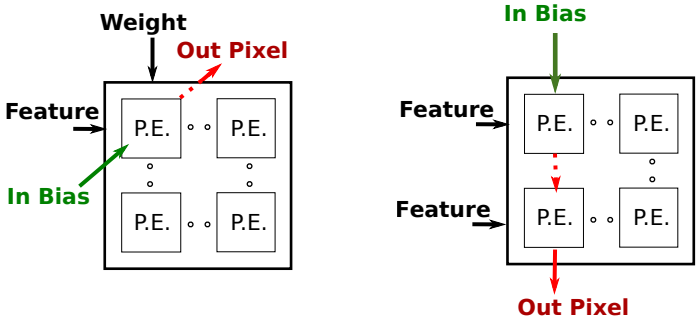
(a) Output Stationary *(b) Weight Stationary* *(c) Row Stationary*

Figure 3.5: PE Fault Models in OS, WS and RS Architectures

For the WS and RS architectures, we only inject faults in the 8 Most Significant Bits (MSBs) of the partial sum, because the 20 Least Significant Bit (LSB)s are inherently discarded in the data-path of this architecture. A fault in the LSBs could carry into the MSBs, however, to be able to fairly compare results, we wanted to have the same number of attacked bits (8), in all our experiments.

3.2.2 Impact of Computational Errors in Systolic Architectures

Due to the architectural differences illustrated in Fig.3.6, we expect the impact of a fault in the data-path of a systolic architecture to differ significantly from one architecture to another.



(a) OS Architecture *(b) WS and RS Architecture*

Figure 3.6: Data Flow Through the PEs for One Output Pixel

Every times the array is loaded in an OS architecture, each PE computes the value of a single output pixel for mapped to that PE. As a consequence, a fault in the data-path of a PE will impact one single pixel, but the pixel value is expected to be numerically, highly impacted by the fault as all the computations could be erroneous.

In the WS and RS architectures, the impact will be different. Each PE focuses on a weight (WS) or row of weights (RS) which are used on every pixels of one output channel. Thus a faulty PE is expected to affect a limited number of MACs for every pixels of an output channel.

Due to the *folding*, these impacts are repeated with different pattern on each architecture, as illustrated in Fig.3.7 As an example, for a PE array of size N*N, one single faulty PE will affect :

- For WS and RS architectures, all the pixels of 1/N output channels.
- For OS architecture, all the pixels of 1/N rows of 1/N column or 1/N input images, while other pixels will not be affected.

Theses patterns depend on the scheduling policies seen in Sec.3.1.1. For the purpose of readability, we present only a limited number of scheduling policies, but we observed that this parameter do not significantly affect the fault tolerance of the tested architectures.

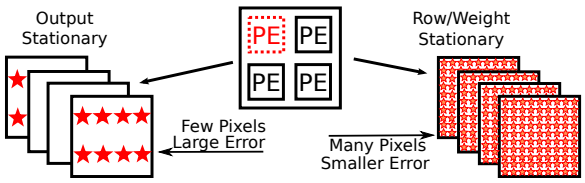


Figure 3.7: Propagation of a Fault in a PE in Different Architectures

3.2.3 Analyzing the Errors Produced by Faults

We performed two fault injection campaigns to assess the robustness of the tested architectures. We used LeNet-5, VGG-16, MobileNet and SqueezeNet models for DNNs, trained for MNIST for LeNet-5 and Imagenet data-set [Russakovsky2015] for other networks. We tested the RS, OS and WS architectures on two sizes of PE array (16x16 and 64x64) for analyzing the effect of *folding* on fault tolerance.

For each fault injection, an experiment consists of randomly selecting :

- One or multiple PEs in the array
- The target bit position(s)
- The stuck-at logic to apply (0 or 1).

After running a batch of images, the fault is removed and another fault injection is performed.

	1st Fault Injection Campaign	2nd Fault Injection Campaign
Selected DNNs	LeNet-5, VGG-16, SqueezeNet	VGG-16, MobileNet
N_{batch_size}	100	100
N_{batch}	1000	5000
Tested architectures	OS, WS and RS	OS and WS
Tested PE Array sizes	16x16 and 64x64	16x16 and 64x64
Number of stuck-at bits	1, 2, 4 or 8	1, 2 and 4
Data-type	8 bit quantized	8 bit quantized

Table 3.1: Parameters for each Fault Injection Campaign

Multiple batches (N_{batch}) are then evaluated, where each batch consists of N_{batch_size} images. For each individual batch, we record the accuracy. After all the batches are completed, we evaluate the *average* accuracy and the *worst case* accuracy. Parameters vary across fault injection campaigns and are presented in Tab.3.1 The pseudo-code is presented in Algorithm 1.

The choice of N_{batch_size} is important to obtain meaningful results for the observed *worst case* accuracy. Even without fault injection, the accuracy of a DNN is never perfect and the *worst case* accuracy depends on N_{batch_size}. To assess whether an observed low accuracy is the effect of the injected faults, and not of an unfortunate selection of images in the current batch, we performed a set of preliminary experiments without faults to record the *worst case* accuracy depending on the N_{batch_size}. The results are plotted in Fig.3.8. For our fault injection campaigns, we set the N_{batch_size} at 100 so that a *worst case* accuracy degraded by more than 10% compared to *average* accuracy can safely be concluded to be the result of injected faults.

Algorithm 1 Fault Injection Procedure

```
for all architecture do
  for all PE Array Size do
    for all DNN do
      for i from 1 to N_batches do
        Fault ← (random(PE, bits, value))
        Inject Fault
        Batch ← N_batch_size random test images
        Classify Batch
        Record accuracy
        Clear Fault
      end for
      Evaluate Average and Worst Case Accuracy
    end for
  end for
end for
```

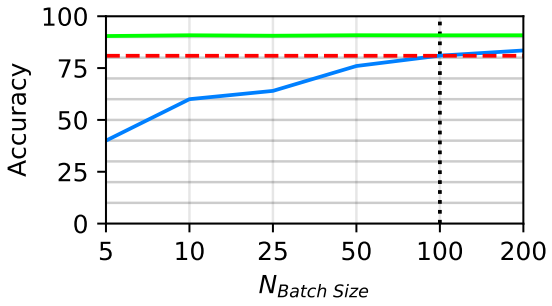


Figure 3.8: VGG-16 Worst Accuracy versus Batch Size - No Faults
Green line is fault-free *average* accuracy. Blue line is the observed fault-free *worst case* accuracy. The red-line is the fault-free *average* accuracy minus 10%. Black dotted line represents the N_{batch_size} chosen for our fault injection experiments.

3.3 Fault Injection Results

3.3.1 Experimental Framework

N2D2 is a C++ based DNN framework that integrates database construction, data pre-processing, network building, bench-marking, quantization and hardware export to various targets. Export targets include central processing unit running C/C++ code, Digital Signal Processor and GPU with Open MP, Open CL, Cuda, CuDNN and TensorRT programming models as well as code generation for custom hardware [Bichler2017]. We have chosen N2D2 for this study, as it has support for quantization and the generated code can be modified for injecting faults on simulated DNNs.

We selected four different networks summarized in Tab.3.2.

- **VGG-16** [Simonyan2014] is representative of heavy, redundant networks used in data-centers and achieve the best accuracy, of the tested networks, at the cost of an important number of MAC operations and weights.
- **MobileNet** [Howard2017] and **SqueezeNet** [Iandola2016] are representative of light-weight networks used in embedded applications. They are less accurate than VGG-16, but rely on 5 to 3 times less MAC operations and 115 to 33 times less weights.

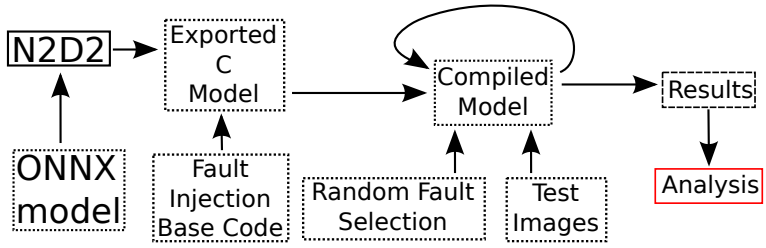


Figure 3.9: Fault Injection Methodology

Network	Num. Neurons	Num. MACs	Num. Weights	Data-set	Accuracy Top-1	Accuracy Top-5
LeNet-5	6518	341 K	60 K	MNIST	99%	-
SqueezeNet	2.6 M	352 M	1.2 M	ImageNet	55%	75%
MobileNetV1	4.7 M	569 M	4.2 M	ImageNet	69%	88%
VGG-16	13.6 M	15 G	138 M	ImageNet	70%	91%

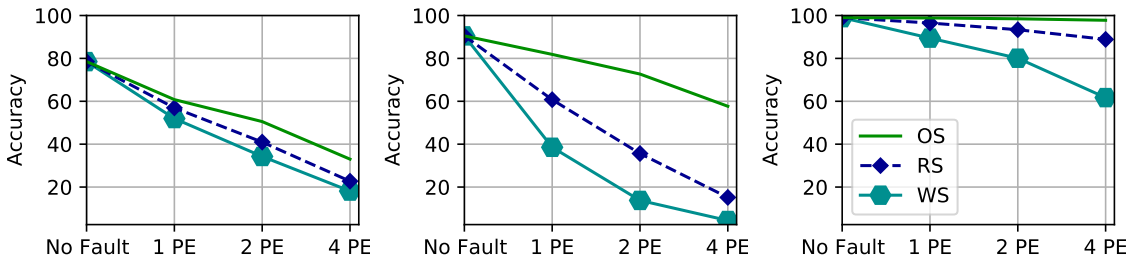
Table 3.2: Characteristics and Accuracy of Selected Networks

- **LeNet-5** was used in order to compare with other studies as it is one of the most used DNN in the literature. It achieves a high accuracy of 99% but can not be directly compared to other DNN as their data-sets are different.

Except for LeNet-5, all of the weights were obtained from the ONNX open repository [ONNX]. The tool flow is shown in figure 3.9. The trained models were quantized to 8-bits [Nagel2019].

3.3.2 Impact of the Architecture on the Robustness of Accelerator to Computational Faults

During our fault injection campaigns, we limited the fault injection scenarios from one to four PEs, as beyond this number, the drop in accuracy is unacceptable. For each data point on the graphs, 1000 randomly selected PEs and bit positions were selected and then 100 randomly selected images were analyzed in order to evaluate the top-5 classification accuracy (top-1 for LeNet-5). The results are summarized in Figure 3.10 and we see that in all cases, the OS architecture has a higher accuracy. Note, when not stated otherwise, we present results for an array of 16x16 PEs.



(a) SqueezeNet *(b) VGG-16* *(c) LeNet-5*

Figure 3.10: Average Accuracy of Three Systolic Architectures (OS, RS, WS) in the Presence of a Single Faulty Bit per PE

3.3.3 Impact of PE Array Size on Robustness

To study the impact of the size of PE array on the robustness, we performed a second fault injection campaign. For readability, we only present results for WS and OS architectures as the robustness of RS is comparable to WS. In Tab.3.3, we present the *average* and the *worst case* accuracy for two array sizes where faults were injected in a single PE using the fault model described in Sec. 3.2.1.

Architecture	Number of stuck-at bits	VGG			MobileNet		
		Worst-Case	Avg	Max	Worst-Case	Avg	Max
	0	86	91	95	73	78	85
OS(16x16 PEs)	1	3.5	81	93	0	65	82
OS(16x16 PEs)	2	0	70	93	0	55	82
OS(16x16 PEs)	4	0	59	94	0	36	80
OS(16x16 PEs)	8	0	47	94	0	29	80
OS(64x64 PEs)	1	83	91	93	30	75	82
OS(64x64 PEs)	2	41	89	94	2	72	82
OS(64x64 PEs)	4	36	87	92	2	65	80
OS(64x64 PEs)	8	20	81	93	1	58	82
WS(16x16 PEs)	1	0	40	89	0	12	63
WS(16x16 PEs)	2	0	40	83	0	2	32
WS(16x16 PEs)	4	0	38	79	0	1	4
WS(16x16 PEs)	8	0	31	78	0	1	4
WS(64x64 PEs)	1	0	51	92	0	43	82
WS(64x64 PEs)	2	0	51	91	0	31	77
WS(64x64 PEs)	4	0	43	90	0	21	64
WS(64x64 PEs)	8	0	48	91	0	14	64

Table 3.3: Accuracy with Random Stuck-At-0 and Stuck-At-1 Faults in a Single PE

As illustrated in the Tab.3.3, we observe that as the size of the PE array increases, the impact of a fault decreases. In a smaller array, more *folding* generates an increased reuse of a faulty PE. We also observe that the OS architecture is more robust, to the extent that faults have a limited impact in a large array as showed on the row OS(64x64 PEs) for 1 such-at bit, where *average* accuracy does not reduce the VGG16 accuracy and reduces the accuracy of MobileNet by 3 percentage points. Conversely, on the similar row with the WS architecture, the *average* accuracy is decreased by 51 percentage points for VGG16 and by 66 percentage points for MobileNet.

A notable observation is that the number of stuck at bits at the PE output does not have a major impact on the reduction in accuracy. This is due to the compact 8-bit numeric format. The *worst case* numerical impact of a fault is limited. More important than the magnitude of the impact is how many output pixels in the calculation are corrupted.

The fact the OS architecture is more tolerant to faults can be explained as each fault in the computational logic impacts a single activation in a subset of the channels, or a small number of activations, when the PE is re-used due to *folding*. In a WS architecture, a faulty PE corrupts all the pixels in the affected channels, significantly impacting the classification accuracy. We conclude that the spatial impact of a fault is more significant than the magnitude of the numerical corruption.

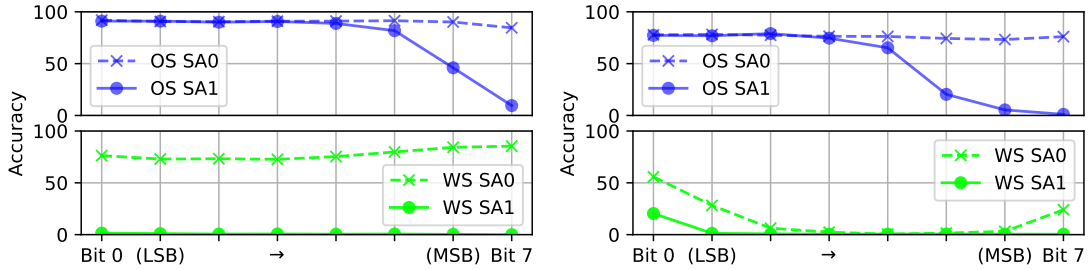
3.3.4 Bit-Level Analysis of the Effect of Faults

Up to this point, the results included both Stuck-At-Zero (SA0) and Stuck-At-One (SA1) faults. In figure 3.11, we separately reported results for both single bit SA0 and SA1 faults in random bits of the 8-bit word. As expected, based on previous works [Salami2018], SA0 faults have a much smaller effect.

With the OS architecture, any Stuck-At (SA) fault on the 4 LSBs has virtually no impact on classification accuracy. Considering only SA0 faults, the OS architecture is tolerant of faults in all bit positions. For the WS architecture, the difference in accuracy between SA0 and SA1 is striking. For VGG all SA1 faults trigger a catastrophic loss in *average* accuracy.

MobileNet remains quite sensitive to SA0 faults. Since MobileNet is highly compressed, any further loss in connections, has an immediate impact on accuracy.

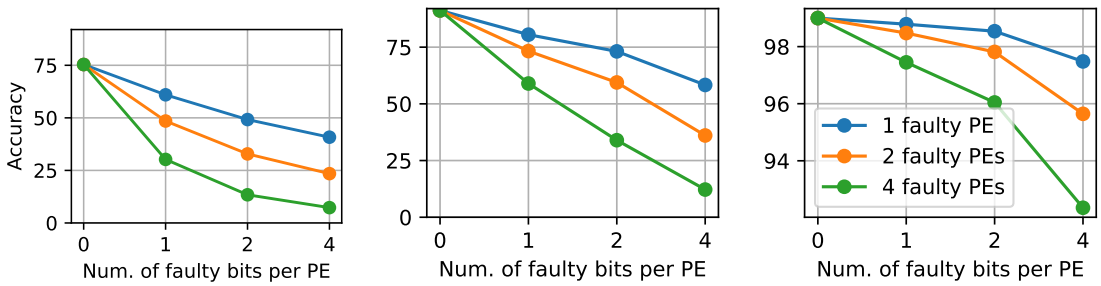
Previous studies have shown that SA0 faults have only a minor impact when applied to an abstract model of a DNN and we have shown this result extends to DNN accelerators. It is remarkably how tolerant the OS architecture is to SA0 faults, and this finding was our inspiration for the design of a fault tolerant DNN accelerator presented in chapter 4.



(a) VGG *(b) MobileNet*

Figure 3.11: Average Accuracy versus Faulty Bit Position

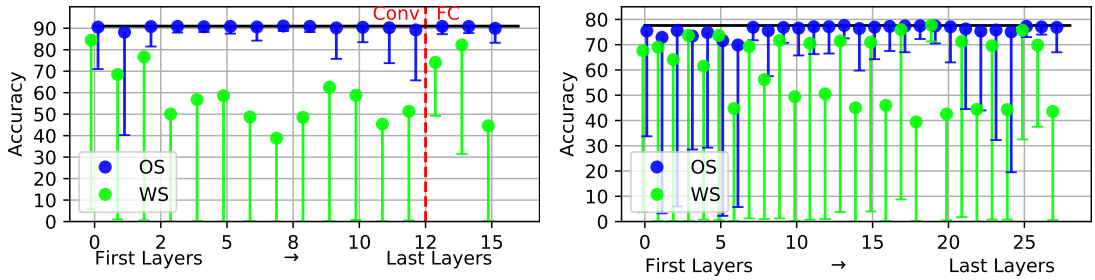
We also investigated the impact of the number of SA bits on accuracy for the OS architecture. The results are shown in Figure 3.12. Each graph has curves for the case of one, two or four faulty PEs. With VGG-16, with a single faulty PE, with four SA bit faults, the accuracy is ≈60%, with two faulty PEs, each with two SA faults, the accuracy is also ≈60% and with four faulty PEs with a single SA fault, the accuracy is still ≈60%. For the OS dataflow, it appears, the drop in accuracy depends on the total number of SA faults, regardless how they are distributed. As the trend in the drop of accuracy is similar, regardless of the number of SA bits, for the remainder of this chapter, we limit our fault model to a single SA bit per faulty PE.



(a) SqueezeNet *(b) VGG-16* *(c) LeNet-5*

Figure 3.12: Impact of Number of Faulty Bits on Three Networks (OS Architecture)

3.3.5 Impact of Faults by Layer



(a) VGG16 *(b) MobileNet*

Figure 3.13: Robustness of Various Layer on Tested DNNs
Top-5 Accuracy versus layer, where a single bit fault is applied to one PE in one single layer in a 16x16 PE array. The upper circles show *average* accuracy and lower bar shows the *worst case*. The black line shows fault-free accuracy.

For assessing the robustness of different layers, specific fault injections were performed layer by layer and the results are presented in Fig.3.13. The results for WS and RS architectures were comparable. For readability, we present only the results on OS and WS architecture. From this figure, we see that there is a significant gap between *average* accuracy (dots) and the *worst case* accuracy (lower bars).

It is clear that the OS architecture (in blue) is more robust. With this architecture, the variation in the *average* accuracy between layers is small.

For the WS architecture, with VGG-16, we see that the fully-connected layers are quite sensitive, particularly the last one. These observations are consistent with the literature as seen in Sec.2.2.1. For example in [Xu2019], where faults were injected in the weights, results are comparable with our experiments. Considering the *worst case* accuracy in the WS architecture, for each layer, there are some faults which cause the *worst case* accuracy to drop to zero - an important finding for safety applications.

In Fig.3.13b, when looking at the *average* accuracy in the WS architecture, we note an alternating pattern (layers 6 to 18). This is because MobileNet consists of two types of convolutional layers : traditional ones and depth-wise ones where the kernel is a single weight. As seen in figure 3.13b, with the WS architecture, the depth-wise layers are more sensitive. This can be explained because *all* the pixels in the given output channel are highly impacted. In other words the link from an input channel to an output channel is completely corrupted.

3.4 Comparison with State of the Art

To the best of our knowledge, there is no similar study that compares the robustness of multiple systolic DNN accelerators.

As we have seen in Sec.2.2.1, studies agree on the fact that faults on weights are more likely to decrease the accuracy of the DNN than faults on activations. If we consider the number of pixels impacted by these faults, we see that they are close to the fault models studied in this chapter (faults on weights will alter all the pixels of a given output channel like an arithmetic fault on WS architectures, while faults on activations will alter a limited number of neighboring pixels on the output channels like an arithmetic fault on OS architectures). Consequently, our conclusions are consistent with the analysis proposed by previous works, namely that fault with a larger spatial impact are more likely to alter the accuracy of a DNN.

Considering systolic accelerators, we note that they are a frequent hardware platform for fault injection studies [Li2017, Abdullah Hanif2020, Liu2021, Ozen2020a, Ozen2020c, Reagen2016a, Schorn2018, Wu2021b, Zhang2018a, Zhang2019]. However, none of these studies quantitatively compares the robustness of different architectures. Instead, they select one existing architecture (most frequently the WS architecture) and consider another aspect such as fault tolerant training, detection and mitigation of faults or the reduction of power consumption made possible with the robustness of DNNs.

3.5 Future Developments

This study on the robustness of systolic accelerators for DNNs is not exhaustive and can be improved in future works.

We have only focused on permanent faults in the data path of such accelerators. The study of other fault models would be of great scientific interest. For example, faults occurring in communications channels between PEs would have spatial impacts that are architecture dependent and, based on our findings, could be very significant. Faults occurring in unprotected control logic could possibly modify the behavior for a large number of computations and thus must be considered when designing an accelerator for mission-critical industrial applications.

We only focused on systolic accelerators, however it is difficult to fully utilize a PE array without increasing the batch size, which complicates their use in real-time applications. The key observation in our work is that the dataflow significantly impacts how faults propagate and we believe that these observations can be applied to future studies of other classes of DNN accelerators.

3.6 Conclusions of the Analysis of Fault Tolerance of Systolic Accelerators

We have presented a methodology for analyzing the impact of faults in a DNN accelerator. Our methodology, based on reporting the *worst case* and *average* accuracy for batches of images enabled us to show that certain faults can have a catastrophic impact on the classification accuracy. This is a key point when analyzing the fault tolerance of DNNs for safety applications. By simply reporting an *average* accuracy, the severity of specific faults is missed.

With the 8-bit numeric format, our results show that, in terms of the impact on loss of accuracy, the spatial scope of a fault is more important than its numerical impact. For example, in a 16x16 PE array running VGG-16 with a single bit SA fault, the *average* accuracy of an OS architecture is reduced to 81% versus 61% for RS and 39% for WS. Thus, the OS architecture, where data-path errors impact a small number of neurons are attractive for fault tolerance.

We used four different networks for this study, VGG-16 which has a huge number of weights and significant redundancy, SqueezeNet and MobileNet, which are more compact, and LeNet-5 which is frequently used in the literature. Not surprisingly, compressed networks are less tolerant to faults, as they have less inherent redundancy. For example, with a 64x64 PE array with an OS architecture, in the presence of a single SA fault, the *worst case* accuracy of VGG-16 is 83% (3% loss) versus 30% (43% loss) for MobileNet. This key result must be considered if highly compact networks are deployed in critical applications.

It is important to note that in our detailed analysis of the impact of the fault multiplicity and the position of the affected bits, in *all* cases, the OS architecture was more robust than the WS. This important observation guided us in the design of the fault tolerant accelerator presented in the following chapter.

4

Design of a Fault Tolerant Systolic Accelerator

In this chapter, we present MOZART+, a Deep Neural Network accelerator architecture which provides fault detection and fault tolerance. MOZART+ is a systolic architecture based on the Output Stationary (Output Stationary) data-flow that inherently limits fault propagation. In addition, MOZART+ achieves fault detection with on-line functional testing of the Processing Elements (Processing Elements). Faulty Processing Elements are swiftly taken off-line with minimal classification impact. We show how to handle the case of layers with a small number of neurons which require special attention. The implementation of our approach on SqueezeNet results in a loss of accuracy of less than 3% in the presence of a single faulty Processing Element, compared to 15-33% without fault mitigation. The area overhead for the test logic does not exceed 8%. Moreover, dropout during training further improves fault tolerance, without a priori knowledge of the faults.

As seen in the literature described in chapter.2, many types of Deep Neural Network (DNN) accelerators exist such as Graphics Processing Units, Field-Programmable Gate Arrays and systolic accelerators. The main focus of the current design of such accelerators is the optimization for performance and energy efficiency [Reuther2019, Reuther2021]. However, when used in safety critical applications, the target accelerator must be tolerant to hardware faults. Architectural fault-tolerance can achieve robustness with minimal overhead.

Integrated circuits used in automotive applications must comply with ISO-26262 [ISO-26262.18] and need to detect and react to faults in an interval that is shorter than the *fault tolerant time interval*, the maximum time the fault can be present without posing a safety risk. As on-line testing becomes mandatory, new techniques must be developed that have lower overhead than traditional on-line testing. Standards also impose stringent requirements on overall Failure In Time (FIT) rate (e.g. $\leq$10 FIT for Automotive Safety Integrity Level D) which are difficult to achieve for large DNN circuits [Li2017]. State-of-the-art automotive DNN accelerators employ Dual Modular redundancy [Talpes2020, Matsubara2021], but this strategy is costly and does not exploit the specificities of DNNs.

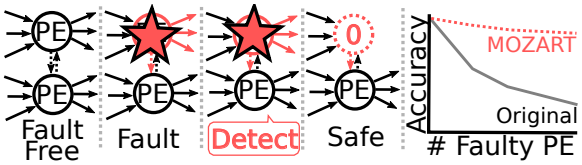


Figure 4.1: MOZART+ Provides On-Line Testing and Fault Masking with Graceful Degradation in Accuracy

In this chapter, we propose an architecture called MOZART+, which is a systolic based DNN accelerator specifically designed to achieve fault tolerance. MOZART+ relies on five points :

- It is built upon a Output Stationary systolic dataflow, which was shown in chapter 3 to be the most robust systolic architecture.
- Additional hardware components in the Processing Element (PE)s perform on-line fault detection of faults occurring in the data-path.
- A fault mitigation technique inspired by the literature that provides graceful degradation of the classification accuracy for the tested applications.
- An innovative fault tolerant training to make the DNN robust to permanent faults without any a priori knowledge of actual faults.
- A fault tolerant scheduling inspired by the literature to further improve the robustness of the most sensitive layers of the DNN.

This chapter is organized as follows : Sec.4.1 presents MOZART+, the fault tolerant systolic accelerator designed during this thesis. Sec.4.2 presents the methodology of the fault injection experiments used to evaluate the ability of MOZART+ to detect and mitigate permanent faults. Sec.4.3 presents our experimental results. Sec.4.4 compares our architecture with the closest works in the scientific literature. Sec.4.5 present ideas to improve MOZART+. Finally, Sec.4.6 concludes this chapter.

4.1 MOZART+ Architecture

MOZART+ is a systolic OS based DNN accelerator with data-path fault detection and mitigation. We have seen in chapter 3 that the data-flow in a DNN accelerator influences how faults propagate and thus has a significant impact on the fault tolerance.

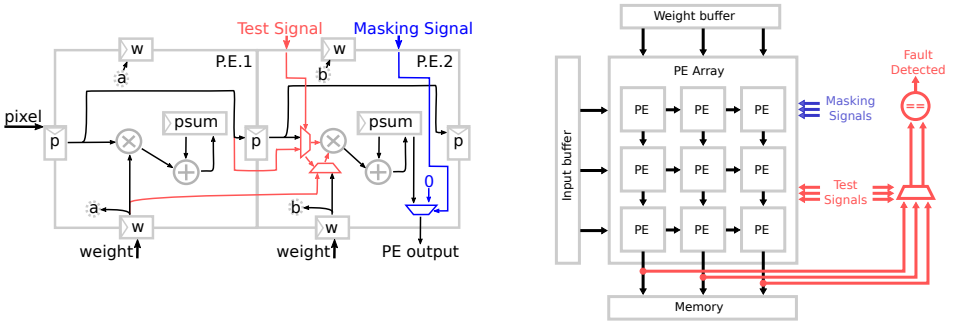
The literature seen in Sec.2.2.1 concludes that in DNNs, faults on weights are more likely to provoke failures than faults on activations. The conclusion of the previous chapter is that the OS data-flow is the most robust of the tested systolic architectures because it limits the spatial propagation of a fault occurring in the data-path. As a consequence, MOZART+ is built upon the OS systolic architecture.

4.1.1 Fault Detection and Mitigation with MOZART+

MOZART+ integrates an on-line functional testing technique which ensures high coverage for impacting faults. The online fault detection procedure principle is based on the fact that during the testing procedure, each PE is taken off-line, one at a time, by setting its output to zero. This exploits the fact that when a single PE is forced to zero, the impact on classification accuracy is negligible [Zhang2019, Reagen2016b]. This enables deterministic scheduling of the testing of every PEs.

Rather than relying on logic Built-In Self Test to generate random inputs, we propose an approach based on using the application data as stimulus for on-line testing. Fig. 4.2 shows the principle and added circuitry (fault detection in red and fault masking in blue).

More in details, Fig. 4.2a shows the details of the PE used in MOZART+. Three multiplexers are added. Two are controlled by the test signal and switch the inputs (weight and activation) of the multiplier of the PE from the local input to the input of its neighbor The last multiplexer is used to mask the output of one PE to zero when the PE is set as faulty given its assigned masking signal. Fig. 4.2b shows MOZART+'s PE array. After computation of a partial sum, an external comparator checks the result of the PE under test with that of its neighbor. In case of faults, if the outputs don't match, the PE under test is taken off-line, meaning that its external outputs remain set to zero. Once a PE is tested, the next PE starts the test procedure. This fine-grained scheduling minimizes detection time, quickly detecting the highly impacting faults and has no impact on the latency of the calculation.



(a) PE *(b) PE Array*

Figure 4.2: Schematic of MOZART+

A transient fault in a PE could also result in a mis-match and thus needlessly force a given PE off-line. This is prevented by including all PEs in the periodic test procedure, even those that are off-line. If an off-line PE shows no further errors after multiple test iterations, it is then taken back on-line, ensuring that transient faults don't result in an unnecessary loss of PE resources.

We synthesized a 16x16 PE array, with 8-bit integer multipliers, 32 bit adders, and registers for the partial sums. Synthesis results show that the overhead for the extra logic required for fault detection and mitigation is only 8%.

4.1.2 Unaware Fault-Tolerant Training for Improving Tolerance to Hard Errors

Fault tolerant training, as presented in Sec.2.3.2, can improve the robustness of DNNs by considering hardware faults during training. The scientific literature covers both training techniques that have knowledge of the actual faults and training techniques that lack knowledge of the faults that will occur.

Studies that use training to improve the robustness of DNNs to permanent faults rely on fault aware retraining [Zhang2019, Abdullah Hanif2020]. We believe that this approach is poorly adapted to industrial applications, as it is not possible to perform custom training for each instance of a component.

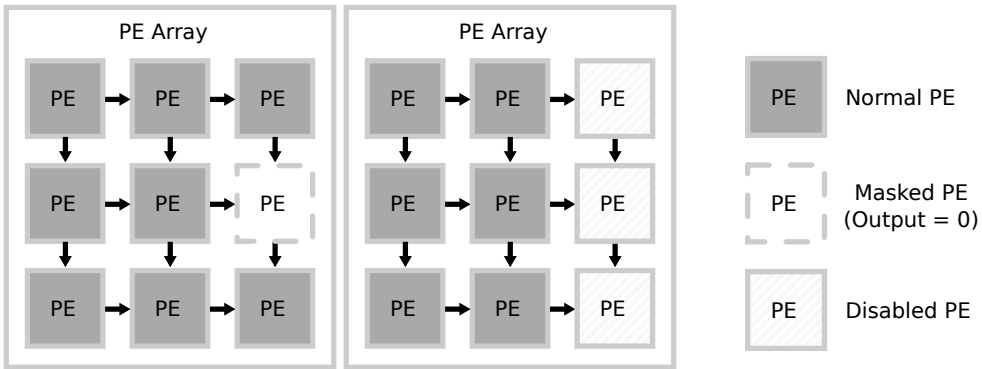
In an OS based architecture, an abstract neuron is mapped to a single PE. Due to the online test mechanisms, once a fault is detected, the faulty PE output is masked to zero which corresponds to disconnecting multiple neurons from the abstract network, similar to dropout. *By design*, the hardware architecture of MOZART+ is suited to exploit dropout during training to improve fault tolerance.

Based on the fact that the same PEs are used to compute all layers, our proposal is to apply random dropout to all layers which is different to standard dropout which is only applied to the last layers. In this way, the training can improve the robustness of the model without a priori knowledge of the location of faults. In the absence of fault, it does not result in a degradation of accuracy but does increase the training time.

4.1.3 Fault Tolerant Scheduling for the Last Layer

The MOZART+ technique was prototyped and the initial results identified a problem in how faults in the last layers were handled. If a neuron in the last layer is masked to zero, then an entire output class is lost, which does have an impact on accuracy, particularly for data-sets with a small number of classes. We thus integrated an enhancement.

In systolic architectures, the layers are processed sequentially by the PE array. When processing fully-connected layers, in a systolic architecture, it is necessary to use batching to fully utilize the PE matrix [Sze2017]. Multiple images are processed simultaneously, with each line processing a different image. To avoid the problem of losing an output class, we propose an alternate and simple technique for the last layer. After one (or several) faulty PE(s) are detected, when processing the last layer, we propose to reduce the batch size. The images in the reduced size batch are mapped to columns that contain no faulty PEs, as illustrated in Fig.4.3. With this approach, in the worst case, we can handle up to N/2 faulty PEs where N is the number of columns of the PE array. The overhead is limited as on modern DNNs, the last layer is computationally much less expensive than other ones.



(a) Normal behavior *(b) Degraded accelerator for last layer* *(c) Legend*

Figure 4.3: Fault Tolerant Scheduling for the Last Layer

4.2 Mozart Evaluation

To evaluate the MOZART+ architecture, we performed a fault-injection study on three different DNNs. To confirm the superior robustness of the Output Stationary (OS) architecture, we also simulated equivalent on-line testing and fault-masking strategies on Weight Stationary (WS) and Row Stationary (RS) architectures.

4.2.1 Applications and Data-Sets

We choose three networks for our experimental study that were previously presented in Sec.2.1.1. 1) SqueezeNet [Iandola2016] represents a compact network that could be used in embedded systems. 2) VGG-16 [Simonyan2014] is a computationally intensive network and represents cloud-based applications. 3) LeNet-5 is a lighter network that has been chosen to compare our results with the literature. As other authors [Arechiga2018, Solovyev2019, Zhang2019] have studied VGG-16 and LeNet-5, we are able to compare our results to similar works in the scientific literature. .

These networks are summarized in Tab. 4.1. This Table summarize, for each DNN, the number of neurons, weights and Multiply And Accumulate operations used to execute one image. The data-sets are MNIST for LeNet-5 and ImageNet for the other DNNs. They are both image classification data-sets. MNIST is composed of 60 K small images (28x28 pixels) of 10 handwritten digits. ImageNet is more complex as it is composed of 1.2 M of pictures of variable size which can be classified into 1000 different categories. Tab. 4.1 reports the Top-1 and Top-5 accuracy of these networks. In Top-1 accuracy only the neuron with the highest activation value is considered to be the prediction of the model. In Top-5 accuracy the prediction is considered to be correct if one of the five neurons with the highest activation values corresponds to the correct label.

For each of these networks an energy-efficient 8-bit integer format was chosen as this data representation has proven to be more robust than floating point, while maintaining a minimal loss in accuracy [dos Santos2019].

Network	Num. Neurons	Num. MACs	Num. Weights	Data-set	Accuracy	
					Top-1	Top-5
SqueezeNet	2.6 M	352 M	1.2 M	ImageNet	55%	75%
VGG-16	13.6 M	15 G	138 M	ImageNet	70%	91%
LeNet-5	6518	341 K	60 K	MNIST	99%	-

Table 4.1: Characteristics and Accuracy of Selected Networks

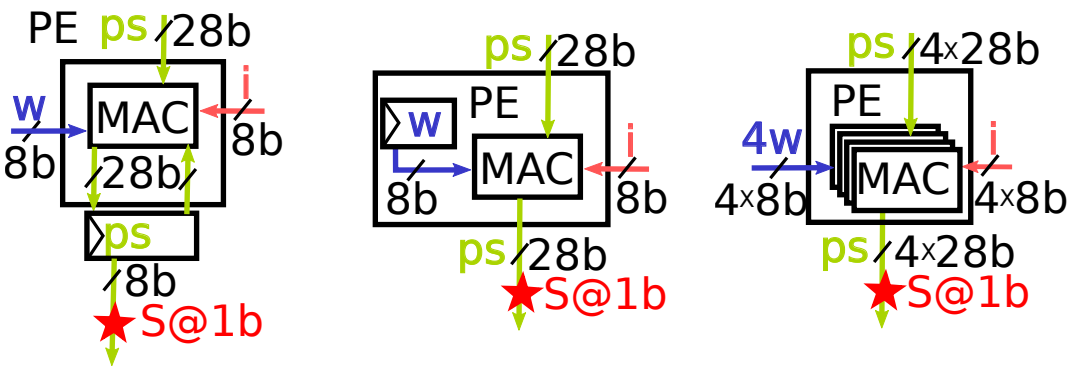
4.2.2 Hardware Fault Model

The MOZART+ architecture focuses on mitigating the computational errors occurring in the datapath of the PE. We define a fault model based on SA faults affecting the outputs of the PE, equivalent to the fault model used in chapter 3 but limited to a single bit stuck-at, as illustrated in Fig.4.4.

Our focus is on the accelerator architecture, thus we adopted a higher-level fault-model that clearly does not model all possible logic errors, but is sufficient to draw architectural conclusions as the fault propagation is the same, regardless of the underlying cause of the faults.

To be consistent and based on the observations made in previous chapter (see Sec.3.2.1), we inject faults in the 8 Most Significant Bit (MSB)s of the registers that store the results of the partial sums in both architectures.

To evaluate the efficiency of our fault detection technique, we voluntarily limit the number of stuck-at bit to a single bit. A single fault in the computational logic may affect several outputs bits, but if our technique can detect single bit faults, it can easily detect multi-bit faults, and we saw in Sec.3.3.4 that the impact of single and multi-bit faults is not that significant for these applications.



(a) Output Stationary *(b) Weight Stationary* *(c) Row Stationary*

Figure 4.4: PE Fault Models in OS, WS and RS Architectures

4.2.3 Fault Injection Methodology

The objective is to evaluate the fault detection and mitigation of our method. Again we consider the accuracy as a metric to evaluate the behavior of the tested DNN in the presence of a fault. For each tested condition, we record the *average* accuracy and the *worst-case* accuracy. As seen in Sec.2.2.1, certain single bit faults can cause the accuracy to drop close to zero.

To be consistent and to be able to compare the impact of our fault mitigation strategy with the previous chapter, we use the same N_{batch} (1000) and N_{batch_size} (100), and we used the same algorithm presented in chapter 3, however, we introduced an additional nested loop to iterate over between 1 a,d 4 faulty PEs.

Our methodology for choosing the N_{batch_size} has been described in chapter 3.

Algorithm 2 Fault Injection Procedure

```
for all For each architecture (OS, RS, WS) do
  for all For each PE Array Size (16x16, 64x64) do
    for all For each network (LeNet-5, SqueezeNet, VGG16) do
      for all For each number of faulty PE (1,2,4) do
        for i from 1 to N_batch do
          Fault ← (random(PE, bit, value))
          Inject Fault
          Batch ← N_batch_size random test images
          Classify Batch
          Record accuracy
          Clear Fault
        end for
        Evaluate Average and Worst Case Accuracy
      end for
    end for
  end for
end for
```

4.3 Fault Injection Results

In this section, we show how the MOZART+ techniques provide effective fault detection and masking. Sec. 4.3.1 shows the effectiveness of the proposed functional test strategy. Sec. 4.3.2 shows the behavior of the networks when PEs are masked to zero. Sec. 4.1.3 shows the efficiency of our scheduling strategy. Sec. 4.3.4 analyzes the efficiency of the dropout technique. Finally, Sec. 4.3.5 demonstrates the scalability of the approach.

4.3.1 Effectiveness of On-Line Test

A key element of the MOZART+ approach is on-line testing, which continuously checks the correct operation of the PEs Sequentially, the testing procedure disconnects the input (weights and activations) of a single PE and connect them to those of its neighbor, so they both perform same operation. Once the PE array has finished the computation of one set of activations (we call this a round of computations), the results of the PE under test and its neighbor are compared to detect a fault. Every times the PE array is loaded with a new set of the testing procedure, but we select the next PE for testing.

To evaluate the detection capability of this technique, we injected Stuck-At-Zero (SA0) and Stuck-At-One (SA1) faults on every output bits for each of the 256 PEs (4096 faults). For each individual fault, 100 randomly selected images were evaluated to check if the fault could be detected by comparing the partial sum of the PE with that of its neighbor. Table 4.2 shows the percentage of all SA1 faults detected after 1, 2, 4, 8 and 100 images.

Percentage of SA1 Faults	Number of Images (N)				
Detected After N Images	1	2	4	8	100
SqueezeNet	100%	100%	100%	100%	100%
VGG-16	100%	100%	100%	100%	100%
LeNet-5	94%	98%	99%	100%	100%

Table 4.2: Percentage of SA1 Faults Detected After N Images

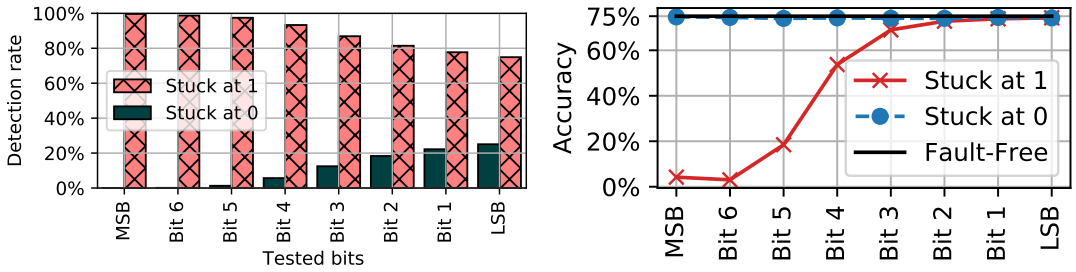
For the larger networks, a *single* image was sufficient to detect all the SA1 faults. This was true for *any* of the 100 random images we used as stimulus for VGG16 and SqueezeNet. For LeNet-5 with a single image, 94% of the SA1 faults could be detected, which is already a good level of coverage. In fact, for LeNet-5, more images are required because as it is a small network, the number of operations performed per image is small (see Table 4.1).

As we seen in Sec.3.3.4, SA1 faults have a higher impact than SA0 faults, particularly for the OS based dataflow, and these experiments show that the low-overhead approach to on-line test can quickly detect the highly impacting SA1 faults.

Detecting SA0 faults is more difficult because the neuron output values are often zero, especially the MSBs of the integer representation. In Figure 4.5a, for SqueezeNet, we show the fraction of Stuck-At (SA) faults that were detected after a single round of computation. SA1 faults are easily detected, especially in the MSBs. In contrast, SA0 faults in the MSBs are hard to detect due to application masking. This is not an important issue, as a single SA0 fault has a negligible impact on classification accuracy, as discussed in section 3.3.4. These results are consistent with [Reagen2016b, Reagen2018].

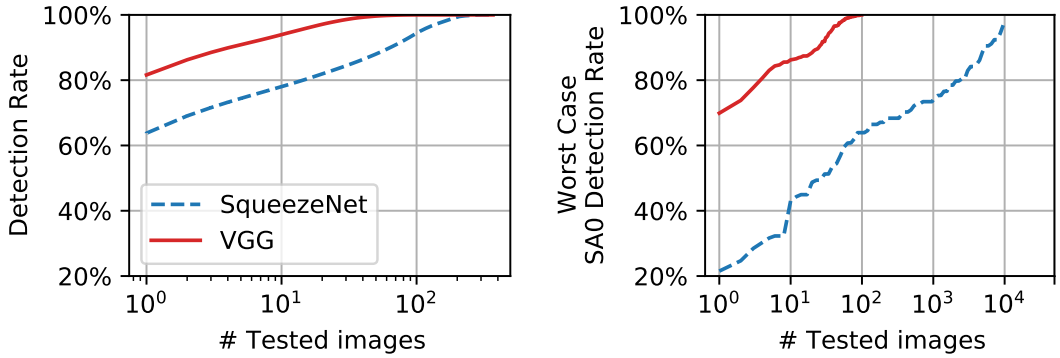
To understand if SA0 faults can be detected with functional test, we performed two further experiments, focusing on SqueezeNet and VGG. First, we considered the 2048 possible SA0 faults, and for each of these faults, we applied 100 randomly selected images. In Figure 4.6a, we report the percentage of these faults that are detected after a given number of images. We see that after

100 images, all SA0 faults were detected for VGG-16 and 94% are detected for SqueezeNet. This difference can be explained since in VGG-16 a typical PE is tested 215 times for every input images, versus only 42 for SqueezeNet. With LeNet-5, as the 16x16 PE array is under-utilized, the SA0 detection is poor and not presented.



(a) Fault Detection Rate after One PE Array Load *(b) Accuracy Impact of Single Bit Fault*

Figure 4.5: Detection Rate and Accuracy with Single Bit SA Faults on SqueezeNet (OS)



(a) Average *(b) Worst case*

Figure 4.6: Number of Images Required to Detect Latent SA0 Faults

To obtain a bound on the worst case number of images required to detect any SA0 faults in SqueezeNet, we performed a second experiment. For every possible SA0 faults, 10,000 images were evaluated. We ranked the images from the one which detected the fewest faults to the one that detected the most. In Figure 4.6b we present the cumulative detection rate, starting on the left, with the image with the lowest coverage. From this graph, we observe a worst case detection time for SA0 faults. After 10,000 images, we detect 97% of the SA0 faults, which is reasonable, given that these faults are benign.

With a more accurate fault model (eg. faults inside the arithmetic logic of the MAC), by relying on the incoming images as stimuli for the on-line testing, there is a risk that latent faults which may occur but will not be detected. If such faults exist, and if the current, real-world stimuli do not activate them, they are not of our immediate concern. If the stimulus changes, and the previously latent fault is activated, then it is quickly detected by our continuous on-line testing.

4.3.2 Fault Tolerance in MOZART Architecture

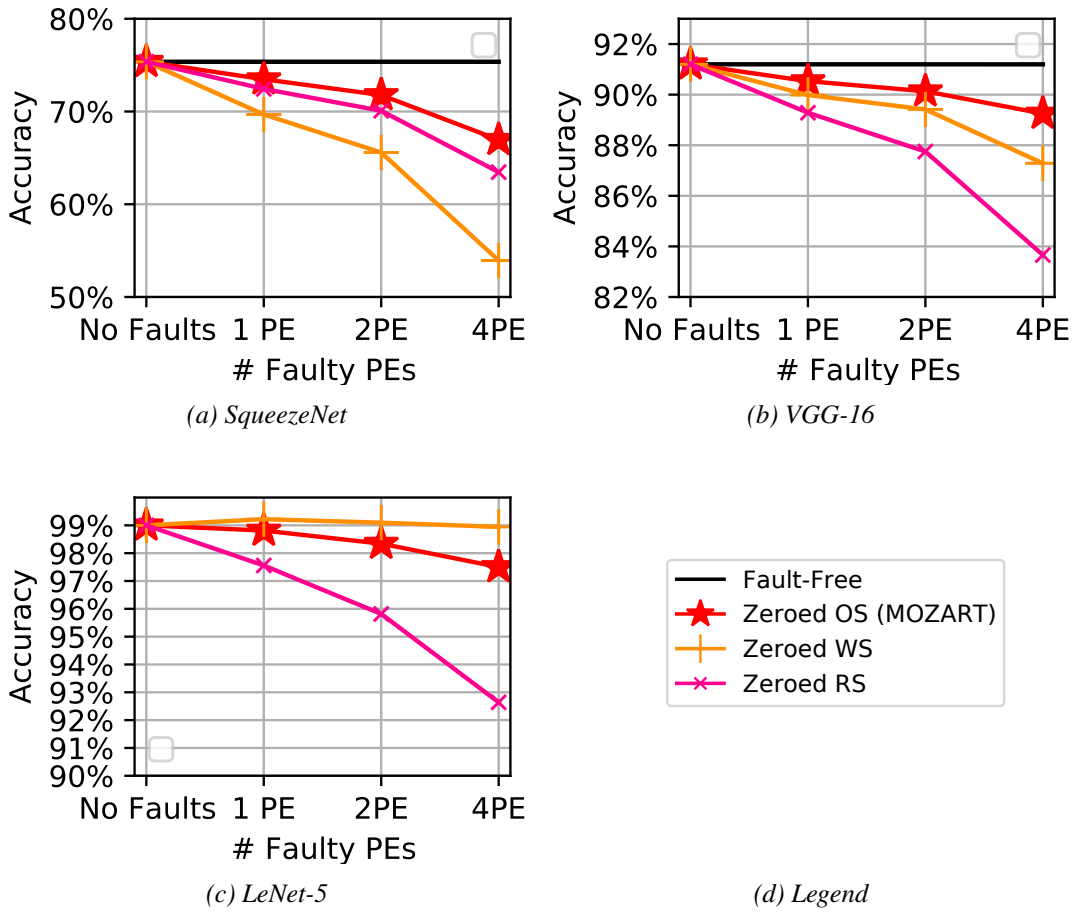
In the previous section, we have discussed the procedure for on-line testing. To ensure a fault tolerant design, after detecting a faulty PE, we suppress the potentially serious impact of a faulty PE by forcing its outputs to zero, removing it from subsequent computations.

a) Average Accuracy

In Figures 4.7a to 4.7c we show the *average* classification accuracy when the output of one, two or four PEs are masked to zero, for each of the three architectures. This shows how the accelerator would behave when one PE is taken off-line during the on-line testing, and how it would behave if additional PEs are taken off-line due to permanent faults.

A key point is that for these networks, the output masking technique is most effective for the OS based architecture (shown in red) and for SqueezeNet and VGG-16, the drop in accuracy when a single PE's output is masked is very small. As more PEs are masked, with the OS based architecture, the loss of accuracy is gradual with the number of PE being masked.

LeNet-5 has only 10 neurons in the output layer so if a PE that is used to compute an output neuron is masked, the impact on accuracy is very significant. In this case the OS architecture is more sensitive to faults but the loss in accuracy with 4 masked PEs still is only 1.5%.



(a) SqueezeNet *(b) VGG-16*

(c) LeNet-5 *(d) Legend*

Figure 4.7: Average Accuracy with PEs Masked to Zero

b) Worst Case Accuracy

Figure 4.7 only shows the *average* accuracy. To study the worst-case, we performed a second series of experiments. We generated ≈100,000 batches of 100 randomly selected images and analyzed the accuracy with each architecture. For the unprotected architectures, we injected random faults (1, 2 or 4 PEs, with 1 SA fault). For the architectures protected with output masking, we set to zero 1,2 or 4 PEs. In Table 4.3, we report the minimal observed accuracy across all the batches. First, note that with the unprotected architectures, in the worst case, the accuracy drops to zero for the large networks, even with faults on a single PE. This is a key observation for safety applications.

The second observation is that, with output of a faulty PE masked to zero, the worst case accuracy of the MOZART+ architecture is *significantly* better than RS and WS, especially for SqueezeNet. With WS and RS architectures, there exist certain faults that cause a drastic drop in accuracy, unlike the OS architecture which limits the propagation, as shown in Figure 3.7. This is an important take away for fault tolerant systolic accelerators.

	Network	Unprotected				Zero Masking		
Number of Faulty PE		Fault Free	OS	WS	RS	OS (Mozart)	WS	RS
1 PE	SqueezeNet	64	0	0	0	**58**	2	52
	VGG	82	0	0	0	**82**	**82**	79
	LeNet	95	5	5	4	95	**96**	77
2 PEs	SqueezeNet	64	0	0	0	**53**	1	45
	VGG	82	0	0	0	**80**	77	74
	LeNet	95	2	3	2	92	**95**	72
4 PEs	SqueezeNet	64	0	0	0	**53**	0	16
	VGG	82	0	0	0	**79**	73	63
	LeNet	95	4	2	2	90	**95**	58

Table 4.3: Worst Case Accuracy for Batches of 100 Images

4.3.3 Efficiency of the Fault Tolerant Scheduling

As seen in Sec.4.1.3, out proposed fault mitigation strategy is not adapted for the last fully-connected layer with an OS data-flow, where each PE is dedicated to an output neuron or image class. As a consequence, masking a PE used for an output neuron drastically impacts accuracy.

This is why in the previous section, the MOZART+ approach provided improved fault tolerance for the larger SqueezeNet and VGG-16 networks, but not for LeNet-5 (Fig. 4.7c) where the WS architecture actually performs better.

In LeNet-5 there are only ten output neurons. If any one of these is faulty, and thus masked to zero, an entire output class is lost, resulting in a major loss in accuracy. For example, the LeNet-5 considered in our studies has an accuracy of 99% without faults or hardware masking but its accuracy drops to 89.1% when one output neuron is masked to zero.

To address this shortcoming, we propose an alternative scheduling algorithm for the last layer, described in the Sec.4.1.3. To avoid the problem of mapping an output neuron to a faulty PE, we propose to schedule the output layer to avoid any columns with a faulty PE. In other words, for the last layer we do not consider the results of columns of the systolic accelerators that contains faulty PEs. It results in an additional number of times the array is loaded for the last layer, described in Tab. 4.4 (SqueezeNet is fully convolutional thus not shown). This results in a limited overhead.

The improved fault tolerance for LeNet-5 is shown in Fig. 4.8. This figure shows the average accuracy for 1000 batches of 100 images for LeNet-5 without faults and simulated on faulty systolic accelerators with masking of erroneous PEs. Three accelerators are simulated : WS, Mozart, and Mozart+ with the usage of new fault tolerant scheduling to disable erroneous PE from the last layer. The systolic accelerators are composed by 16x16 PEs and 1, 2 and 4 PE are considered faulty. As we seen earlier in Fig.4.7c, the WS architecture with masked PEs is more fault tolerant in every cases compared to Mozart. This is due to the fact that when an output neuron assigned to the expected label is masked to zero, then the prediction of the DNN is almost always false.

When this issue is prevented with our specific fault tolerant scheduling, our accelerator is more fault tolerant than the WS architecture and the loss of accuracy is negligible with up to 4 faulty PEs. [1]

Network	All but last FC Layer	Last FC Layer	Mozart+ Overhead
LeNet-5	83	1	1.2%
VGG	55 616	63	0.1%

Table 4.4: Number of Loads of a 16x16 OS PE Array for Each Network

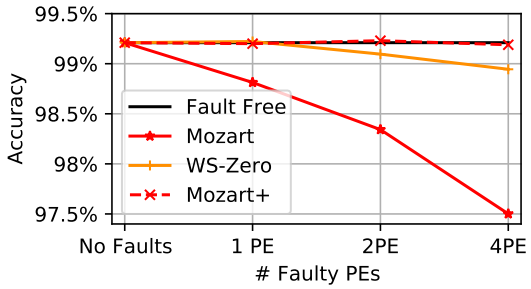


Figure 4.8: Fault Tolerance of LeNet-5 with Mozart+ Technique

4.3.4 Efficiency of Dropout During Training

Dropout during training can be used to improve the robustness during inference. We trained SqueezeNet with 5% dropout on *all* layers, whereas typically dropout is only applied to the last layers. The training time increased ($\approx 3\times$), but the resulting configuration provided additional fault tolerance.

Fig. 4.9 shows results comparing SqueezeNet trained with and without random dropout on every layers when executed with MOZART+ with an increasing number of faulty PEs whose outputs are masked to zero. With faults in four PEs, due to the new training, the accuracy increased from 67% to 72%. This modified training was performed with no knowledge of the faults. This is different from [Zhang2019], where the re-training was done based on specific position of the faults.

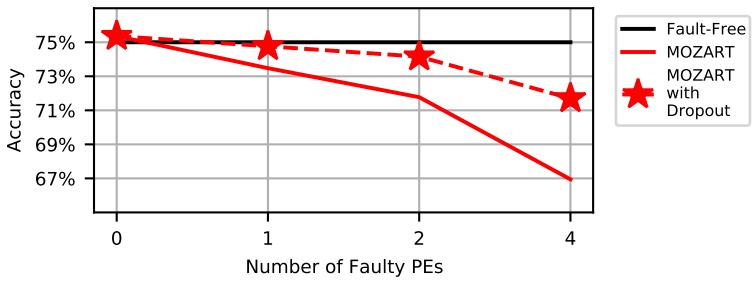


Figure 4.9: SqueezeNet Average Accuracy with Faulty PEs when Trained with Dropout

[1] We only simulated up to 4 faulty PEs to be consistent with the other simulations in this chapter, but as our results with the WS architecture are consistent with [Zhang2019], our accelerators is expected to be more robust to higher rate of faulty PEs. Further experiments are required to confirm this estimate.

4.3.5 Scalability of MOZART+ on Bigger PE Array

Up to now, data has been presented for a 256 PE (16x16) array. We tested the MOZART+ architecture for different PE array sizes and the results are shown in Figure 4.10. These results concern SqueezeNet with a single SA fault on a single PE. As expected, when the array size increases, the impact of a single faulty PE is reduced and become important for smaller size of arrays. In all cases, MOZART+ provides a significant improvement and it is clear that for even larger arrays, taking a single PE off-line for testing is very much a realistic solution which has negligible impact on accuracy.

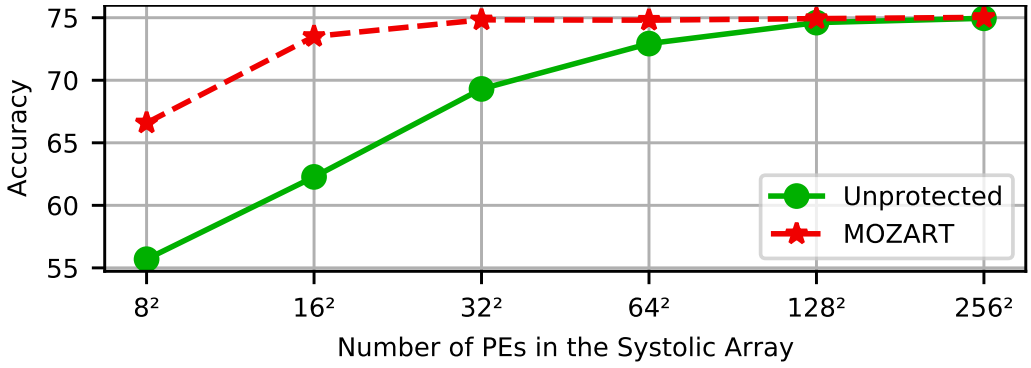


Figure 4.10: SqueezeNet Average Accuracy for Varying Array Sizes (1 SA fault on 1 PE)

4.4 Comparison with the State of the Art

The MOZART+ architecture was inspired by ideas already present in the scientific literature, as well as the results of the fault-injection study presented in chapter 3.

The fault tolerance of systolic arrays for general level computing has been extensively studied in the literature [Kim1989]. For general purpose computing, the requirement is that the computational result must be exactly correct (bit accurate), but for DNNs, since their prediction are never perfect, this constraint can be relaxed.

The fault tolerant training of MOZART+ was inspired by existing work on training for fault tolerance as presented in Sec.2.3.2. Often, fault tolerant training techniques require knowledge of the actual faults. Instead, our training does not rely on specific faults knowledge and thus is close to the technique proposed in [Solovyev2019]. Our contribution is that we can detect the highly impacting SA1 faults and remove them by masking a PE's output to zero.

The work of Zhang [Zhang2019] is also similar to that presented in this chapter. They propose fault mitigation for a WS accelerator, using a multiplexer to mask the output of the erroneous multiplier of a PE and a fault tolerant training technique. Their results, but have certain limitations. First, they have chosen a WS Architecture, which we have shown is more likely to propagate faults. Also, their proposed training techniques requires knowledge of the hardware faults prior to training. Finally, they do not propose test technique to detect faults and only study LeNet-5, which is not representative of modern DNNs. For these reasons, we believe that MOZART+ represents an improvement over the work of [Zhang2019].

4.5 Future Work

We proposed a systolic DNN accelerator specifically designed for robustness in which PEs are able to perform functional on-line test to identify and mitigate permanent arithmetic errors and achieve this with a limited overhead. However, this is not sufficient to meet the safety standards, that require a system to be globally robust to faults [ISO-26262.18]. Our testing and fault mitigation strategy focuses on errors in the data-path of the PEs. In future works, an improvement of MOZART+ that would be able to also detect faults in the communication channels between PEs or in buffers is considered. We note that there already exist contributions on these two domains [Motaman2019].

Similarly, this chapter only focuses on arithmetic faults, but the memories in accelerators are subject to faults. With a small overhead, memories can be protected using traditional Error Correcting Codes and in the next chapter, we present an innovative technique that can mitigate the impact of faults in memories with zero area zero overhead.

4.6 Conclusions of the Design of MOZART+

In this chapter, we presented MOZART+, a fault-tolerant systolic DNN architecture integrating fast on-line test and masking of faulty PEs. It also integrates optimizations to handle faults in the output neurons and a modified training technique. Almost all the impacting SA1 faults can be detected in the time to process a single image, and we showed that the reduction of accuracy in the presence of faulty PEs is gradual. This is achieved with a hardware overhead of approximately 8%.

One limitation of MOZART+ is that this architecture is dedicated to the detection and mitigation of permanents faults. This highlight the needs to protect the memory of DNNs and guided us in the design of an adapted fault detection and mitigation technique presented in the following chapter.

5

Techniques for Protecting the Weights of DNNs

In the previous chapter, we focused on fault tolerance techniques for the data-path. In this one, we propose a technique to protect the weight storage memory, another important component of any DNN accelerator. We notice that the numerical format used for representing weights and activations plays a key role in the computational efficiency and robustness of Convolutional Neural Network (CNN)s. Comparison of the robustness of state-of-the art Convolutional Neural Network (CNN)s implemented with 8-bit integer, Brain-Float 16 and 32-bit floating point formats is performed. We also introduce an error detection method called Opportunistic Parity (OP), which alters the Least Significant Bit (LSB) of certain weights of a given Convolutional Neural Network (CNN) to ensure that all the weights have the same parity code. This technique can detect an odd number of errors in the weights. By setting the faulty weight to zero, the robustness of floating point based weights to bit-flips can be improved by up to three orders of magnitude.

Convolutional Neural Network (CNN)s are increasingly used in applications where safety requirements must be respected, notably in autonomous driving systems. The implementation of these networks on hardware platforms must thus be tolerant of faults of any kind, including soft errors. As has been presented in Sec.2.2.1, the data type plays a key-role in the robustness of DNNs. While quantized DNNs are tolerant to a large number of bit-flips [Schorn2019], there are cases where a single bit-flip can cause the accuracy of a floating-point based DNN to drop dramatically [Malekzadeh2021]. Meanwhile, as the quantization process may induce a loss of accuracy that could be problematic for some applications, then floating point formats still remain important. In this chapter, we propose Opportunistic Parity (OP), a zero-overhead error detection technique combined with an error masking strategy.

Several authors have shown that CNNs can tolerate a significant number of errors in the Least Significant Bit (LSB)s of their weights [Li2017] Based on this observation, we propose a hardware-based technique to detect and mask errors on the weights of CNNs. The concept is to use some of the least significant bits in the weights in order to introduce a simple parity code to detect single (or any odd number) of bit-flips. If the parity code detects an error, the erroneous weights are replaced with with zero.

In CNNs, it is known that masking an erroneous weight or activation value to zero has a minimal impact on the classification accuracy [Zhang2018b, Reagen2016a]. Therefore, when a parity signals a detected error, the impact of the error is minimized by replacing the data with zero.

This technique was tested on three CNNs with different numerical formats. Our results demonstrate that the OP technique improves the robustness of floating-point-based CNNs by one to three orders of magnitude, with zero area storage overhead.

This chapter contains two main contributions. First, we perform a study of the robustness of three modern CNNs including their execution using the recent *bf16* numerical format[1]. Secondly, we propose a new fault mitigation technique, OP, that reduces the impact of bit errors in the weights. Tolerance to bit flips on the weights is important in high-reliability applications, where safety is a concern. Indeed, by making a CNN resilient to weight errors, we open the path to a wider acceptation of weight storage using emerging, lower-energy memory technologies, which are more prone to bit errors due to defects and variations.

This chapter is organized as follows : Sec.5.1 covers relevant background material. Sec.5.2 presents the OP, our zero overhead method to mitigate faults in weights. Sec.5.3 presents our experimental results that demonstrate the efficiency of our technique. Sec.5.4 compares our technique with the scientific literature. Sec.5.5 proposes ideas to improve the OP. Finally, Sec.5.6 concludes this chapter.

5.1 Context

5.1.1 Case Study

This subsection presents the tested CNNs and data types.

To improve the adoption of DNNs in embedded systems, the reduction of their power consumption and memory footprint is currently an active field of research. Three main avenues are being explored :

- **1)** Optimization of the network topology, which has resulted in the development of networks such as MobileNet or SqueezeNet which reach state-of-the-art performance while reducing the number of operations and weights.

[1] [Li2017], studies the IEEE-754 16-bit floating point format, which is slightly different

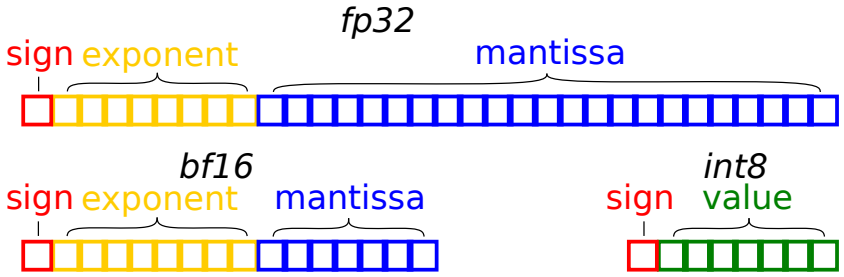


Figure 5.1: Three Numerical Formats used for CNN Weight Values

- **2)** The development of dedicated hardware accelerators, such as Eyeriss [Chen2016] and Shidiannao [Du2015], which are optimized for the evaluation of neural networks.

- **3)** The usage of custom data-types, like reduced precision floating-point or integer that reduce memory bandwidth and storage requirements. *brain float 16 (bf16)* is a recent 16-bit floating-point numeric format used in new Intel Xeon processors [Intel]. It has the same exponent width as a standard IEEE 754 single precision float. The length of the mantissa is simply reduced to 7 bits. When used in CNN accelerators, the multiply operations are performed using *bf16*, however, the sum is accumulated in a standard *32 bits floating point (fp32)* format to limit the rounding error, as described in [Kalamkar2019,Micikevicius2018]. In [Tiwari2022], it was shown that when using *bf16*, the inference process performs 2.21× and 4.19× better in terms of area and power compared to *32 bits floating point (fp32)* on equivalent systolic array accelerator. These numerical formats are illustrated in Fig. 5.1 and considered in our study.

We consider three CNNs : ResNet50 [He2016], SqueezeNetV1 [Iandola2016] and MobileNetV2 [Howard2017]. All three networks are tested with the ImageNet data-set. ResNet50 was chosen as an example of high performance network. It achieves state-of-the-art performance using deep residual layers, which consist of shortcut connections between layers to address the problem of a vanishing gradient during training.

SqueezeNetV1 and MobileNetV2 were chosen as examples of optimized CNNs. SqueezeNetV1 is fully convolutional and uses *fire* modules, which are composed of squeeze layers with a 1x1 convolution followed by an expand layer with 3x3 convolutions. It achieves low latency and has the lowest number of weights of the three networks. MobileNetV2 is based on an inverted residual structure with residual connections between bottleneck layers.

For each of the tested networks, we used publicly available DNN models from [ONNX]. In fact, the weights were converted to *bf16* format by simply rounding the mantissa to 7 bits. The conversion to *int-8* format was performed by the N2D2 platform [Bichler2017] using the methodology described in [Nagel2019].

The classification accuracy is slightly lower when the numerical format for the weights is reduced. In Tab. 5.1, we present the accuracy of the networks, in the absence of faults, as well the network resources requirements in terms of number of Multiply And Accumulate (MAC) operations and the number of stored weights. Compared to the compressed networks, ResNet50 is the most accurate at the cost of significantly more resources usages as it relies on one order of magnitude more Multiply And Accumulate and weights. The *fp32* is the most accurate numerical format. The compression of the network slightly reduces the accuracy of the Deep Neural Network (DNN) from 0 to 0.36 percentage points for *bf16* and from 2.21 to 3.82 percentage points for *int8* quantization.

Network	Num. MACs	Num. Weights	Numerical Format	Top-5 Accuracy
ResNet50	3.9 G	25.5 M	*fp32*	91.90%
			bf16	91.83%
			int8	88.08%
SqueezeNetV1	352 M	1.2 M	*fp32*	80.35%
			bf16	80.35%
			int8	78.74%
MobileNetV2	300 M	3.4 M	*fp32*	89.91%
			bf16	89.55%
			int8	86.39%

Table 5.1: Characteristics and Accuracy of Selected Networks for ImageNet Data-set

5.1.2 Fault Model

The memory of DNNs could be subject to soft-errors induced by transient fault but also of timing errors or retention errors due to the memory being operated at an extremely low voltage or in other worst case corners [Kim2018]. For our experiments, we chose a bit-flip fault model for the weights, and we evaluate the accuracy of the network, as a function of the bit error rate in the memory used to store weights.

5.2 Opportunistic Parity

Numerous well known techniques exist for protecting weight storage memories from bit-flips, including error detection and correction codes with different detection and correction capabilities. In all standard approaches, additional check bits are added to the initial data, in order to provide protection or error correction. Unlike these approaches, we propose to directly modify the stored weights to encode a parity code directly in the data without any additional check bits overhead. In CNN applications, the least-significant bits of the weights are not critical [Li2017]. Indeed, flipping a small number of these LSBs, leads to a very small loss in accuracy of the network.
The Fig. 5.2 illustrates the OP techniques. The left matrix represents a set of 8-bits weights stored in memory before the OP is applied. Each gray case represents a bit that store 1, and each white case represents a bit that store 0. The OP compute the parity code of each weight. The OP switches the logic state of the LSBs of each weight that has an odd parity code.
The right matrix represents the same set of weights once the OP has been applied. As a results, all the weights now have an even number of bit at 1. Since the used parity code can only detect errors but does not contain enough information to deduct which bit is faulty, OP technique is not able to correct only the single faulty bit.

This technique requires no additional storage for the weights, as we use the LSBs of the weights in order for them to match a given parity code. Indeed, we propose a simple parity technique, rather than a stronger code, as a stronger code would require modifying more bits of the weights, or adding additional ones, hence increasing the overhead.

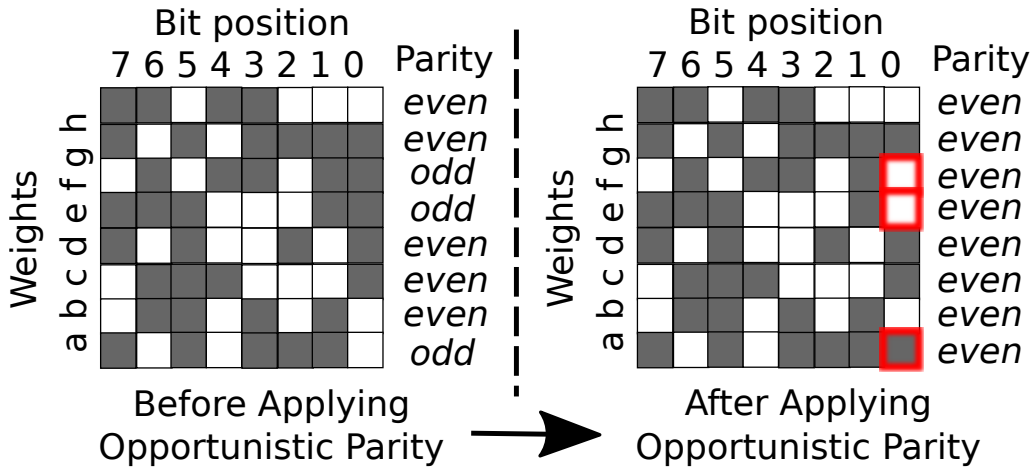


Figure 5.2: Opportunistic Parity : Altering LSB to Obtain Even Parity

5.2.1 Zero Masking Technique

When a parity error is detected, the weight values can be replaced with fault-free values from the reliable memory. If the weight restoration is not available, we propose to mask the erroneous weight. Many authors [Zhang2018b, Reagen2016a] have shown that in CNNs, masking erroneous values to zero is an effective fault mitigation technique, as presented in Sec.2.3.2. We chose this error masking methods as it prevents error scenarios where one of the Most Significant Bit (MSB)s gets flipped to one.

The Fig. 5.3 illustrate the OP techniques combined with this error mitigation methods. The left matrix represents a set of 8-bits weights stored in memory on which OP has been applied before a fault occurred in one of the bits of the weight $\boldsymbol{e}$. Each gray case represents 1, and each white case represents 0. The parity code of each weight is computed, and an error is detected on the weight $\boldsymbol{e}$ since it has an odd number of bits set at one.

The right matrix represents the same set of weights, once the detected erroneous weights have been set to zero. Since the used parity code can only detect errors but does not contain enough information to deduct which bit is faulty, we are not able to correct only the single faulty bit.

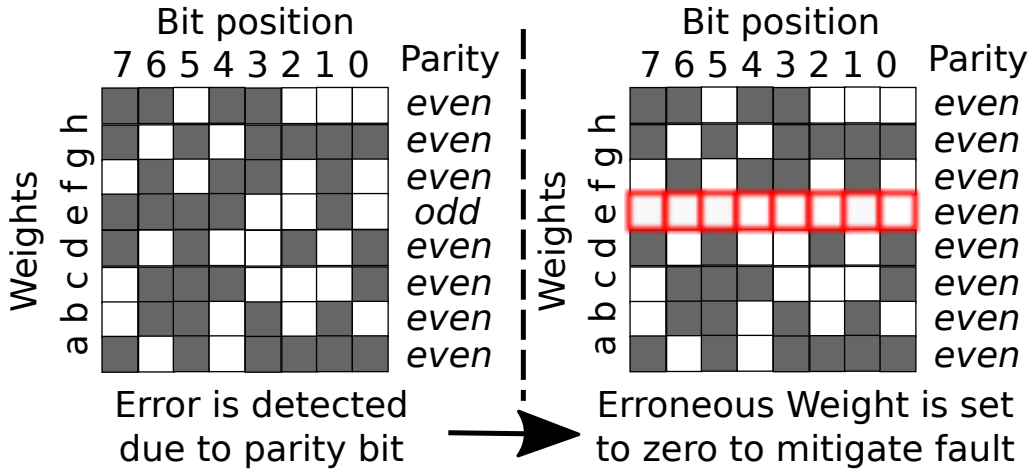


Figure 5.3: Opportunistic Parity : Masking Detected Faults with Zero

At the system level, the parity error can be tested to trigger higher level safety mechanisms. In this case, the proposed zero masking technique is only used to minimize the impact of the fault, until the system is able to reach a safe state.

5.2.2 Bit-Width of Opportunistic Parity

As presented, the LSB of every weights could be potentially flipped, which would have an impact on accuracy. However, we note that the storage word size of most modern memories is quite large (at least 64 to 256 bits). Therefore Multiple (N) weights are stored in one memory word. We can cover the entire memory word with a single parity bit, thus only the LSB of one of the N weight values needs to be flipped. This reduces the number of perturbations on the weights, but decreases the fault detection granularity from one to N weights.

Different techniques could be used in order to select which of the N weight values to modify. One strategy would be to pick the weight with the largest absolute value, to minimize the relative error. Another strategy consists of picking the weight which is the least critical, but this would imply a costly analysis of the criticality of the weights. In the interest of simplicity, we arbitrarily pick the LSB of the last weight in the memory word.

5.2.3 Methodology of analysis

We performed a series of fault injection campaigns to analyze the efficiency of OP technique.

We are working with the ImageNet dataset, which is composed of 1000 labels. The trends of the results are similar for top-1 and top-5 accuracy. In the interests of simplicity, we have chosen to report only top-5 accuracy.

Due to the fact that the classification rate in the absence of faults is not 100%, the uncertainty in the fault free accuracy varies with the batch size. We set a N_{batch_size} of 400 images so the fault-free accuracy batch is in the range $Average\_Accuracy \pm 5 point$ with a confidence interval of 98.518% for SqueezeNet, and more than 99.9% for MobileNetV2 and ResNet50. Consequently, if a batch has an accuracy under this threshold (rounded at the average accuracy -5 percentage points for all the tested networks), we can safely conclude that the simulated fault has resulted in an accuracy loss.

To visually inspect the effect of batch size on the worst case batch accuracy, the minimum and maximum accuracies were measured for 1000 batches of varying size as shown in Fig. 5.4. Although a batch size of 50 appears sufficient to ensure a variation of under 5%, we have chosen a larger batch size of 400. This was done, in order to ensure the uncertainty in the measured results, is well below 5%, for all three networks and numerical formats.

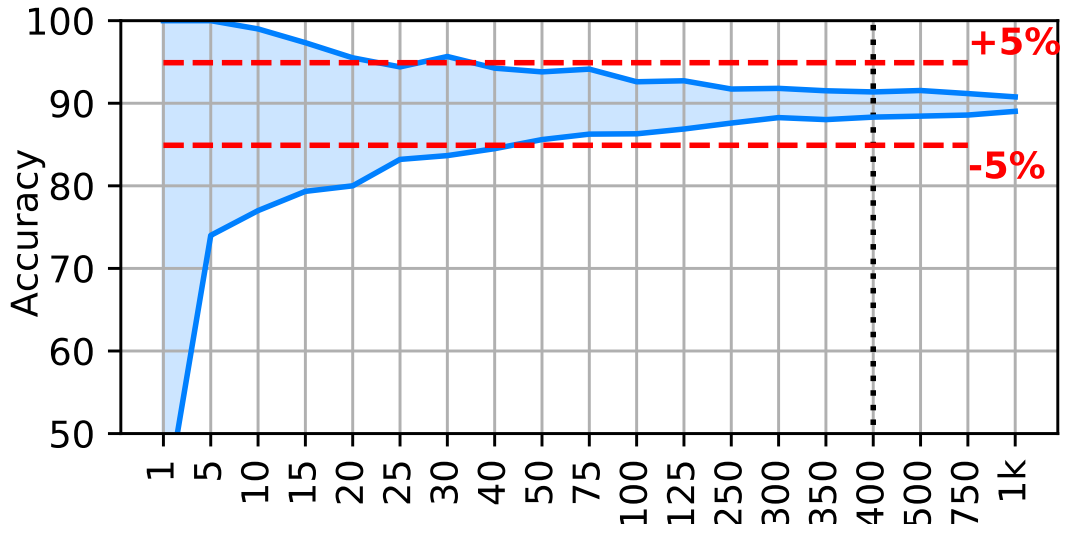


Figure 5.4: Range of Accuracy for a given Batch Size for MobileNetV2 using fp32

5.2.4 Methodology of fault injection

The algorithm for fault injection is presented in Alg. 3.

Each fault injection experiment consists of applying N_{batch_size} images and computing the accuracy for that batch.

Each of the data points we present in the following sections is the average accuracy over N_{exp}= *20* fault injection experiments.

N_{exp} are executed for each of the following variables :

- **Network** : ResNet50, MobileNetV2, SqueezeNet.
- **Numerical Format** : *fp32*,*bf16*,*8 bits integer numeric format (int8)*.
- **Bit Error Rate** : 1e-9 to 9e-1
- **Fault Mitigation Strategy** : *None*, OP over *individual weights*, *64 bits* or *256 bits* words.

Algorithm 3 Fault Injection Procedure for Analysis of OP

```
for all Net in (ResNet50, MobileNetV2, SqueezeNetV1) do
  for all Numerical_format in (fp32, bf16, int8) do
    for all OP in (None, Weight, 64 bits, 256 bits) do
      for BER from 1e-9 to 9e-1 do
        for i from 1 to N_exp do
          N_faults ← (num_weights * BER)
          Faults ← random(weights, N_faults)
          Inject Faults in weights
          for j from 1 to N_batchsize do
            Classify Image_j
          end for
          Record average accuracy for batch
          Clear Faults in weights
        end for
        Evaluate average accuracy for N_exp batches
      end for
    end for
  end for
end for
```

Classification Methodology

When reporting the impact of bit-flips on the weights, we have adopted the maximum Bit Error Rate with Zero Accuracy Drop (BERZAD) metric proposed by Sabbagh [Sabbagh2019]. BERZAD is defined as the maximum bit-error rate (number of erroneous weight bits divided by the total number of weight bits). A higher BERZAD value indicates that the network is more robust.

5.3 Experimental Results

In this section we present the results of the fault injection experiments. First, we present an analysis of the sensitivity of the weights, per erroneous bit position. This first experiment is done to validate our approach of inverting the LSBs. Secondly we present the results of experiments where faults are injected in the weights, both with and without the OP technique.

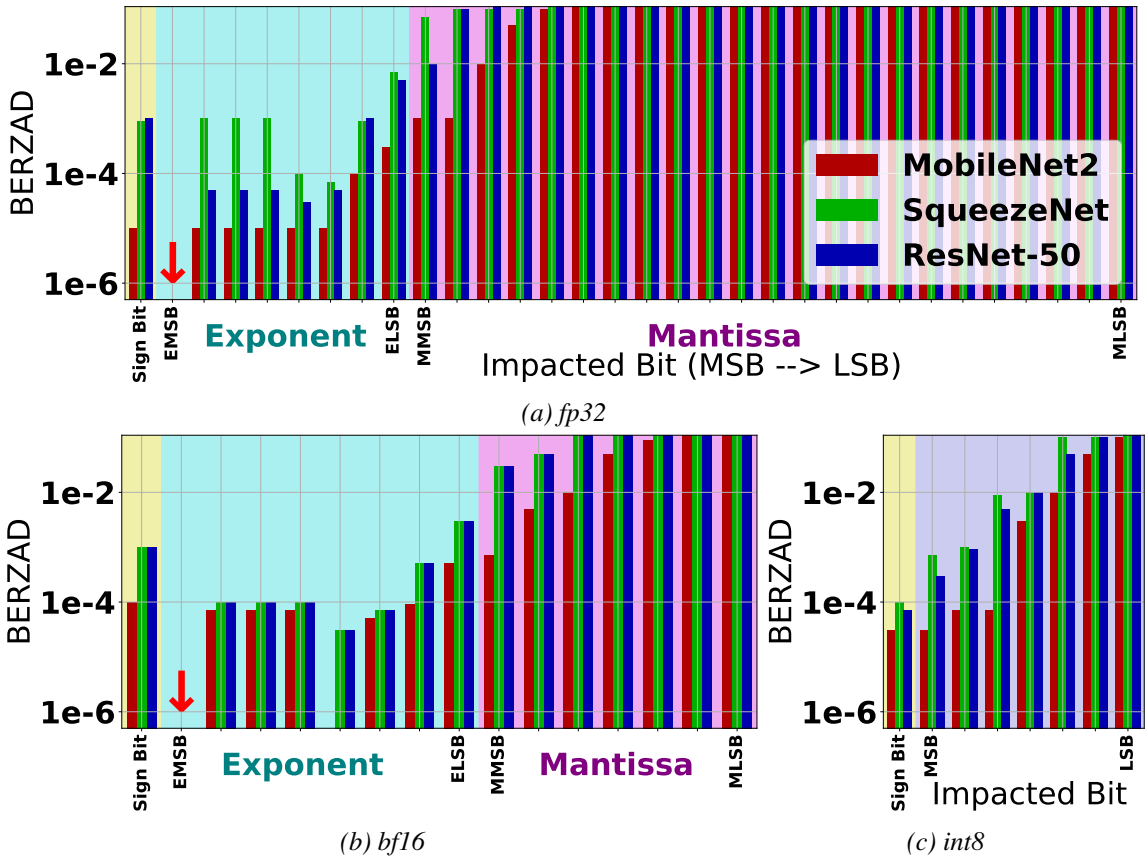
5.3.1 Analysis of DNNs Sensitivity by Bit Position

The sensitivity of all the bit positions in the weights was studied for the three numerical formats (*fp32*, *bf16* and *int8*) for the three networks and the results are shown in Fig. 5.5.

Consistent with previous studies [Li2017], we see that exponent bits are more sensitive than mantissa bits. Also, for all the formats, including the *int8* format, the LSB can withstand a BERZAD of nearly 10e-1. This observation confirms our hypothesis and justifies the proposed OP protecton technique.

We note that for both floating point numerical formats, there are cases where a single bit flip in the MSB of the exponent causes the accuracy to drop to zero. Thus, the BERZAD is zero, and hence off the graph (shown with a red arrow, due to the log scale). This is not the case for the *int8* quantized format, where a small number of bit flips in the MSBs does not produce an accuracy drop over 5%.

Amongst the three networks, we observe that there is a signficant variation in their tolerance to bit-flips of the exponent. SqueezenetV1 is more tolerant to these faults and MobileNetV2 and ResNet50 are more sensitive. One possible explanation is that MobileNetV2 and ResNet50 both have residual connections whereas SqueezeNetV1 does not. It is worth noting that MobileNetV2 and SqueezeNetV1 have a similar number of weights, so the number of weights alone does not explain this difference. It is clear that the structure of the network plays a key role in its sensitivity to weight errors.



(a) fp32

(b) bf16

(c) int8

Figure 5.5: BERZAD as a Function of Bit Position, for Three Numeric Formats and Three Networks

5.3.2 Analysis of the Opportunistic Parity Technique

As discussed previously, OP can be applied based on different memory word sizes. We have considered three cases : (i) parity is adjusted for each weight individually, (ii) parity is adjusted for 64-bit words, (iii) parity is adjusted for 256-bit words. For comparison, we also present the classification accuracy in the absence of fault mitigation. These results are summarized in Tab. 5.2.

In all cases, with both floating point formats, OP greatly increases the BERZAD, indicating the network is more robust. For example, with MobileNetV2, with the *fp32* format, the BERZAD increases from 1e-8 to 7e-6, due to the OP at individual word level. Note that the impact of the weight modifications on the fault free classification accuracy is negligible.

For the case where the *int8* numeric format is used, when OP is applied at the 8-bit word granularity, there is a noticeable drop in fault-free accuracy (for example from 88.1% to 86.5% for ResNet50). However, it does provide a nearly 2.5x (4e-5 to 1e-4) increase in BERZAD. Indeed, when there is a parity error, only a single weight value is zeroed out, which has a minor impact on the accuracy. Still, for the same *int8* format, when OP is applied over groups of N=8 or N=32 weights (64, 256 bit words, respectively), there is virtually no drop in fault free accuracy, as very few bits are perturbed. But, in these cases, the BERZAD is actually lower (worse), than with no protection, since when there is a parity error, N weights get zeroed out and this has a real impact on the classification accuracy. Considering only the BERZAD metric, the benefits of OP are limited for the *int8* format.

One shortcoming with the BERZAD metric, is that it provides no insights into the behaviour of the network, after the initial loss of accuracy. In Fig. 5.6, we plot the accuracy, as a function of the bit error rates. In all these graphs, we can see how the accuracy degrades with increasing bit error rate.

The black curves show the behaviour in the absence of any protection, and we see that for the floating point formats, the drop is quite sudden. This is due to the fact that, as soon as one, or a few, exponent bits are modified, the accuracy drops drastically. However, with OP, we prevent these extreme cases, and thus typically gain two orders of magnitude in BERZAD, before the accuracy drops off. In other words, using OP with floating point weights, it is possible to obtain the same fault tolerance (BERZAD) as a network using quantized weights stored in an *int8* format.

For the *int8* format, the results are mixed. For example, with ResNet50, Fig. 5.6c, with OP at the 8-bit word level (blue trace), we see that the accuracy starts to drop at a higher bit-error rate, and that the drop in accuracy is more gradual. For SqueezenetV1, Fig. 5.6i, OP at the word level, also provides a slight benefit. Unfortunately, for MobileNetV2, Fig. 5.6f, OP, even at the word level, degrades the fault tolerance. To summarize, the OP technique is highly effective at protecting weights stored in floating point formats. For 8-bit integer weights, in some cases, it can provide an improvement, but the results depend on the type of network.

5.4 Comparison with the State of the Art

The existing work in the literature that proposes the closest methodology to the OP technique is presented in [Guan2019]. In this article, similar to our idea, the authors propose to map an Error Correction Code (ECC) in the weights of a network to improve the robustness of the model with virtually zero overhead. The difference is that these authors only consider quantized DNNs. They propose to restrict the training and the quantization process to alter the weight distribution to increase the number of unused MSBs after the quantization. These unused bits are then used to store an ECC. The strengh of this article is that they can insert an ECC in the weight of a quantized DNN with no impact on the accuracy. However, we believe that their methodology has a hidden drawback, as they constrain 7 of 8 weights values to fit in 7 bits range, which restrict the quantization and induce a 12.28% overhead (bits used to store the ECC)

Network	Num. Format	Number of Masked Bits	No-Fault Accuracy	BERZAD
MobileNet	*fp32*	None	89.9	1e-8
		32	89.9	7e-6
		64	89.9	5e-6
		256	89.9	8e-7
	bf16	None	89.6	0
		16	89.3	2e-6
		64	89.6	5e-7
		256	89.6	9e-7
	int8	None	86.4	2e-5
		8	79.0	9e-6
		64	85.8	7e-6
		256	86.1	6e-8
SqueezeNetV1	*fp32*	None	80.3	9e-8
		32	80.6	4e-5
		64	80.5	4e-5
		256	80.3	4e-6
	bf16	None	80.6	7e-8
		16	80.6	4e-5
		64	80.6	3e-5
		256	80.4	2e-6
	int8	None	78.7	1e-4
		8	77.9	2e-4
		64	78.7	3e-5
		256	78.8	8e-6
ResNet50	*fp32*	None	91.5	2e-9
		32	91.9	8e-6
		64	91.9	7e-6
		256	91.5	2e-6
	bf16	None	91.7	0
		16	91.8	7e-6
		64	91.7	3e-6
		256	91.8	4e-6
	int8	None	88.1	4e-5
		8	86.5	1e-4
		64	88.3	1e-5
		256	88.2	6e-6

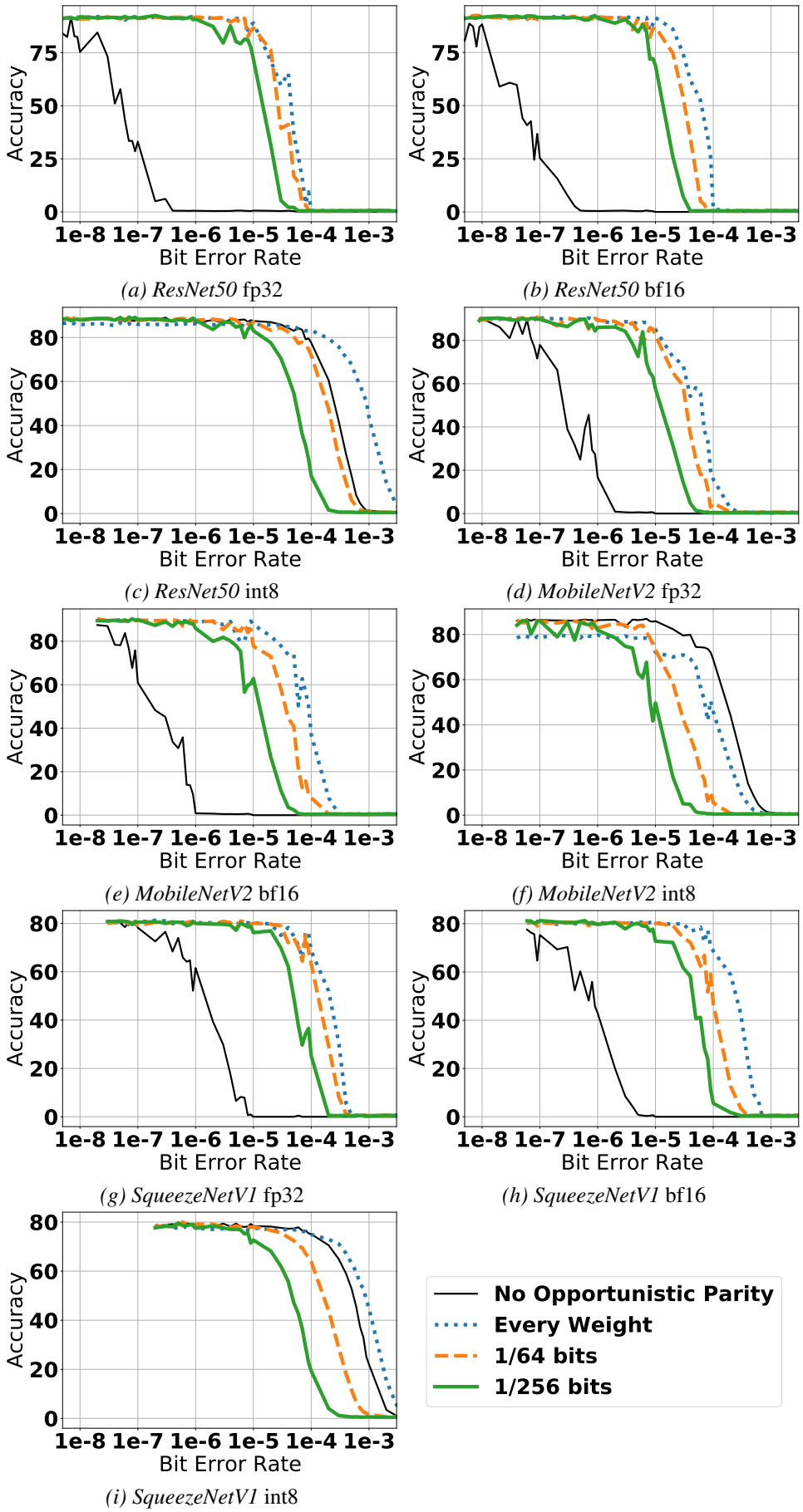
Table 5.2: Characteristics and Accuracy of Opportunistic Parity

5.5 Suggested Improvements

The proposed methodology in this chapter considers parity-bit to detect bit-flips. This technique only detect an odd number of bit-flips, and once a fault is detected the whole word or range of words are masked.

This technique could be improved in two ways :

- **1)** As seen in Fig.5.5, LSBs of floating-point based DNNs are robust to a large number of faults. While we only altered the LSB to code a parity error check, we could have used more bits to use a stronger ECC.
- **2)** As seen in Fig.5.6, OP results in a loss of accuracy in int8 quantized DNNs. We could alter the quantization or the training process so that the DNN adapts to the weight modifications induced by OP.



(a) ResNet50 fp32 *(b) ResNet50* bf16

(c) ResNet50 int8 *(d) MobileNetV2* fp32

(e) MobileNetV2 bf16 *(f) MobileNetV2* int8

(g) SqueezeNetV1 fp32 *(h) SqueezeNetV1* bf16

(i) SqueezeNetV1 int8

Figure 5.6: Impact of Opportunistic Parity on Accuracy

5.6 Conclusion of the Opportunistic Parity

In this chapter, we have presented a fault injection study targetting the weights for three CNN architectures. Our results show that the sensitivity of modern, highly compressed networks, such as SqueezeNetV1 and MobileNetV2 to bit-flips in the weights, varies significantly. We believe the topology and the presence of residual layers plays a role. We also studied three numerical formats for the weights, including the *bf16* format. This numerical format is quite sensitive to bit-flips, as the exponent occupies a large fraction of the total bits.

We proposed a simple technique, called opportunistic parity, which can be used to protect the memory used for weight storage for a CNN accelerator and requiring zero storage overhead. In the case of weights stored in a floating point format, including *bf16*, it provides an improvement in bit error rate tolerance between 1-3 orders of magnitude with practically no loss in fault-free accuracy. For *int8* formats, the technique can provide approximately a 5x improvement in bit error rate tolerance, with a minor loss in fault free accuracy, however, the results depend on the type of network.

6

Symptom Based Error Detection

Semantic segmentation of images is essential for autonomous driving and modern DNNs now achieve high accuracy. In this chapter, we present an analysis of the effect of faults in Google's DeepLabV3+ network processing an industrial data-set. A new symptom-based fault detection algorithm is proposed and show to detect >99% of critical faults with zero false positives and a compute overhead of 0.2%. Furthermore, these faults can be masked, virtually eliminating all critical errors.

For many years, Deep Neural Network (DNN)s have been used for image classification and today they achieve better accuracy than humans on well-known data-sets such as ImageNet. Semantic segmentation [Everingham2009] is a more difficult task where the goal is to assign labels to every pixels in an image. In an autonomous vehicle, semantic segmentation enables the system to identify the objects in the environment (pedestrians, vehicles, road signs, road) with pixel-level accuracy.

Advanced Driver Assistance Systems are described in terms of automation levels [Tax2021] and systems designed for the highest levels of automation must comply with ISO-26262 Automotive Safety Integrity Level D [ISO-26262.18], requiring a high-level of fault-detection and reliability.

Another approach to detect hardware faults, called Syndrome-Based Error Detection (SBED), is based on analyzing the aggregate behavior of the values, when the network is performing inference, and looking for symptoms that suggest abnormal behavior induced by a fault. Li [Li2017] proposed a technique based on detecting excessively large activation values and tested it on three smaller networks for image classification. Li's approach considered a single statistic (MAXimum (MAX)) and suffered from a high false positive rate (reporting of a fault, when indeed there was no fault). To improve on this, we propose a new SBED that considers four different statistics (MINimum (MIN),MAX,AVeraGe (AVG),STandard Deviation (STD)) and uses a methodology to set detection thresholds so that false positives can be eliminated.

This chapter provides the following contributions. The work presented in this chapter is one of the first studies of the fault tolerance of DNNs used for semantic segmentation. We show how an existing taxonomy [Corneliou2021] for classifying the impact of faults can be extended to semantic segmentation. We analyse the fault tolerance of the state-of-the-art DeepLabV3+ [Chen2018] network applied to an industrial autonomous driving data-set [Yogamani2019]. We present a new SBED technique, using multiple statistics, which has close to 100% detection of critical errors and no false positives using our data-set and fault-models. Finally, we demonstrate that it is possible to mask the critical faults and virtually prevent critical error cases.

This chapter is organized as follows : Sec.6.1 presents background material. Sec.6.2 presents our fault injection methodology to assess the robustness of DeepLabV3. Sec.6.3 presents our technique based on symptoms to detect the propagation of faults. Sec.6.4 presents our fault mitigation technique. Sec.6.5 compares our technique with the related works. Sec.6.6 proposes ideas to improve this fault mitigation technique. Finally, Sec.6.7 concludes this chapter.

6.1 Context

6.1.1 Semantic Segmentation and WoodScape

For many years, DNNs have been used to perform image classification, and many researchers have studied the well-known ImageNet [Krizhevsky2012] data-set. More recently, the computer vision community has shifted its focus to more difficult tasks such as object detection and semantic segmentation.

The latter (Image segmentation) consists of labeling each pixel in an image according to a set of known classes, while tagging the remaining pixels as background. Two of the major hurdles in image segmentation are to recognize objects at a range of different scales and to achieve a high output feature resolution. DeepLabV3+ employs an encoder that uses of a series of *atrous* convolutions with different rates (a technique known as Atrous Spatial Pyramid Pooling) to address these issues and, with a relatively simple decoder, it achieves high accuracy and spatial resolution. We have used a version of DeepLabV3+ based on ResNet-101 [He2015] as the backbone and Tab. 6.1 summarizes the key network characteristics.

Layers	Numeric Format	Weights	Neurons	FLOPs	Input Image	Output Image
109	FP32	58.6M	107M	309 G	640x483	160x121

Table 6.1: Characteristics of DeepLabV3+

To come closer to real industrial problem, we use the Woodscape [Yogamani2019] dataset, a publicly available multi-camera fish-eye data-set for autonomous driving including annotations for semantic segmentation. With four cameras, the fish-eye perspective provides a 360° view, motivating our selection of this data-set containing 6587 training images and 1647 test images.

6.1.2 Fault Model

The final reliability of the system depends on the DNN as well as the processing unit (Graphics Processing Unit (GPU), Field-Programmable Gate Array, Tensor Processing Unit, as seen in chapter.4). In a fault-tolerance study, the choice of a fault model is important. Common fault models used previously include faults in external memories [Nguyen2019], on-chip Static Random Access Memorys [Kim2018], arithmetic units [Mahdiani2012] as seen in chapter3 and 4, interconnects [Motaman2019] or abstraction at the level of DNN weights as proxy of memory errors, as seen in chapter 5.

The focus of this chapter is on the new semantic segmentation application, thus, to remain independent of a specific hardware platform, we have chosen one fault model on two targets : bit-flips occurring in the weights (Weight Fault Model (WFM), similar to the study seen in chapter 5) and in the activations (Activation Fault Model (AFM)). The bit-flips in the weights represent errors in external memory or on-chip RAMs, while the latter is a proxy for logic errors in the arithmetic units. These errors are injected after the batch-normalization but prior to the *ReLU* activation function as shown in Fig. 6.1. During inference, batch normalization is simply a linear transform of activations and hence can be merged with the previous convolution [Ioffe2015]. Thus motivate our choice to inject faults after the batch-normalization.

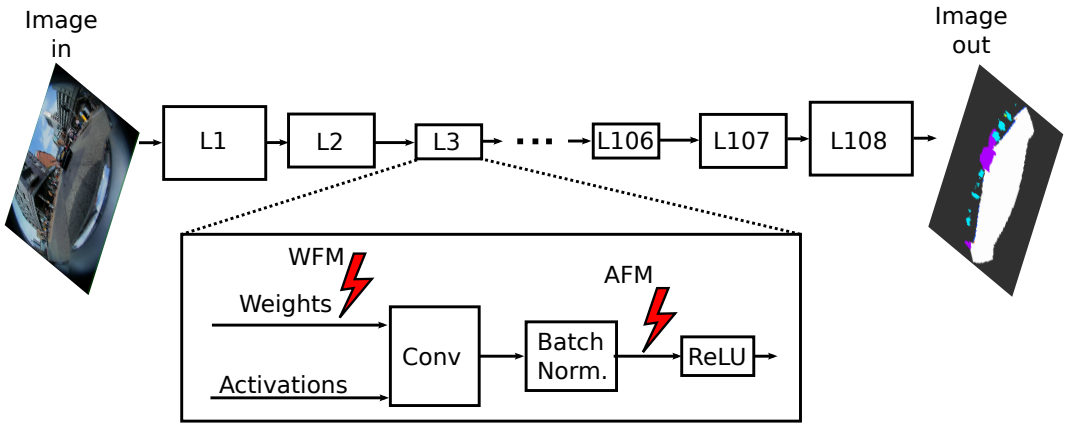


Figure 6.1: Faults Model for Weights and Activations

6.1.3 Error Taxonomy

Previous authors have classified the impact of faults on micro-processors and in [Weaver2004] the authors introduced the notions of Silent Data Corruption (SDC) and Detected Unrecoverable Error, as seen in Sec.2.1. Recently, this has been extended to DNNs performing image classification [Corneliou2021]. In image segmentation, the output is a full image, not a single class, thus this approach can not directly be applied, therefore, we propose a modified taxonomy shown in Fig. 6.2.

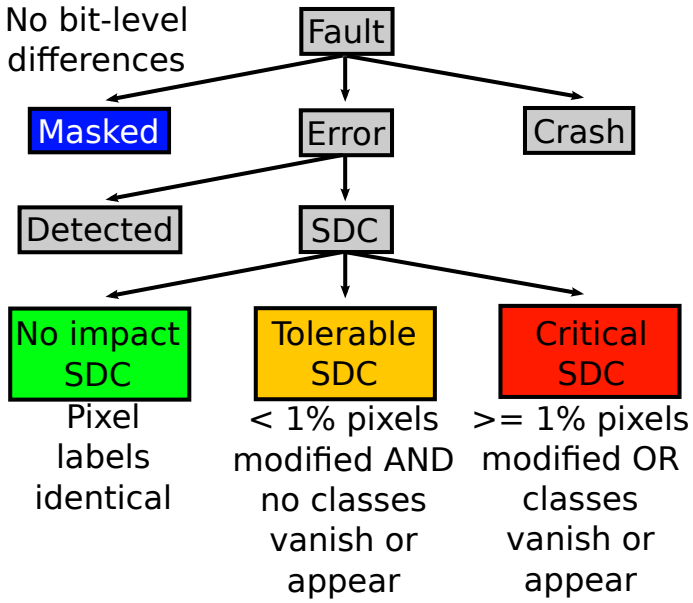


Figure 6.2: Taxonomy of Error Outcomes for Semantic Segmentation

In some cases, a fault is completely masked and the final output of the network is identical at bit level with the fault-free execution. This case is called *masked* and shown in blue. If a fault is not detected and the output is modified, then an SDC has occurred. If the change in the floating-point output values is minor to the point that none of the labels of the pixels is modified, then we denote this as a *No Impact SDC*, shown in green. If the labels of only a small number of pixels (less than 1%) are modified *and* no class appears or disappears as a result of the fault, then we classify the outcome as a *tolerable SDC*, shown in yellow. Finally, if more than 1% of the pixels are modified, or a class vanishes or appears, we say it is a *critical SDC*, shown in red. Our distinction between *tolerable SDCs* and *critical SDCs* is somewhat subjective. However, visually it is difficult for a human to observe the impact of the *tolerable SDCs*. Moreover if no class appears or disappears, the downstream algorithms are unlikely to make a different decision. Image segmentation algorithms do not achieve 100% accuracy, so the overall system must be resilient against minor errors.

In Tab. 6.2 we show examples of two different images where two different faults were injected. In the first row, two different activation faults were injected. One produced a *tolerable SDC* and the other a *critical SDC*. It is remarkable to see how a single fault in the activations can produce an output image that is massively corrupted. Although DNNs have a certain inherent fault tolerance, there do exist minor faults that can cause serious errors, as seen in Sec.2.2.1. In the second row, we shown an example of two different faults in the weights (WFM). These examples show the importance of analyzing the impact of faults and preventing the critical SDCs going undetected.

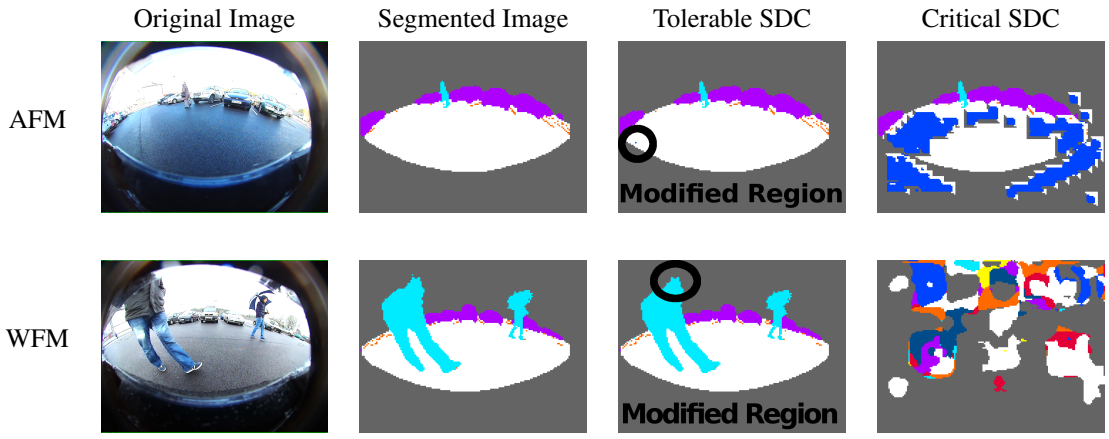


Table 6.2: Examples of Tolerable and Critical SDCs

6.2 Fault Injection Methodology

Fault were injected into randomly selected images from the test data-set, with the fault model described in Sec.6.1.2 (AFM and WFM). We performed two fault injections campaign, one with single bit-flip and one with multiple bit-flips (1 to 10 bit-flips, with a normal distribution centered at 5). Our experiments were performed using PyTorch v1.9 and the effects on the segmented images were classified using the taxonomy described in the Sec.6.1.3.

6.2.1 Impact of Single Bit-Flip

The single-bit fault injection campaign was performed with 8,000 randomly selected single-bit faults. In Fig. 6.3 we show an example of a single bit-flip in an activation value that occurred during our experiments. In this example, the bit-flip impacts a bit in the exponent resulting in an extremely large erroneous activation value. As seen in the figure, this single erroneous activation value is sufficient to completely corrupt the segmented image, resulting in a *critical SDC*.

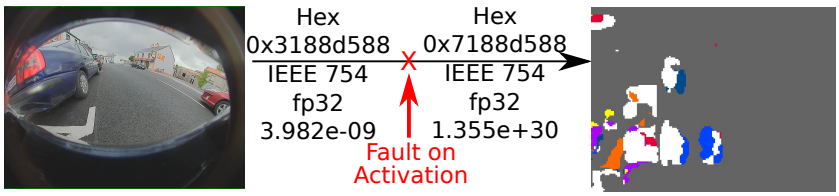
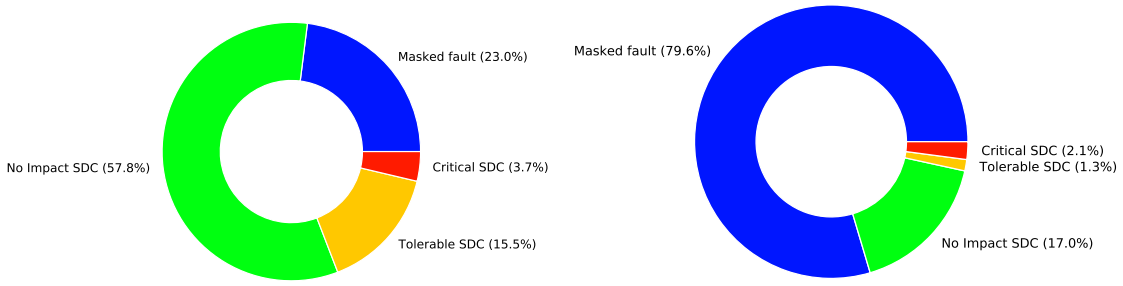


Figure 6.3: Impact of Single Bit-flip in MSB of an Activation Resulting in a Critical SDC

The aggregate results of the fault injection study are presented in Fig. 6.4a for faults in the weights and in Fig. 6.4b for faults in the activations. In the activations, nearly 80% of the faults are fully masked, whereas in the weights only 23% are fully masked. This can be explained as each weight is used repeatedly. Even though the weights are used multiple times, a large fraction (57.8%) of the faults result in *No Impact SDCs*. The rate of critical faults in both fault models is similar (2.1% for activations and 3.7% for weights). We also note that *Tolerable SDCs* are rarely produced (1.3%) as a result of a single bit flip in an activation, however, they are more common with weight faults (15.5%), which is again explainable by the re-use of the weights.

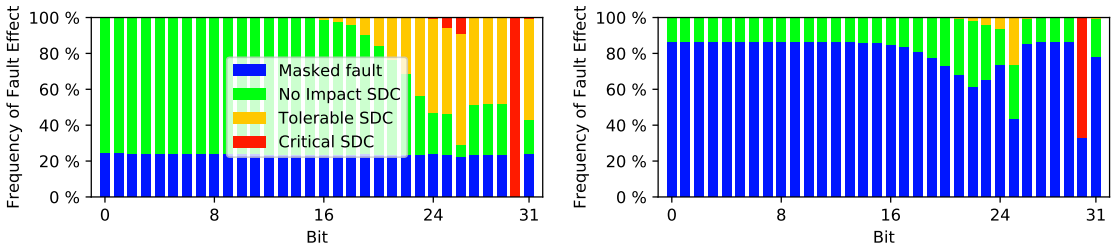


(a) Fault on Weights *(b) Fault on Activations*

Figure 6.4: Aggregate Impact of Single Bit Faults

Bit Position Analysis

When working with data in a FP32 numeric format (see Fig. 6.6), upsets in the exponent are expected to be more critical than those in the mantissa. In Fig. 6.5a and Fig. 6.5b we plot the impact of faults in each bit position, in the weights and in the activations respectively. As expected, the results of faults in the 16 Least Significant Bit (LSB)s of the mantissa, are either *masked* or a *No Impact SDC* outcomes. For weight faults, upsets in the Most Significant Bit (MSB)s of the mantissa, start to produce *Tolerable SDCs*.



(a) Fault on Weights *(b) Fault on Activations*

Figure 6.5: Impact of Single Bit Faults by Bit Position

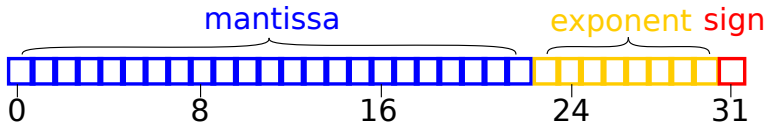


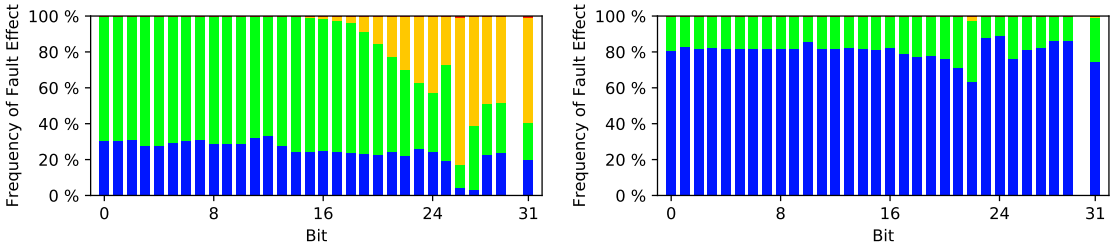
Figure 6.6: IEEE-754 32 bit floating point

It is interesting to note that all the *Critical SDCs* are the result of bit-flips in the exponent bits. Indeed, for the activations, it is only the MSBs of the exponent (bit 30) that results in *Critical SDCs*. From these observations, we see that, with the exception of a few exponent bits, the application is actually highly tolerant of bit-flips.

Analysis by Bit Value

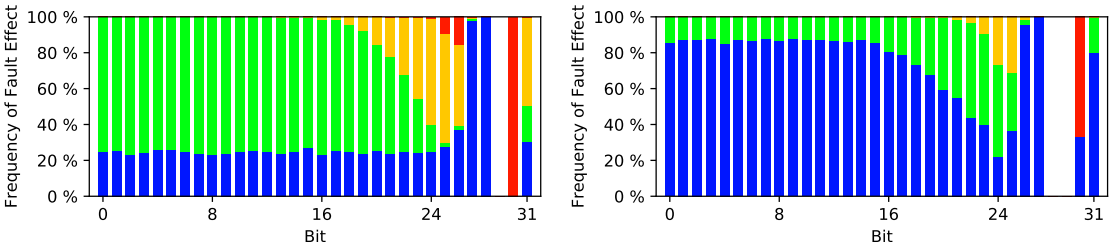
In the literature [Zhang2018b, Reagen2016b], it has been shown that when values (weights or activations) in a DNN are replaced with zeroes, the impact on the classification accuracy is small. To confirm this result for this application we separately analyzed the impact of $1 \rightarrow 0$ and $0 \rightarrow 1$

faults, in each bit position, for both the weights and the activations. The results are shown in Fig. 6.7a to Fig. 6.8b. We note that in some figures, for certain bit positions (28-30), no bars are shown. This is because across the full data-set, there was never a case when the bit took on the initial value of 0 or 1, as required for the bit-flip.



(a) Fault on Weights *(b) Fault on Activations*

Figure 6.7: Impact of Single Bit $1 \rightarrow 0$ Faults



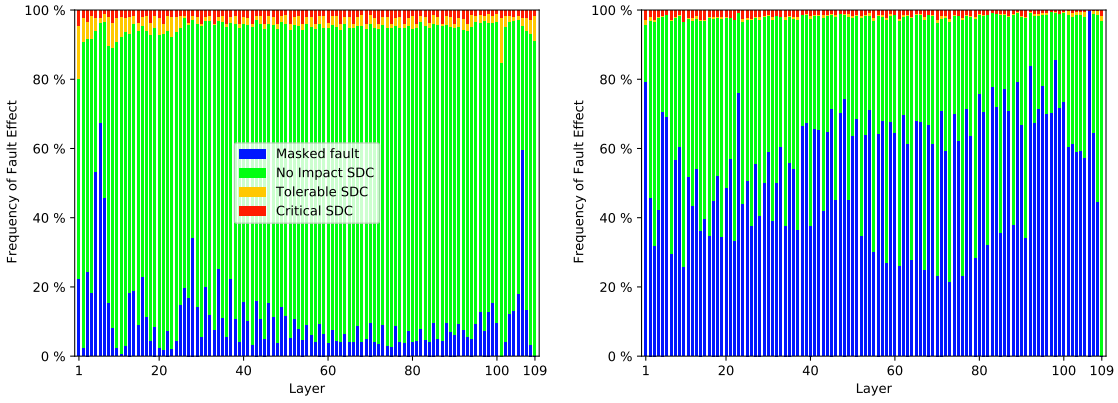
(a) Fault on Weights *(b) Fault on Activations*

Figure 6.8: Impact of Single Bit $0 \rightarrow 1$ Faults

The first thing we notice is that all the *Critical SDCs* are the result of $0 \rightarrow 1$ bit-flips. Conversely, the $1 \rightarrow 0$ bit-flips are much less impacting and even the exponent is tolerant. Overall, nearly 20% of the $0 \rightarrow 1$ bit-flips are fully *masked* in the weights and close to 80% in the activations.

Analysis by Layer

We then analyzed the impact of faults in each layer of the network. For this analysis, we injected 3000 single bit faults in either the weights (WFM) or activations (AFM), for each layer. The results are shown in Fig. 6.9a and Fig. 6.9b. The percentage of critical errors (red) is small, for all layers and activation faults (AFM in Fig. 6.9b) are much more likely to be masked (taller blue bars). For the encoder layers (ResNet backbone from layer 1 to 101), we see a pattern that repeats every three layers that corresponds to the *Residual Bottlenecks*. For the WFM (Fig. 6.9a), we see that the initial layers, which extract low level features appear less sensitive (large blue bars).



(a) Weights *(b) Activations*

Figure 6.9: Impact of Single Bit-Flip on Different Layers

6.2.2 Impact of Multiple Bit-Flip

A second fault injection campaign was performed where 20,000 randomly selected faults (1 to 10 bit-flips for each). The aggregate results are presented in Tab. 6.3. The results are comparable with single-bit fault model seen in Sec.6.2.1. Few faults are fully masked and the most common outcome is a *No Impact SDC* Between 0.9% and 7.9% of the faults produce *Tolerable SDCs*. In line with the literature as described in Sec.2.2.1, faults in the weights have a more significant impact, as each weight is used many times, and 9.4% of the weight faults produce *critical SDCs*.

Fault Model	Masked	No Impact SDC	Tolerable SDC	Critical SDC
AFM	32.75%	63.55%	0.924%	2.774%
WFM	0.364%	82.35%	7.851%	9.430%

Table 6.3: Aggregate Results of Fault Injection (multi-bit fault model)

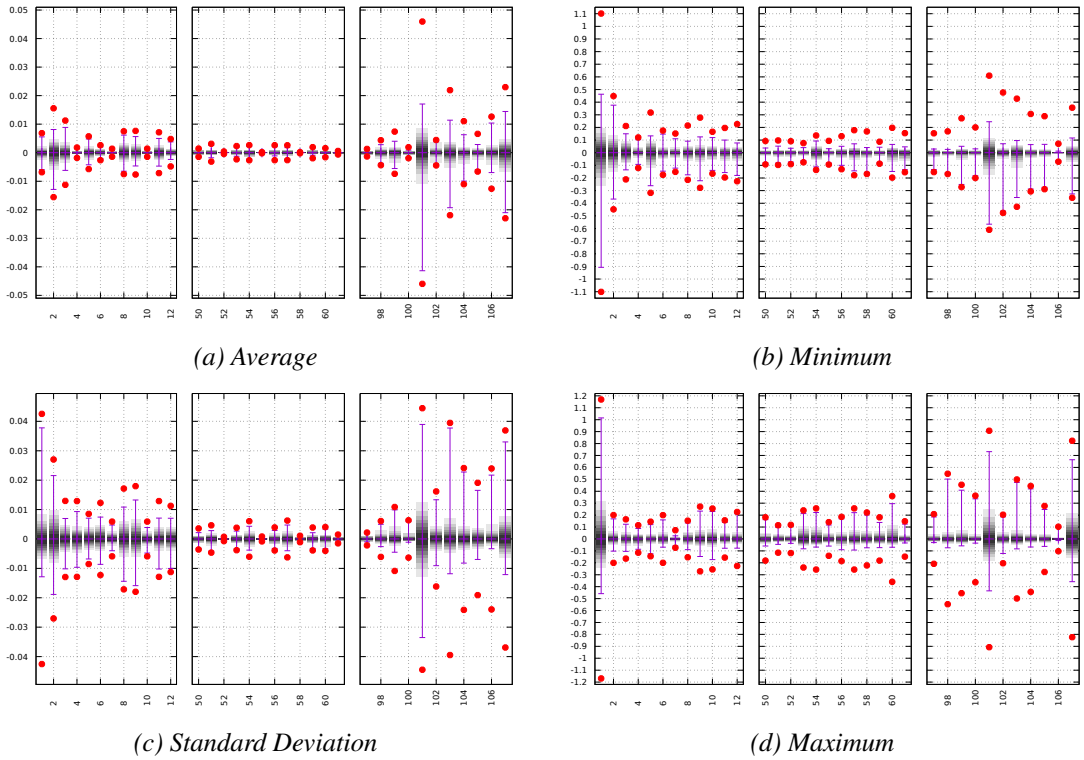
6.3 Symptom Based Error Detection Technique

As we have seen in Sec.2.3.1, the idea of SBED is to identify an incorrect behavior based on a symptom that is abnormal. This concept has been applied to micro-processors [Hari2012] and Li [Li2017] proposed an extension to DNNs. Authors [Hong2019, Hoang2020], including Li, report that faults that produce extremely large activation values are the ones most likely to result in errors. Li proposed a SBED technique where a fault detection threshold is set for each layer, based on the maximal neuronal value of the layer observed when executing the training data-set. Li arbitrarily adds 10% to this threshold. With this approach, with a 32-bit fp numeric format, he reports a false-positive rate of $\approx 3\%..5\%$ (called *precision* in the paper) and a SDC detection rate of $\approx 92\%..97\%$. The detection rate is good but the false positive rate is not acceptable for real-world applications. Our focus was to find a SBED technique based on a systematic approach and that achieves a false positive rate of zero.

6.3.1 Single Statistic Approach

In DeepLabV3+, all the layers (except the final one) are trained with batch normalization, thus, in the absence of faults, we expect the activations to be normally distributed. We started by analyzing the statistics of the activations with the training data-set. We identified four statistics of interest : minimum value (MIN), average value (AVG), maximum value (MAX) and standard-deviation (STD). We calculated these statistics, for each layer and for each image in the training sequence. This creates a distribution of values based on the input images and the results are plotted in Fig.6.10. We show the distribution of the values using shades of gray (10^{th} to 90^{th} percentiles) and the error bars show the extreme values. The activations are analyzed prior to the ReLU, thus the negative values.

Due to the large number of layers, we broke the graphs into three sections. Layers 1 to 11 correspond to the first part of the encoder. Low level features from the 11^{th} layer are directly connected to the decoder. In the middle of the ResNet-101 encoder, a repeated pattern can be observed, two layers with broad distributions, followed by a layer with a narrow distribution, corresponding to the 3-layer *Residual Bottlenecks*. The layer with a narrow distribution is the last 1x1 convolution. Finally, layers 101 to 109 are the decoder layers. For most of the layers, the extreme values are outlined well beyond the 90^{th} percentile.



(a) Average *(b) Minimum*

(c) Standard Deviation *(d) Maximum*

Figure 6.10: Statistical Distribution of Activation Values by Layer

Note that the variation of the distributions varies significantly across layers. For each statistic, and each layer, we propose to fix an error detection threshold that is *one standard-deviation beyond the minimum or maximal value observed during the training set*. This allows us to have a better detection when the distribution is tight and reduces the risk of false positives when the distribution is wide. Our calculated error detection thresholds are shown with the red dots.

In Tab. 6.4, we present the SBED detection results using each of the four statistics independently, both for the AFM (blue) and WFM (yellow). We declare that a fault is detected if a neuronal value exceeds the threshold (red dots). The results are quite promising. In all cases, well over 99% of the *Critical SDCs* are detected. The false positive rate is well below 1%, except in the case of the MAX statistic. However, in a real application, even this low rate of false positives would be unacceptable.

Statistic	False Positive (FP)	Fault Model	Tolerable SDC Detection (TSDC)	Critical SDC Detection (CSDC)
AVG	0.55%	AFM	14.7%	99.8%
		WFM	27.1%	99.3%
MIN	0.36%	AFM	16.2%	99.8%
		WFM	35.6%	99.4%
MAX	1.15%	AFM	58.1%	99.8%
		WFM	13.0%	99.7%
STD	0.24%	AFM	14.1%	99.8%
		WFM	28.0%	99.3%

Table 6.4: Fault Detection Capability with Single Statistic

6.3.2 Double Statistic Approach

To improve on these results, we decided to combine the statistics and only declare an error when at least **two** different statistics are out of range. The results are presented in Tab. 6.5. The results using the AFM are shown in the blue cells (upper right) while those for the WFM are shown in the yellow cells (lower left). For each pair of statistics, we show the false positive rate (FP), the detection rate for Tolerable Silent Data Corruption (TSDC)s and the detection rate for Critical Silent Data Corruption (CSDC)s. We observe that if we use the combination of AVG and MIN (bold text), then for both fault models we have no false positives and we have a detection rate of critical SDCs that is over 99.4%. In terms of critical fault detection rate, these results are in-line with the expectations of safety standards, while also meeting the expectations of vehicle manufacturers of having zero false positives.

Statistic	Metric	AVG	MIN	MAX	STD
AVG	FP		**0.0%**	0.0%	0.18%
	TSDC		**13.6%**	11.1%	14.1%
	CSDC		**99.8%**	99.8%	99.8%
MIN	FP	**0.0%**		0.0%	0.0%
	TSDC	**25.6%**		13.6%	13.7%
	CSDC	**99.4%**		99.8%	99.8%
MAX	FP	0.0%	0.0%		0.06%
	TSDC	3.39%	3.15%		11.1%
	CSDC	99.3%	99.3%		99.8%
STD	FP	0.18%	0.0%	0.06%	
	TSDC	26.8%	26.5%	3.63%	
	CSDC	99.4%	99.4%	99.3%	

Table 6.5: Fault Detection Capability with Two Statistics

6.3.3 Implementation and Overhead

The calculation of the statistics required to implement the techniques that we have presented can easily be implemented in software on a platform such as a GPU. They can also be calculated on-the-fly in hardware on a dedicated accelerator[1]. To calculate all statistics, for all layers, and to compare the output of each neuron with two thresholds requires a total of 755M FLOPs, a compute overhead of only 0.2%, compared to the total number of FLOPs to evaluate the network.

6.4 Fault Mitigation

In the previous section, we proposed a *fault detection* technique. To reduce the impact of faults, we now propose a *fault mitigation* technique which masks faults. Similar to ClipAct [Hoang2020], the ReLU activation function is modified so that if the input value is above a threshold, then the output is zero. In this way, extreme values can not propagate. In ClipAct, the algorithm for selecting the threshold is complex, requiring fault injections. In their paper, they demonstrate that the technique is effective for the small CIFAR-10 data-set.

[1] Standard deviation can be computed on the fly as $\sigma = \sqrt{\frac{\sum x^2}{N} - (\frac{\sum x}{N})^2}$

We propose to use the same threshold for mitigation as we used for fault detection with the MAX statistic. This threshold is set, per layer, one standard deviation above the maximum observed on the training data-set. If any activation value (prior to ReLU) exceeds this threshold, it is replaced with a zero. The impact of this technique on fault-free accuracy, as measured with the mIoU [Fernandez-Moral2018] is so small, that it can barely be measured (63.77255% versus 63.77253%), which is to be expected because the thresholds are set to be beyond the extreme values seen during the training data-set.

With this mitigation technique in place, over 99% of the critical faults are transformed into either *Masked*, *No Impact* or *Tolerable SDCs*, as shown in Tab. 6.6. In this table, the rows show the classification of the fault before mitigation, and the columns show how the image is classified after the mitigation (clipped ReLU). For example, with the AFM, 95.286% of the *critical SDCs* (CSDCs) become *No Impact SDCs* after mitigation.

Before \ After	Fault	Masked	No Impact	TSDC	CSDC
Masked	AFM	98.374%	0.6702%	0.9554%	0%
No Impact		0.0267%	99.074%	0.8893%	0%
TSDC		1.0101%	54.040%	44.949%	0%
CSDC		**3.3670%**	**95.286%**	**1.1785%**	0.1684%
Masked	WFM	98.717%	1.2821%	0%	0%
No Impact		0%	99.025%	0.9754%	0%
TSDC		0%	5.8894%	94.051%	0.0595%
CSDC		**0.0005%**	**58.742%**	**40.366%**	0.8420%

Table 6.6: Fault Mitigation Effect

We did, however, notice a very small number of cases (0.0595% shown in red) where a fault in the weights was originally classified as tolerable (TSDC) but with our mitigation technique it became critical (CSDC), which is obviously undesirable. We studied all the images where this occurred and concluded that these correspond to borderline cases. The difference in the critical and non-critical image was always very minor. In Fig. 6.11 we show the most visible example. In this case, a pixel appeared in a class that was not previously present, thus the image after mitigation was classified as critical. In total, only 3 pixels (0.015%) were modified. It is important to note, our detection technique identified the presence of this fault, even though the mitigation could not render it non-critical.

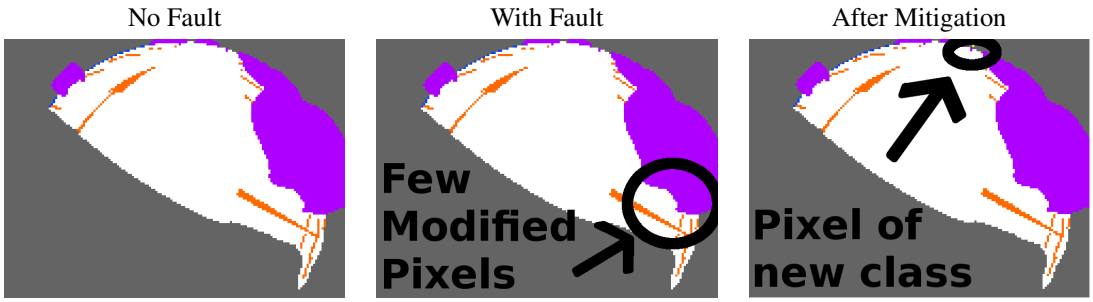


Figure 6.11: Borderline Non-Critical Fault Becoming Critical

6.5 Comparison with Related Works

Draghetti [Draghetti2019] proposed a fault detection technique for object detection for automotive image processing. The technique is based on measuring the difference (mean square error) between two consecutive frames and comparing this with the extent of change in the resulting bounding boxes. This is a form of temporal redundancy, and may not be well suited to detecting permanent faults which could corrupt every frames. This technique did not achieve a zero false positive rate and was not demonstrated for semantic segmentation.

In dos Santos [dos Santos2018] they propose several detection techniques. Their *Reliable Max pooling* technique is based on a threshold for the Max Pooling layers that they arbitrarily set at 10x the maximum value observed during the training. They report a 98% detection rate (compared to 99.4% in this work). Their network has many Max Pooling layers, but DeepLabV3+ has a single Max Pooling layer, so their technique is not directly applicable. Their mitigation strategy is to replace the output of the Max Pooling layer with the second largest value but only $\approx 85\%$ of the critical faults are mitigated, whereas our technique masks more than 99% of the critical faults.

Hoang [Hoang2020] proposed an error mitigation technique based on a clipped Rectified Linear Unit activation function and they provide an algorithm to optimally set the threshold. This technique provides error masking but it does not provide fault detection, a requirement in safety standards. The selected threshold could be below the maximum activation value in the training set, thus the error-free behavior of the network is modified. Finally, they present results for CIFAR-10, and it is not clear that their threshold selection scales to larger applications.

The techniques proposed in this chapter achieve high detection (99.4%) with no false positives (unlike [Draghetti2019]). The proposed technique is compatible with any layers which use batch-normalization (unlike in [dos Santos2018], which is restricted to Max Pooling Layers). Finally, our technique require no fault injections, only the one-time calculation of statistics from the training data-set. Taken together, these represents a significant improvement over the state-of-the-art in SBED for DNNs.

6.6 Perspective

The methodology presented in this chapter only considers limited fault models on a single floating-point based DNN. Future work could improve the presented technique in three ways :

1) Quantized DNNs are preferred for applications that require robustness, as discussed in Sec.2.2.1. Yet, to the best of our knowledge no SBED approach has proven to be effective on quantized DNNs. With a quantized numeric format, it is more likely that a bit-flip will result in an overflow rather than an extremely large value. Thus it would be of scientific interest to test our technique on such quantized models.

2) In this study, we focus on a limited fault model (1 to 10 bit-flips in weights and activations). Other fault models could be explored, and the technique could be tested on a real hardware platform, such as a GPU.

3) We only focused on DeepLabV3+ trained for the Woodscape data-set. Demonstration of our methods on other network topologies and data-sets would indicate that our technique can scale to other case study. We expect that the technique can be applied to networks which have layers where batch-normalization is applied.

6.7 Conclusion of the Syndrome-Based Error Detection

We have performed a fault-injection study of DNNs performing semantic segmentation and showed how fault effects can be classified as critical and tolerable SDCs. Using a state-of-the-art network and an industrial data-set, we analyzed the sensitivity of the network for two fault models and found that between 3% to 9% of faults produce *critical SDCs*.

We analyzed four statistics (AVG,MIN,MAX and STD) and using this analysis, set thresholds for detecting anomalous behavior. This approach is different from previous authors [Li2017, dos Santos2018], who used only the MAX and set a threshold arbitrarily. By considering the variation of the activations for each layer, our approach can be applied to other applications. Using multiple statistics to trigger error detection, we can eliminate false positives, as demonstrated on the Woodscape data-set, while maintaining a critical fault detection over 99.4%. We also proposed to combine our error detection technique with a zero-masking technique (similar to [Hoang2020]), but with a simpler approach for setting the threshold. This mitigation technique allows us to mask more than 99% of the critical faults. Taken together, the contributions in this chapter provide a methodology for analyzing the impact of faults in image segmentation applications, providing the level of fault detection required by safety standards and a technique for masking nearly all critical faults.

7
Conclusions

This chapter concludes this manuscript. We briefly review the contributions presented in this thesis, and present ideas for future work.

The work undertaken in this thesis builds on existing background in the literature, as the fact that Deep Neural Network (DNN)s are tolerant to certain types of faults had already been well established. For example, previous works have shown that faults resulting in high numerical divergence are likely to impact the final result, however, this type of fault can be detected by a statistical analysis of the dataflow.

We presented the current scientific literature in the field of DNN fault tolerance and split these works into two parts. Many studies simply analyze the intrinsic fault tolerance of DNNs, while others attempt to improve the robustness. In the literature, we note that authors analyze a single platform with their selected fault model, but there were few studies which compare different platforms, using the same fault model and test cases.

The key observation of the works analyzing the fault tolerance of DNNs is that the most impacting faults are those resulting in a high numerical divergence of neurons or weights. This fact has frequently been exploited in the works focused on improving fault tolerance by restricting the numerical range, for example by adjusting the numerical format through quantization or through truncated activation functions. Such techniques can improve the tolerance to bit errors in activations and weights by several orders of magnitude. For networks which are already quantified, further improvements in fault tolerance are more difficult to achieve.

To detect hardware faults, many techniques rely on adapting well-known strategies traditionally used in general purpose computing to DNNs, such as Syndrome-Based Error Detection or Algorithm-Based Fault Tolerance. Once faults have been detected, masking erroneous values with zeros is a common and effective technique to mitigate the impact of the faults on the accuracy of the network.

Building on the results of these existing works, we proposed several new contributions. First, we performed an in-depth analysis of the fault tolerance of several systolic architectures, and drew conclusions on how data-flow impacts the propagation of faults. Specifically, our results show that an architecture, such as the Output Stationary dataflow, that limits the spatial propagation of faults is more robust.

We used these observations to design a fault tolerant systolic accelerator. The proposed architecture, coined MOZART+, combines an Output Stationary architecture, on-line testing, fault tolerant scheduling, fault tolerant training and masking of arithmetic errors with zeros. Our experiments showed that with our test cases, MOZART+ was able to quickly detect all the impacting Stuck-at-1 faults. We showed that the reduction in accuracy in the presence of faulty Processing Elements is gradual and MOZART+ drastically limits the worst-case impact. This is achieved with a hardware overhead of approximately 8%.

While MOZART+ focuses on the arithmetic unit, our next contribution, the Opportunistic Parity technique, focuses on transient faults altering the weights of DNNs. Our initial studies, showed that DNNs are robust to changes in the Least Significant Bits (LSBs) of their weights. Using this idea, we proposed to encode a parity bit in the LSB of the weights, thus introducing virtually no storage overhead. When combined with the zero-masking technique to mask the weights with erroneous parity, on floating-point based DNNs, an improvement in bit error rate tolerance between one to three orders of magnitude was achieved with practically no-loss in fault-free accuracy. For a 8-bit integer numeric format, the technique is less effective but can still provide approximately a 5x improvement, with a minor loss in fault free accuracy.

These two techniques are limited to faults in arithmetic units and faults in the weights, so in order to address a broader scope of faults, we proposed a syndrome based approach to fault detection, adapted to DNNs. This method is a statistics-based technique to detect faults occurring anywhere in a DNN by monitoring its data-flow. We analyzed four statistics (AVeraGe,MINimum,MAXimum and STandard Deviation) on the fault-free data-flow to establish thresholds for detecting anomalous behavior. We tested our method on a dataset for autonomous driving, and our results show that using multiple statistics to trigger error detection, we can eliminate false positives while maintaining a critical fault detection rate over 99.4%. We then combined this tech-

nique with the zero-masking strategy, allowing us to mask more than 99% of the critical faults. To our knowledge, this was the first work to address fault mitigation for image segmentation applications, one of the key parts of autonomous driving systems.

Some topics related to the fault tolerance of DNNs have barely been explored. In existing works, the fault tolerance of a platform is evaluated using costly fault injection campaigns. In this thesis, we developed a technique to accelerate fault injections in DNNs (fault calendar described in Appendix A), but there is still a need to develop algorithms and methods to quickly predict the fault tolerance of a DNN on a hardware platform, without resorting to fault injection, and this problem was not addressed in this thesis. Also, the fault tolerance of other types of DNNs, such as recurrent neural networks and transformers has not been explored in the existing works. Both these topics merit further in-depth investigation.

In addition, there are many topics which have been only been partially explored. For example, we proposed an analysis of systolic accelerators, but we believe that a quantitative exploration of other architectures such as Single Instruction Multiple Data-type and those inspired from digital signal processor data-paths is very important. In our study of MOZART+, we only considered faults in arithmetic units. An exhaustive study of all possible faults (e.g. control path, communication channels, ...) is needed and would help identify which fault mitigation techniques provide the most coverage.

Towards the end of this thesis, we identified opportunities to improve our proposed techniques. For example, our OP technique is effective but could be significantly enhanced. More specifically, we could consider to code an Error Correction Code (ECC) in the unimportant bits of the weights. Furthermore, if the constraint of the parity or ECC were imposed in parallel with the training phase, it is possible that the loss in fault free precision could be eliminated. Regarding the syndrome based techniques, in our work, we simply decoded whether or not a fault had occurred. We believe that an in-depth analysis of the signatures could localize the fault. We believe that further work on both these techniques could yield further improvements.

Some of the techniques proposed in this thesis improved the fault tolerance of DNNs evaluated with a floating-point numeric format. Quantized DNNs are intrinsically more robust, however, identifying new fault tolerance techniques for quantized DNNs is a challenging problem. Due to the quantization, there is less redundancy and the numeric range of values is already fully utilized, leaving less opportunity for detecting anomalies. We note that the on-line test proposed with the MOZART+ does indeed detect faults, regardless of the numeric format.

As we have seen, DNNs are increasingly being used in mission critical applications and there is a need to ensure fault tolerance, with a minimal hardware overhead. In this thesis, there were three major contributions: a fault-tolerant systolic architecture, a technique for error detection in weight storage memories and a syndrome-based error detection method. It is important to note that all three of these techniques introduce minimal overhead and thus are highly attractive for commercial DNN accelerators that must be fault tolerant. For society to adopt artificial intelligence, it must be trustable. Hardware fault tolerance is one of many of the required elements to build trust. The contributions to fault tolerance in this thesis will help build reliable systems integrating artificial intelligence.

Bibliography

[Abbasi2022] Alireza Abbasi, Farbod Setoudeh, Mohammad Bagher Tavakoli and Ashkan Horri. *A novel design of high performance and robust ultra-low power SRAM cell based on memcapacitor*. Nanotechnology, vol. 33, no. 16, 2022.

[Abdullah Hanif2020] Muhammad Abdullah Hanif and Muhammad Shafique. *Salvagednn: salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping*. Philosophical Transactions of the Royal Society A, vol. 378, no. 2164, 2020.

[Adam2021a] Khalid Adam, Izzeldin Ibrahim Mohamed and Younis Ibrahim. *A selective mitigation technique of soft errors for dnn models used in healthcare applications: Densenet201 case study*. IEEE Access, vol. 9, 2021.

[Adam2021b] Khalid Adam, Izzeldin I Mohd and Younis M Younis. *The impact of the soft errors in convolutional neural network on GPUs: Alexnet as case study*. Procedia Computer Science, vol. 182, pages 89–94, 2021.

[Alippi1995] C. Alippi, V. Piuri and M. Sami. *Sensitivity to errors in artificial neural networks: a behavioral approach*. IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, vol. 42, no. 6, 1995.

[Arechiga2018] Austin P Arechiga and Alan J Michaels. *The robustness of modern deep learning architectures against single event upset errors*. In 2018 IEEE High Performance extreme Computing Conference (HPEC). IEEE, 2018.

[Azizimazreah2018] Arash Azizimazreah, Yongbin Gu, Xiang Gu and Lizhong Chen. *Tolerating soft errors in deep learning accelerators with reliable on-chip memory designs*. In 2018 IEEE International Conference on Networking, Architecture and Storage (NAS). IEEE, 2018.

[Badia2021] Jose M. Badia, German Leon, Jose A. Belloch, Mario Garcia-Valderas, Almudena Lindoso and Luis Entrena. *Comparison of parallel implementation strategies in GPU-accelerated System-on-Chip under proton irradiation*. IEEE Transactions on Nuclear Science, 2021.

[Bichler2017] Olivier Bichler. *N2D2 DNN Framework*. In https://github.com/CEA-LIST/N2D2, 2017.

[Blaiech2019] Ahmed Ghazi Blaiech, Khaled Ben Khalifa, Carlos Valderrama, Marcelo AC Fernandes and Mohamed Hedi Bedoui. *A survey and taxonomy of FPGA-based deep learning accelerators*. Journal of Systems Architecture, vol. 98, 2019.

[Brown1960] David T Brown. *Error detecting and correcting binary codes for arithmetic operations*. IRE Transactions on Electronic Computers, no. 3, 1960.

[Burel2021a] Stéphane Burel, Adrian Evans and Lorena Anghel. *Accélérateur systolique de réseau de neurons and système électronique and procédé de test associés*. In Patent FR2105808, 2021.

[Burel2021b] Stéphane Burel, Adrian Evans and Lorena Anghel. *MOZART: Masking Outputs with Zeros for Architectural Robustness and Testing of DNN Accelerators*. In 2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS). IEEE, 2021.

[Burel2021c] Stéphane Burel, Adrian Evans and Lorena Anghel. *Zero-Overhead Protection for CNN Weights*. In 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). IEEE, 2021.

[Burel2022a] Stephane Burel and Adrian Evans. *Calendrier de fautes pour accélération de l'analyse de fiabilité des circuits intégrés*. In Patent FR2205817, 2022.

[Burel2022b] Stéphane Burel, Adrian Evans and Lorena Anghel. *Improving DNN Fault Tolerance in Semantic Segmentation Applications*. In 2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). IEEE, 2022.

[Burel2022c] Stéphane Burel, Adrian Evans and Lorena Anghel. *MOZART+: Masking Outputs with Zeros for Improved Architectural Robustness and Testing of DNN Accelerators*. IEEE Transactions on Device and Materials Reliability, 2022.

[Cardoso2017] Jorge Cardoso and Goetz Brasche. *Fault Injection System and Method of Fault Injection*. In Patent US2020301798A1, 2017.

[Chen2016] Yu-Hsin Chen, Tushar Krishna, Joel S Emer and Vivienne Sze. *Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks*. IEEE journal of solid-state circuits, vol. 52, no. 1, pages 127–138, 2016.

[Chen2018] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff and Hartwig Adam. *Encoder-Decoder with Atrous Separable Convolution for Semantic Image Segmentation*. In Proceedings of the European Conference on Computer Vision (ECCV), September 2018.

[Chen2021] Zitao Chen, Guanpeng Li and Karthik Pattabiraman. *A low-cost fault corrector for deep neural networks through range restriction*. In 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, 2021.

[Clay1992] Reed D Clay and Carlo H Sequin. *Fault tolerance training improves generalization and robustness*. In 1992 IJCNN International Joint Conference on Neural Networks, volume 1. IEEE, 1992.

[Collobert2008] Ronan Collobert and Jason Weston. *A unified architecture for natural language processing: Deep neural networks with multitask learning*. In Proceedings of the 25th international conference on Machine learning, 2008.

[Corneliou2021] P. Corneliou, P. Nikolaou, M. K. Michael and T. Theocharides. *Fine-Grained Vulnerability Analysis of Resource Constrained Neural Inference Accelerators*. In 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Los Alamitos, CA, USA, oct 2021. IEEE Computer Society.

[Couellan2021] Nicolas Couellan. *Probabilistic robustness estimates for feed-forward neural networks*. Neural Networks, vol. 142, 2021.

[Datta2020] Leonid Datta. *A survey on activation functions and their relation with xavier and he normal initialization*. arXiv preprint arXiv:2004.06632, 2020.

[Deng2013] Li Deng, Geoffrey Hinton and Brian Kingsbury. *New types of deep neural network learning for speech recognition and related applications: An overview*. In 2013 IEEE international conference on acoustics, speech and signal processing. IEEE, 2013.

[dos Santos2018] Fernando Fernandes dos Santos, Pedro Foletto Pimenta, Caio Lunardi, Lucas Draghetti, Luigi Carro, David Kaeli and Paolo Rech. *Analyzing and increasing the reliability of convolutional neural networks on GPUs*. IEEE Transactions on Reliability, vol. 68, no. 2, pages 663–677, 2018.

[dos Santos2019] Fernando Fernandes dos Santos, Philippe Navaux, Luigi Carro and Paolo Rech. *Impact of reduced precision in the reliability of deep neural networks for object detection*. In 2019 IEEE European Test Symposium (ETS). IEEE, 2019.

[dos Santos2021] Fernando Fernandes dos Santos, Siva Kumar Sastry Hari, Pedro Martins Basso, Luigi Carro and Paolo Rech. *Demystifying gpu reliability: comparing and combining beam experiments, fault simulation, and profiling*. In 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 289–298. IEEE, 2021.

[Draghetti2019] Lucas Klein Draghetti, Fernando Fernandes dos Santos, Luigi Carro and Paolo Rech. *Detecting Errors in Convolutional Neural Networks Using Inter Frame Spatio-Temporal Correlation*. In 2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS), pages 310–315, July 2019.

[Du2015] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen and Olivier Temam. *ShiDianNao: shifting vision processing closer to the sensor*. pages 92–104. ACM Press, 2015.

[Ernst2004] Dan Ernst, Shidhartha Das, Seokwoo Lee, David Blaauw, Todd Austin, Trevor Mudge, Nam Sung Kim and Krisztián Flautner. *Razor: circuit-level correction of timing errors for low-power operation*. IEEE Micro, vol. 24, no. 6, 2004.

[Everingham2009] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John M. Winn et Andrew Zisserman. *The Pascal Visual Object Classes (VOC) Challenge*. International Journal of Computer Vision, vol. 88, 2009.

[Feng2019] Xin Feng, Youni Jiang, Xuejiao Yang, Ming Du and Xin Li. *Computer vision algorithms and hardware implementations: A survey*. Integration, vol. 69, pages 309–320, 2019.

[Fernandez-Moral2018] Eduardo Fernandez-Moral, Renato Martins, Denis Wolf and Patrick Rives. *A New Metric for Evaluating Semantic Segmentation: Leveraging Global and Contour Accuracy*. In 2018 IEEE Intelligent Vehicles Symposium (IV), 2018.

[Fratin2018] Vinicius Fratin, Daniel Oliveira, Caio Lunardi, Fernando Santos, Gennaro Rodrigues and Paolo Rech. *Code-dependent and architecture-dependent reliability behaviors*. In 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 13–26. IEEE, 2018.

[Fu2021] Hsiao-Wen Fu, Ting-Yu Chen, Cheng-Di Tsai, Meng-Wei Shen and Tsung-Chu Huang. *AN-Coded Redundant Residue Number System for Reliable Neural Networks*. In 2021 IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW). IEEE, 2021.

[Gal2016] Yarin Gal and Zoubin Ghahramani. *Dropout as a bayesian approximation: Representing model uncertainty in deep learning*. In international conference on machine learning. PMLR, 2016.

[Gambardella2019] Giulio Gambardella, Johannes Kappauf, Michaela Blott, Christoph Doehring, Martin Kumm, Peter Zipf and Kees Vissers. *Efficient Error-Tolerant Quantized Neural Network Accelerators*. In 2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT). IEEE, 2019.

[Ganapathy2019] Shrikanth Ganapathy, John Kalamatianos, Bradford M. Beckmann, Steven Raasch et Lukasz G. Szafaryn. *Killi: Runtime Fault Classification to Deploy Low Voltage Caches without MBIST*. In 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2019.

[Gao2020] Zhen Gao, Xiaohui Wei, Han Zhang, Wenshuo Li, Guangjun Ge, Yu Wang and Pedro Reviriego. *Reliability Evaluation of Pruned Neural Networks against Errors on Parameters*. In 2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2020.

[Gao2022] Zhen Gao, Han Zhang, Yi Yao, Jiajun Xiao, Shulin Zeng, Guangjun Ge, Yu Wang, Anees Ullah and Pedro Reviriego. *Soft Error Tolerant Convolutional Neural Networks on FPGAs With Ensemble Learning*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2022.

[Ghavami2021] Behnam Ghavami, Mani Sadati, Zhenman Fang and Lesley Shannon. *FitAct: Error Resilient Deep Neural Networks via Fine-Grained Post-Trainable Activation Functions*. arXiv preprint arXiv:2112.13544, 2021.

[Goldstein2021] Brunno F Goldstein, Victor C Ferreira, Sudarshan Srinivasan, Dipankar Das, Alexandre S Nery, Sandip Kundu and Felipe MG França. *A lightweight error-resiliency mechanism for deep neural networks*. In 2021 22nd International Symposium on Quality Electronic Design (ISQED). IEEE, 2021.

[Gu2019] Xiaofeng Gu, Qingqing Li and Zhiguo Yu. *A software and hardware collaborative acceleration method and system for fault injection*. In Patent CN109947609A, 2019.

[Guan2019] Hui Guan, Lin Ning, Zhen Lin, Xipeng Shen, Huiyang Zhou and Seung-Hwan Lim. *In-place zero-space memory protection for cnn*. Advances in Neural Information Processing Systems, vol. 32, 2019.

[Hari2012] Siva Kumar Sastry Hari, Sarita V. Adve and Helia Naeimi. *Low-cost program-level detectors for reducing silent data corruptions*. In IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012), 2012.

[Hari2017] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler et Joel Emer. *Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation*. In 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2017.

[Hari2021] Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai and Stephen W Keckler. *Making convolutions resilient via algorithm-based error detection techniques*. IEEE Transactions on Dependable and Secure Computing, 2021.

[He2015] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. Deep Residual Learning for Image Recognition. 2015. arXiv:1512.03385.

[He2016] Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun. *Deep Residual Learning for Image Recognition*. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 770–778, Las Vegas, NV, USA, June 2016. IEEE.

[He2021] Shaoming He, Hyo-Sang Shin and Antonios Tsourdos. *Computational missile guidance: a deep reinforcement learning approach*. Journal of Aerospace Information Systems, vol. 18, no. 8, 2021.

[Hills2014] Gage Hills, Max Shulaker, Hai Wei, Hong-Yu Chen, H-S Philip Wong and Subhasish Mitra. *Robust design and experimental demonstrations of carbon nanotube digital circuits*. In Proceedings of the IEEE 2014 Custom Integrated Circuits Conference. IEEE, 2014.

[Hoang2020] Le-Ha Hoang, Muhammad Abdullah Hanif and Muhammad Shafique. *FT-ClipAct: Resilience Analysis of Deep Neural Networks and Improving their Fault Tolerance using Clipped Activation*. In Design, Automation Test in Europe (DATE), 2020.

[Hong2019] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida and Tudor Dumitraş. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks. USENIX, 2019.

[Horowitz2014] Mark Horowitz. *1.1 Computing's energy problem (and what we can do about it)*. In 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014.

[Howard2017] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. 2017. arXiv:1704.04861.

[Howard2019] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan*et al. Searching for mobilenetv3*. In Proceedings of the IEEE/CVF international conference on computer vision, 2019.

[Huang1984] Kuang-Hua Huang and Jacob A Abraham. *Algorithm-based fault tolerance for matrix operations*. IEEE transactions on computers, vol. 100, no. 6, 1984.

[Iandola2016] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally and Kurt Keutzer. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size*. arXiv preprint arXiv:1602.07360, 2016.

[Ibrahim2020] Younis Ibrahim, Haibin Wang, Junyang Liu, Jinghe Wei, Li Chen, Paolo Rech, Khalid Adam and Gang Guo. *Soft errors in DNN accelerators: A comprehensive review*. Microelectronics Reliability, vol. 115, page 113969, December 2020.

[IDTechEx2022] IDTechEx. Autonomous cars, robotaxis and sensors 2022-2042. 2022.

[Intel] Intel. *3rd Generation Intel Xeon Scalable Processors*. In https://ark.intel.com/content/www/us/en/ark/products/series/204098/3rd-generation-intel-xeon-scalable-processors.html.

[Ioffe2015] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. CoRR, vol. abs/1502.03167, 2015.

[ISO-26262.18] ISO-26262. *Road vehicles – Functional safety*, 2018.

[Ito1997] Takehiro Ito and Itsuo Takanami. *On fault injection approaches for fault tolerance of feedforward neural networks*. In Proceedings Sixth Asian Test Symposium (ATS'97). IEEE, 1997.

[Ito2021] Kojiro Ito, Yangchao Zhang, Hiroaki Itsuji, Takumi Uezono, Tadanobu Toba et Masanori Hashimoto. *Analyzing DUE Errors on GPUs With Neutron Irradiation Test and Fault Injection to Control Flow*. IEEE Transactions on Nuclear Science, vol. 68, no. 8, 2021.

[Kalamkar2019] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul and Pradeep Dubey. *A Study of BFLOAT16 for Deep Learning Training*, 2019. arXiv:1905.12322.

[Khoshavi2020] Navid Khoshavi, Arman Roohi, Connor Broyles, Saman Sargolzaei, Yu Bi and David Z Pan. *Shieldenn: Online accelerated framework for fault-tolerant deep neural network architectures*. In 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020.

[Kim1989] J. H. Kim and S. M. Reddy. *On the design of fault-tolerant two-dimensional systolic arrays for yield enhancement*. IEEE Transactions on Computers, vol. 38, no. 4, 1989.

[Kim2018] Sung Kim, Patrick Howe, Thierry Moreau, Armin Alaghi, Luis Ceze and Visvesh Sathe. *MATIC: Learning around errors for efficient low-voltage neural network accelerators*. In 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018.

[Kim2021] Youngbae Kim, Shreyash Patel, Heekyung Kim, Nandakishor Yadav and Kyuwon Ken Choi. *Ultra-low power and high-throughput SRAM design to enhance ai computing ability in autonomous vehicles*. Electronics, vol. 10, no. 3, 2021.

[Kosaian2021] Jack Kosaian and KV Rashmi. *Arithmetic-intensity-guided fault tolerance for neural network inference on gpus*. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021.

[Krizhevsky2012] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. *ImageNet Classification with Deep Convolutional Neural Networks*. In Proceedings of the 25th International Conference on Neural Information Processing Systems, NIPS'12, 2012.

[Kumar2021] Harekrishna Kumar and VK Tomar. *A review on performance evaluation of different low power SRAM cells in nano-scale era*. Wireless Personal Communications, vol. 117, no. 3, 2021.

[Kwon2016] Sangheon Kwon, Kyungmin Lee, Yoonsoo Kim, Kyungah Kim, Changmin Lee and Won Woo Ro. *Measuring error-tolerance in SRAM architecture on hardware accelerated neural network*. In 2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), pages 1–4. IEEE, 2016.

[LeCun2015] Yann LeCun, Yoshua Bengio and Geoffrey Hinton. *Deep learning*. Nature, vol. 521, no. 7553, pages 436–444, May 2015.

[Li2017] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer and Stephen W. Keckler. *Understanding error propagation in deep learning neural network (DNN) accelerators and applications*. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM Press, 2017.

[Li2019] Li Li, Dawen Xu, Kouzi Xing, Cheng Liu, Ying Wang, Huawei Li and Xiaowei Li. *Squeezing the Last MHz for CNN Acceleration on FPGAs*. In 2019 IEEE International Test Conference in Asia (ITC-Asia), pages 151–156. IEEE, 2019.

[Li2020a] Wenshuo Li, Guangjun Ge, Kaiyuan Guo, Xiaoming Chen, Qi Wei, Zhen Gao, Yu Wang and Huazhong Yang. *Soft error mitigation for deep convolution neural network on FPGA accelerators*. In 2020 2nd IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS). IEEE, 2020.

[Li2020b] Wenshuo Li, Xuefei Ning, Guangjun Ge, Xiaoming Chen, Yu Wang and Huazhong Yang. *FTT-NAS: Discovering Fault-Tolerant Neural Architecture*. pages 211–216. IEEE, January 2020.

[Libano2020] F. Libano, B. Wilson, M. Wirthlin, P. Rech and J. Brunhaver. *Understanding the Impact of Quantization, Accuracy, and Radiation on the Reliability of Convolutional Neural Networks on FPGAs*. IEEE Transactions on Nuclear Science, vol. 67, no. 7, 2020.

[Libano2021] F Libano, P Rech, B Neuman, J Leavitt, M Wirthlin and J Brunhaver. *How reduced data precision and degree of parallelism impact the reliability of convolutional neural networks on fpgas*. IEEE Transactions on Nuclear Science, vol. 68, no. 5, 2021.

[Liu2018] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang and Trevor Darrell. *Rethinking the value of network pruning*. arXiv preprint arXiv:1810.05270, 2018.

[Liu2021] Cheng Liu, Cheng Chu, Dawen Xu, Ying Wang, Qianlong Wang, Huawei Li, Xiaowei Li and Kwang-Ting Cheng. *HyCA: A Hybrid Computing Architecture for Fault Tolerant Deep Learning*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2021.

[Mahdiani2012] H. R. Mahdiani, S. M. Fakhraie and C. Lucas. *Relaxed Fault-Tolerant Hardware Implementation of Neural Networks in the Presence of Multiple Transient Errors*. IEEE Trans. on Neural Networks and Learning Systems, August 2012.

[Mahmoud2021] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W Fletcher, Sarita V Adve, Charbel Sakr, Naresh Shanbhag, Pavlo Molchanov, Michael B Sullivan, Timothy Tsai and Stephen W Keckler. *Optimizing Selective Protection for CNN Resilience*. In 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). IEEE Computer Society, 2021.

[Malekzadeh2021] Elaheh Malekzadeh, Nezam Rohbani, Zhonghai Lu and Masoumeh Ebrahimi. *The Impact of Faults on DNNs: A Case Study*. In 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2021.

[Matsubara2021] Katsushige Matsubara, Lieske Hanno, Motoki Kimura, Atsushi Nakamura, Manabu Koike, Kazuaki Terashima, Shun Morikawa, Yoshihiko Hotta, Takahiro Irita, Seiji Mochizuki, Hiroyuki Hamasaki and Tatsuya Kamei. *A 12nm Autonomous-Driving Processor with 60.4TOPS, 13.8TOPS/W CNN Executed by Task-Separated ASIL D Control*. In 2021 IEEE International Solid- State Circuits Conference, volume 64, 2021.

[McCarthy2006] John McCarthy, Marvin L Minsky, Nathaniel Rochester and Claude E Shannon. *A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955*. AI magazine, vol. 27, no. 4, pages 12–12, 2006.

[Micikevicius2018] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh and Hao Wu. *Mixed Precision Training*, 2018. arXiv:1710.03740.

[Mittal2020] Sparsh Mittal. *A survey on modeling and improving reliability of DNN algorithms and accelerators*. Journal of Systems Architecture, vol. 104, page 101689, March 2020.

[Moradi2020] Reza Moradi, Reza Berangi and Behrouz Minaei. *A survey of regularization strategies for deep models*. Artificial Intelligence Review, vol. 53, no. 6, 2020.

[Motaman2019] Seyedhamidreza Motaman, Swaroop Ghosh and Jongsun Park. *A Perspective on Test Methodologies for Supervised Machine Learning Accelerators*. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 9, no. 3, September 2019.

[Mukherjee2005a] Shubhendu S Mukherjee, Joel Emer and Steven K Reinhardt. *The soft error problem: An architectural perspective*. In 11th International Symposium on High-Performance Computer Architecture. IEEE, 2005.

[Mukherjee2005b] S.S. Mukherjee, J. Emer and S.K. Reinhardt. *The Soft Error Problem: An Architectural Perspective*. pages 243–247. IEEE, 2005.

[Nagel2019] Markus Nagel, Mart Van Baalen, Tijmen Blankevoort and Max Welling. *Data-Free Quantization Through Weight Equalization and Bias Correction*. pages 1325–1334. IEEE, October 2019.

[Neggaz2018] Mohamed A. Neggaz, Ihsen Alouani, Pablo R. Lorenzo and Smail Niar. *A Reliability Study on CNNs for Critical Embedded Systems*. In 2018 IEEE 36th International Conference on Computer Design (ICCD), 2018.

[Neggaz2019] Mohamed A Neggaz, Ihsen Alouani, Smail Niar and Fadi Kurdahi. *Are cnns reliable enough for critical applications? an exploratory study*. IEEE Design & Test, vol. 37, no. 2, 2019.

[Nguyen2019] Duy-Thanh Nguyen, Nhut-Minh Ho and Ik-Joon Chang. *St-DRC: Stretchable DRAM Refresh Controller with No Parity-overhead Error Correction Scheme for Energy-efficient DNNs*. In 2019 56th ACM/IEEE Design Automation Conference (DAC), 2019.

[Nijhuis1990] J. Nijhuis, B. Hofflinger, A. van Schaik and L. Spaanenburg. *Limits to the fault-tolerance of a feedforward neural network with learning*. In [1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium, June 1990.

[ONNX] ONNX. *Open Neural Network Exchange Model Zoo*. In https://github.com/onnx/models.

[Otter2021] Daniel W. Otter, Julian R. Medina and Jugal K. Kalita. *A Survey of the Usages of Deep Learning for Natural Language Processing*. IEEE Transactions on Neural Networks and Learning Systems, vol. 32, no. 2, 2021.

[Ozen2020a] Elbruz Ozen and Alex Orailoglu. *Boosting bit-error resilience of DNN accelerators through median feature selection*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 11, 2020.

[Ozen2020b] Elbruz Ozen and Alex Orailoglu. *Just Say Zero: Containing critical bit-error propagation in deep neural networks with anomalous feature suppression*. In 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). IEEE, 2020.

[Ozen2020c] Elbruz Ozen and Alex Orailoglu. *Low-cost error detection in deep neural network accelerators with linear algorithmic checksums*. Journal of Electronic Testing, vol. 36, no. 6, 2020.

[Park2015] Seongwook Park, Kyeongryeol Bong, Dongjoo Shin, Jinmook Lee, Sungpill Choi et Hoi-Jun Yoo. *4.6 A1.93TOPS/W scalable deep learning/inference processor with tetra-parallel MIMD architecture for big-data applications*. pages 1–3. IEEE, February 2015.

[Qiang2019] Liu Qiang and Li Bochao. *Low-cost state control method for hardware emulation of fault attack*. In Patent CN109581207A, 2019.

[Radosavovic2020] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He and Piotr Dollár. *Designing network design spaces*. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2020.

[Rao2022] M Damodhar Rao, YV Narayana and VVKDV Prasad. *Ultra Low Power Offering 14 nm Bulk Double Gate FinFET Based SRAM Cells*. Sustainable Computing: Informatics and Systems, 2022.

[Reagen2016a] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose Miguel Hernandez-Lobato, Gu-Yeon Wei and David Brooks. *Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators*. pages 267–278. IEEE, June 2016.

[Reagen2016b] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose Miguel Hernandez-Lobato, Gu-Yeon Wei and David Brooks. *Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators*. pages 267–278. IEEE, June 2016.

[Reagen2018] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks and Gu-Yeon Wei. *Ares: A framework for quantifying the resilience of deep neural networks*. IEEE, June 2018.

[Reuther2019] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi and Jeremy Kepner. *Survey and benchmarking of machine learning accelerators*. arXiv preprint arXiv:1908.11348, 2019.

[Reuther2021] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi and Jeremy Kepner. *AI Accelerator Survey and Trends*. In 2021 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2021.

[Ronneberger2015] Olaf Ronneberger, Philipp Fischer and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. CoRR, vol. abs/1505.04597, 2015.

[Russakovsky2015] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. *ImageNet Large Scale Visual Recognition Challenge*. arXiv:1409.0575 [cs], January 2015. arXiv: 1409.0575.

[Sabbagh2019] Majid Sabbagh, Cheng Gongye, Yunsi Fei and Yanzhi Wang. *Evaluating Fault Resiliency of Compressed Deep Neural Networks*. In 2019 IEEE International Conference on Embedded Software and Systems (ICESS), 2019.

[Salami2018] Behzad Salami, Osman S Unsal and Adrian Cristal Kestelman. *On the resilience of rtl nn accelerators: Fault characterization and mitigation*. In 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pages 322–329. IEEE, 2018.

[Samuel1959] Arthur L. Samuel. *Some Studies in Machine Learning Using the Game of Checkers*. IBM Journal of research and development, vol. 3, 1959.

[Schorn2018] Christoph Schorn, Andre Guntoro and Gerd Ascheid. *Efficient on-line error detection and mitigation for deep neural network accelerators*. In International Conference on Computer Safety, Reliability, and Security. Springer, 2018.

[Schorn2019] Christoph Schorn, Andre Guntoro and Gerd Ascheid. *An Efficient Bit-Flip Resilience Optimization Method for Deep Neural Networks*. pages 1507–1512. IEEE, March 2019.

[Shitao2017] Peng Shitao. *Fault injection time mark unification method, control device and fault injection system*. In Patent CN108023659A, 2017.

[Simonyan2014] Karen Simonyan and Andrew Zisserman. *Very deep convolutional networks for large-scale image recognition*. arXiv preprint arXiv:1409.1556, 2014.

[Solovyev2019] R. A. Solovyev, A. L. Stempkovsky and D. V. Telpukhov. *Study of Fault Tolerance Methods for Hardware Implementations of Convolutional Neural Networks*. Optical Memory and Neural Networks, vol. 28, no. 2, pages 82–88, April 2019.

[Souvatzoglou2021] Ioanna Souvatzoglou, Athanasios Papadimitriou, Aitzan Sari, Vasileios Vlagkoulis and Mihalis Psarakis. *Analyzing the Single Event Upset Vulnerability of Binarized Neural Networks on SRAM FPGAs*. In 2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2021.

[Srivastava2014] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov. *Dropout: a simple way to prevent neural networks from overfitting*. The journal of machine learning research, vol. 15, no. 1, 2014.

[Strigl2010] Daniel Strigl, Klaus Kofler and Stefan Podlipnig. *Performance and Scalability of GPU-Based Convolutional Neural Networks*. In 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010.

[Stutz2021] David Stutz, Nandhini Chandramoorthy, Matthias Hein and Bernt Schiele. *Bit error robustness for energy-efficient dnn accelerators*. Proceedings of Machine Learning and Systems, vol. 3, 2021.

[Syed2021] R. T. Syed, M. Ulbricht, K. Piotrowski and M. Krstic. *Fault Resilience Analysis of Quantized Deep Neural Networks*. In 2021 IEEE 32nd International Conference on Microelectronics (MIEL), 2021.

[Sze2017] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang and Joel S. Emer. *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*. Proceedings of the IEEE, vol. 105, no. 12, pages 2295–2329, December 2017.

[Sze2020] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang and Joel S. Emer. *Efficient Processing of Deep Neural Networks*. vol. 15, no. 2, pages 1–341, 2020.

[Talpes2020] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee, Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti and Gagandeep S. Sachdev. *Compute Solution for Tesla's Full Self-Driving Computer*. IEEE Micro, vol. 40, no. 2, 2020.

[Tax2021] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. Standard J3016_202104, Society of Automotive Engineers, 2021.

[Tiwari2022] Ankita Tiwari, Saras Mani Mishra, Prithwijit Guha, Pidanic Jan, Zdenek Nemec et Gaurav Trivedi. *Design of a Low Power and Area Efficient Bfloat16 based Generalized Systolic Array for DNN Applications*. In 2022 32nd International Conference Radioelektronika (RADIOELEKTRONIKA). IEEE, 2022.

[Torres-Huitzil2017] Cesar Torres-Huitzil and Bernard Girau. *Fault and Error Tolerance in Neural Networks: A Review*. IEEE Access, vol. 5, pages 17322–17341, 2017.

[Tsai2021] Timothy Tsai, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa et Stephen W. Keckler. *NVBitFI: Dynamic Fault Injection for GPUs*. In 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2021.

[Umuroglu2017] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre and Kees Vissers. *Finn: A framework for fast, scalable binarized neural network inference*. In Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays, 2017.

[Wan2021] Zishen Wan, Aqeel Anwar, Yu-Shun Hsiao, Tianyu Jia, Vijay Janapa Reddi et Arijit Raychowdhury. *Analyzing and Improving Fault Tolerance of Learning-Based Navigation Systems*. In 2021 58th ACM/IEEE Design Automation Conference (DAC). IEEE, 2021.

[Weaver2004] C. Weaver, J. Emer, S.S. Mukherjee and S.K. Reinhardt. *Techniques to reduce the soft error rate of a high-performance microprocessor*. In Proceedings. 31st Annual International Symposium on Computer Architecture, 2004., 2004.

[Webb2018] Stefan Webb, Tom Rainforth, Yee Whye Teh and M Pawan Kumar. *A statistical approach to assessing neural network robustness*. arXiv preprint arXiv:1811.07209, 2018.

[Wilkerson2008] Chris Wilkerson, Hongliang Gao, Alaa R Alameldeen, Zeshan Chishti, Muhammad Khellah and Shih-Lien Lu. *Trading off cache capacity for reliability to enable low voltage operation*. ACM SIGARCH computer architecture news, vol. 36, no. 3, 2008.

[Wirthlin2015] Michael Wirthlin. *High-reliability FPGA-based systems: space, high-energy physics, and beyond*. Proceedings of the IEEE, vol. 103, no. 3, 2015.

[Wu2021a] Jun-Shen Wu, Chi-En Wang and Ren-Shuo Liu. *Value-Aware Error Detection and Correction for SRAM Buffers in Low-Bitwidth, Floating-Point CNN Accelerators*. In 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC). IEEE, 2021.

[Wu2021b] Tony F Wu, Doyun Kim, Daniel H Morris and Edith Beigne. *Evaluation of Low-Voltage SRAM for Error-Resilient Augmented Reality Applications*. In 2021 IEEE Workshop on Signal Processing Systems (SiPS). IEEE, 2021.

[Xiang2019] Lin Xiang, Xiaoqin Zeng, Yuhu Niu and Yanjun Liu. *Study of Sensitivity to Weight Perturbation for Convolution Neural Network*. IEEE Access, vol. 7, pages 93898–93908, 2019.

[Xu2019] Dawen Xu, Kouzi Xing, Cheng Liu, Ying Wang, Yulin Dai, Long Cheng, Huawei Li et Lei Zhang. *Resilient neural network training for accelerators with computing errors*. In 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), volume 2160, pages 99–102. IEEE, 2019.

[Xu2021] Dawen Xu, Ziyang Zhu, Cheng Liu, Ying Wang, Shuang Zhao, Lei Zhang, Huaguo Liang, Huawei Li and Kwang-Ting Cheng. *Reliability evaluation and analysis of fpga-based neural network acceleration system*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 29, no. 3, 2021.

[Yang2017] Lita Yang and Boris Murmann. *SRAM voltage scaling for energy-efficient convolutional neural networks*. In 2017 18th International Symposium on Quality Electronic Design (ISQED), 2017.

[Yang2019] Jiwei Yang, Xu Shen, Jun Xing, Xinmei Tian, Houqiang Li, Bing Deng, Jianqiang Huang and Xian-sheng Hua. *Quantization networks*. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019.

[Yogamani2019] Senthil Yogamani, Ciaran Hughes, Jonathan Horgan, Ganesh Sistu, Padraig Varley, Derek O'Dea, Michal Uricar, Stefan Milz, Martin Simon, Karl Amende, Christian Witt, Hazem Rashed, Sumanth Chennupati, Sanjaya Nayak, Saquib Mansoor, Xavier Perrotton and Patrick Perez. *WoodScape: A Multi-Task, Multi-Camera Fisheye Dataset for Autonomous Driving*. In Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), October 2019.

[Yu2019] Fuxun Yu, Zhuwei Qin, Chenchen Liu, Liang Zhao, Yanzhi Wang and Xiang Chen. *Interpreting and evaluating neural network robustness*. arXiv preprint arXiv:1905.04270, 2019.

[Yvinec2021] Edouard Yvinec, Arnaud Dapogny, Matthieu Cord and Kevin Bailly. *RED: Looking for Redundancies for Data-FreeStructured Compression of Deep Neural Networks*. Advances in Neural Information Processing Systems, vol. 34, 2021.

[Zhang2018a] Jeff Zhang, Kartheek Rangineni, Zahra Ghodsi and Siddharth Garg. *Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators*. In Proceedings of the 55th Annual Design Automation Conference, 2018.

[Zhang2018b] Jeff Jun Zhang, Tianyu Gu, Kanad Basu and Siddharth Garg. *Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator*. In 2018 IEEE 36th VLSI Test Symposium (VTS), pages 1–6. IEEE, 2018.

[Zhang2019] Jeff Jun Zhang, Kanad Basu and Siddharth Garg. *Fault-Tolerant Systolic Array Based Accelerators for Deep Neural Network Execution*. IEEE Design & Test, vol. 36, no. 5, pages 44–53, October 2019.

A

Appendix : Fault Calendar

The design of safety critical systems must include a fault tolerance analysis. The analysis of the impact of faults occurring in a parallel computer often requires Register-Transfer Level (RTL) simulations, in order to model the propagation of data between compute units. Such RTL simulations are compute intensive and in this Appendix we present a technique, called the Fault Calendar, to analyze the impact of faults in a parallel system, using a high-level architectural model that still takes into account the data-flow.

A.1 Introduction

Hardware accelerators for Deep Neural Network (DNN)s often employ highly parallel architectures, where data flows between compute units. When a fault occurs in such a system, either it may propagate directly, or an erroneous result may propagate. For this reason, the impact of faults can not simply be modeled by injecting faults into an abstract model executing the calculations sequentially. In this appendix, we present a technique called the fault calendar, which allows the effects of faults in a parallel accelerator to be accurately modeled in a sequential implementation. In this way, the impact of faults can be modeled using the simpler, abstract model.

A.1.1 Scientific Background

Many authors have studied the fault tolerance of parallel and systolic compute systems. These works can be divided into three categories:

- **1.** Those who have studied the impact of faults on an abstract representation of an algorithm. In the context of neural networks, these authors view the neural network as a sequence of mathematical operations, and they inject faults when performing these calculations. The problem with this approach, is that these abstract faults do not correspond to the actual faults that would occur in the accelerator. These authors include [Arechiga2018, Reagen2018]. The limit of theses studies it that they do not reflect the impact of faults in a real accelerator.
- **2.** Those who have chosen a specific hardware accelerator. These authors evaluate the impact of faults using digital simulation on their accelerator. [Zhang2019] This approach is accurate, but the simulations are slow.
- **3.** Those who have taken a working system and evaluated the impact of faults by injecting faults in the actual system, typically using radiation. [dos Santos2018, dos Santos2019]

Our idea makes it possible to obtain the speed of evaluation of the works in group (1) with the accuracy of the works in group (2).

A.2 Fault Calendar

The main idea of the Fault Calendar is to enumerate every operation performed during the execution of the algorithm, and to associate a unique number to each operation. The numbers are assigned so that if operation OP_i occurs before OP_j then i<j. Without loss of generality, consider a parallel processing system with 4 parallel processors (P0..P3) which execute over 3 time-steps (TS0..TS2), with the operations numbered OP0 through OP11, as shown in Fig.A.1.

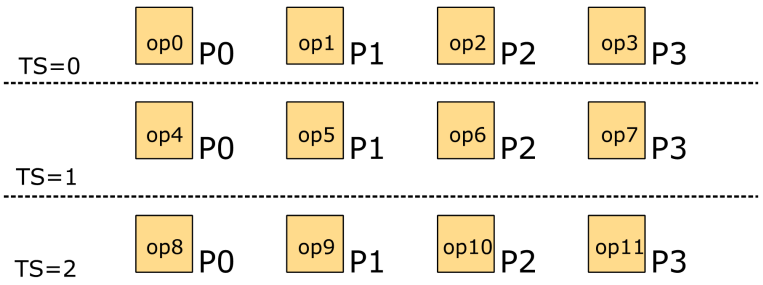


Figure A.1: Example of Parallel System

The same computation could be performed sequentially on a single threaded CPU, as shown in Fig.A.2.

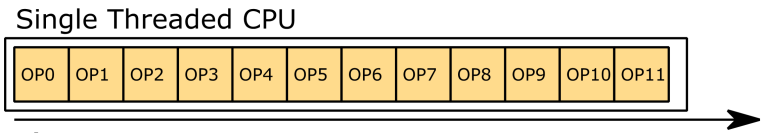


Figure A.2: Example of Sequential Execution

Of course, in real hardware, the parallel execution will be faster. However, during the design phase, evaluating the parallel implementation requires a simulation tool which can model a parallel system and such tools are typically slow. The sequential execution can be evaluated on a traditional CPU, and can be executed faster.

Let us suppose there is a fault in processor P1 in the parallel implementation, as shown in Fig.A.3.

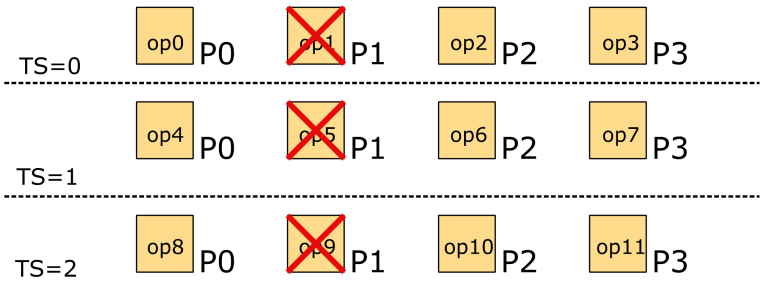


Figure A.3: Hardware Fault in Parallel System

As we can see, a fault in P1 affects all the operations mapped to this processor, specifically, OP1, OP5 and OP9 in this example. Assuming that the mapping of operations to processors is static, then we can emulate a fault in P1 in the sequential execution, as shown in Fig.A.4.

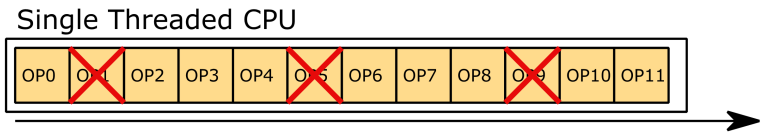


Figure A.4: Projecting Faults onto Sequential Execution

The fault calendar makes it possible to run the sequential execution and know when faults need to be injected, with a minimal run time overhead. Specifically, it consists of a table with one entry per actual fault, plus a single counter that must be maintained by the CPU. For, the above, example, the fault calendar would be initialized as shown in Tab.A.1.

Fault Number	F1	F2	F3
Counter	1	3	3

Table A.1: Initial Fault Calendar for Example

When the fault-injection is executed on a single-threaded CPU, only a single counter is required to track when faults need to be injected. This is shown in Fig.A.5. The counter is loaded with the initial value for F1, from the fault calendar (1 in this example). After each operation, the CPU decrements the counter. When the counter reaches zero, it is time to inject a fault. After a

fault is injected, the counter is reloaded with the next value from the fault calendar, and the process repeats. Faults are injected each time the counter reaches zero.

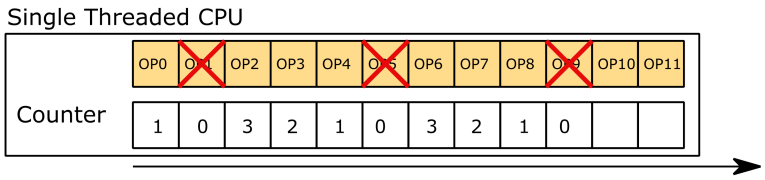


Figure A.5: Counter Combined with Fault Calendar for Injecting Faults

The counter is loaded with the count of the number of operations before the next fault injection. After each operation, the CPU decrements the counter and checks if it is zero. When it reaches zero, then the simulation injects a fault into the computation, and reload the counter with the next value (the new number of operation before the next fault injection). By using a single counter, the number of simulated fault is scalable (even for an important number of simulated faults, the extra cost of this method is the simple use of a counter). The algorithm is illustrated in Fig.A.6.

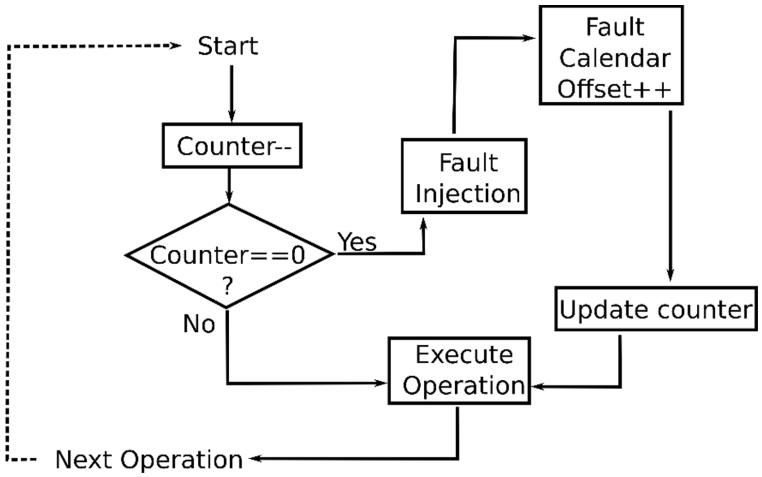


Figure A.6: Algorithm for Decrementing Counter and Injecting Fault

To summarize, in the parallel architecture, shown in Fig.A.3, a single fault can affect multiple operations. To emulate the same effect on a CPU (architectural level simulator), a static fault calendar can be built, as shown in Fig.A.1. Using this calendar, the algorithm can be executed on a single-threaded CPU and faults can be injected on the same operations as those that were affected by the fault in the parallel architecture.

As presented up to this point, we have dealt with a single, generic type of fault. In fact, there may be different types of faults, or the faults may have attributes such as which bit position is affected. The fault calendar can, of course, store this information in each entry. When the counter reaches zero, the CPU can consult the fault calendar, and inject the appropriate type of fault.

A.3 Patenting of the Fault Calendar

The following patents proposes idea that are similar to the Fault Calendar.

- **CN109581207A Low-cost circuit state control method for fault injection attack hardware simulation [Qiang2019]** The focus of this patent is to reduce the energy consumption of scan chains used for fault injection. Both this patent and our idea rely on the careful study of time to deduce at which moment to perform fault injection. But this method is for fault injection in real hardware using a scan chain in a FPGA.
- **CN109947609A A software and hardware collaborative acceleration method and system for fault injection [Gu2019]** This patent describes a technique for injecting faults into micro-processors. It is focused on a FPGA platform for fault injection, and unlike this proposal, does not address parallel architectures.
- **CN108023659A Fault injection time mark unification method, control device and fault injection system [Shitao2017]** This patent focuses on a methodology for synchronizing fault injections on several components. The context is testing real-world applications in post-production. A common point with our proposal is the handling of the time of the fault injection. However, we differ in the level of application of the fault injection since we focus on architectural simulation whereas this paper deals with actual hardware.
- **US2020301798A1 Fault Injection System and Method of Fault Injection (Huawei) [Cardoso2017]** This patent deals with the fault injection of large-scale distributed systems (e.g. clouds). A component called the Proxy Manager serves as a proxy between several targets. The Proxy Manager handles the communication between targets and can send fault injection orders to synchronize simultaneous fault injection on different targets.

We distinguish ourselves from these patents in two ways.

- **1.** Many of the existing patents propose techniques to inject faults in real hardware. In some cases (CN109581207A) this injection occurs at the logic level in an integrated circuit. In other cases (US2020301798A), it occurs in a system-level environment. In our proposal, we address how to inject faults in a high-level model of a data-path accelerator, running on a CPU.
- **2.** Our proposal is focused on parallel hardware implementations, such as systolic data-paths

The key characteristics of our invention are the following :

- Ability to model faults occurring in a parallel, possibly systolic, architecture by running a high-level, fast model on a single-threaded CPU. This provides high speed execution.
- Use of a data-structure (fault-calendar and down-counter) which requires minimal run-time overhead to determine the time when to inject the specific faults. This ensures the fidelity of the fault-injection.

Consequently, the idea behind the Fault Calendar was patented [Burel2022a].

Architectures pour la sureté de fonctionnement des systèmes à base de réseaux de neurones

Architectures to ensure the functional safety of neural network based systems

Résumé

Les réseaux de neurones sont utilisés dans des systèmes informatiques critiques tels que ceux utilisés pour la conduite autonome. Ces systèmes doivent respecter les normes de sûreté de fonctionnement ; il est donc essentiel d'assurer leur bon fonctionnement même en présence de fautes matérielles. Le même réseau formel peut être réalisé sur différentes plateformes matérielles (CPU, FPGA, etc), selon les besoins en performance. La fiabilité des systèmes numériques classiques (micro-contrôleurs, RAM, etc) est déjà bien étudiée mais les approches de ce domaine ne sont pas toujours adaptées aux réseaux de neurones. L'objectif de cette thèse est de trouver des nouvelles approches, à faible coût, pour améliorer la tolérance aux fautes des réseaux de neurones. Dans un premier temps, un résumé des travaux existants dans le domaine est présenté. Ensuite, la première contribution scientifique se concentre sur une étude de la robustesse des architectures systoliques existantes afin de proposer un nouvel accélérateur tolérant aux fautes grâce à des tests en ligne. Ensuite deux autres techniques de détection de fautes sont présentées : une qui se concentre sur les mémoires des réseaux de neurones et une autre qui détecte des anomalies statistiques induites par les fautes. Ces méthodes de détection de fautes sont combinées avec un système de masquage de fautes préexistant. Ces approches sont étudiées sur plusieurs cas d'étude, et prises dans leur globalité, elles ouvrent la voie vers la réalisation d'un accélérateur matériel pour les réseaux de neurones, tolérant aux fautes avec un surcout minimal.

Mots-clés : CNN, Intelligence artificielle, Testabilité, DNN

Abstract

Neural networks are increasingly used in mission critical systems such as those used in autonomous vehicles. These systems must comply with functional safety standards; therefore, it is essential to ensure they operate correctly in the presence of hardware faults. The same formal neural network can be implemented on different hardware platforms (CPUs, FPGAs, etc.), depending on the required level of performance. The reliability of classical digital circuits (micro-controllers, RAMs, etc.) has been well studied, but the techniques used in this domain are not always well adapted for neural networks. The objective in this thesis is to find new, low-cost, techniques to improve the reliability of neural networks. The thesis starts with an overview of the existing works in this field. The first scientific contribution is an analysis of the robustness of systolic accelerators leading to a proposal for a new architecture that achieves fault tolerance using on-line test. Next, two fault detection techniques are presented: one that focuses on the memory and the other that detects statistical anomalies caused by faults. These approaches are studied using several case studies, and taken together, they pave the way for the development of a fault tolerant neural network accelerator with minimal hardware overhead.

Keywords : CNN, Testability, Artificial intelligence, DNN