POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A novel approach for error modeling in a Cross-Layer Reliability Analysis of Convolutional Neural Networks

Tesi di Laurea Magistrale in
Computer Science and Engineering

Author: **Dario Passarello**

Student ID: 963220
Advisor: Prof. Antonio Miele
Co-advisors: Luca Cassano, Alessandro Nazzari
Academic Year: 2022-2023

Abstract

In the future, more and more systems will adopt AI-based computation in safety-critical applications. Convolutional Neural Networks (CNNs) are one of the pillars of this AI revolution since they can perform tasks on images that traditional computer vision algorithms are not able to perform, such as image classification, object detection and segmentation. These tasks have a notorious application in Automatic Driving Systems (ADS), which employ, for example, real-time object detection networks for recognizing street signs and obstacles from a video stream. The criticality of those systems prompted the scientific community to study in depth their reliability. Cosmic rays entering the atmosphere are an insidious physical phenomenon that can thwart the robustness of these systems by silently causing temporary bit-flips in the memory elements in the hardware, altering the course of the computation. Graphic Processing Units (GPUs), the preferred hardware platforms for running CNNs, are very vulnerable to cosmic radiation due to their high density of transistors and lack of protection mechanisms in their chip area. This makes it necessary to perform a reliability analysis at the early stage of the design of this class of systems. In the literature, various frameworks perform reliability analysis of CNNs; one of these is CLASSES, a cross-layer methodological framework that combines platform-level fault injection with an application-level error simulator. The interface between the two abstraction levels in CLASSES is constituted by the error models extracted from the results of the injections and used for generating errors in the simulation. Error models are a crucial part of the CLASSES methodology and need to represent with an acceptable fidelity the errors coming from fault injection. In the first version of CLASSES, the structure of the error models is quite rudimentary and can limit the fidelity of the generated error, especially when dealing with error patterns not present in the original work. Additionally, a big part of the error modeling process is manual, making the definition of those models a long task. The goal of this work is to improve the CLASSES framework by removing the weaknesses discussed above.

The first contribution of this work is the definition of a new structure of the error models, introducing a refined representation of the spatial and domain distributions of the corrupted values emerging from the outputs of the fault injection. These changes have the

goal of increasing the fidelity of the simulation while making the models easily reusable for multiple application analysis. Another contribution is a new systematic approach to the process of defining the error models. We rationalized the process of extraction of the operators from the CNN under test and the planning of the fault injection experiments. We designed and implemented a software tool that visualizes the error patterns emerging from the results of the fault injection. Starting from a first empirical analysis of visualized error patterns, the user defines a classification of the corrupted outputs in spatial classes that they define using code. The code definition of the classes is then used by the tool to generate the error models. While still being semi-automatic, the proposed workflow leaps towards the complete automation of the error model generation process. The methodological framework we propose as a whole still maintains the flexibility that is needed to make it operate with different CNNs and with various underlying hardware architectures.

Finally, to validate the proposed methodology and show the flexibility of the revised framework in producing various insights on the reliability of the network under test, we applied the proposed improvements by designing an injection campaign and a subsequent application analysis on YOLOv3, a state-of-the-art object detection network. In the analysis, we evaluate the vulnerability of the different operators and layers that constitute the network and we study how the different domain and spatial distribution of the errors in the feature maps affect the reliability of the network.

Keywords: Deep Learning, Convolutional Neural Networks, Reliability, Graphic Processing Units, Object Detection, Automatic Driving Systems

Abstract in lingua italiana

In futuro sempre più sistemi adotteranno la computazione basata sull'intelligenza artificiale (IA) in applicazioni critiche per la sicurezza. Le reti neurali convolutive (RNC) sono uno dei pilarstri di questa rivoluzione basata sull'IA. Infatti le RNC sono in grado di eseguire compiti, come la classificazione e il rilevamento di oggetti nelle immagini, che i tradizionali algoritmi di visione artificiale non riescono a svolgere. Queste attività hanno una ben nota applicazione nei Sistemi a Guida Autonoma (SGA) che impiegano, per esempio, reti neurali per rilevamento di oggetti in tempo reale, con lo scopo di individuare segnali e ostacoli dal flusso video proveniente da telecamere. La criticità di questi sistemi ha mosso la comunità scientifica nello studiare in profondità la loro affidabilità. Per esempio, i raggi cosmici che entrano nell'atmosfera sono un insidioso fenomeno fisico che può minacciare la robustezza di questi sistemi causando silenziosamente delle inversioni di bit negli elementi di memoria all'interno dell'hardware, alterando il corso della computazione. Le Unita di Elaborazione Grafica (UEG), sono l'hardware più adatto per eseguire le RNC, tuttavia sono molto vulnerabili alle radiazioni cosmiche a causa dell'alta densità di transistor al loro interno e della mancanza di meccanismi di protezione all'interno dell'area del chip. Questi problemi rendono necessario uno studio dell'affidabilità di questi sistemi già nelle prime fasi della progettazione. Nella letteratura sono presenti diversi framework il cui obbiettivo è eseguire una analisi dell'affidabilità anticipata delle RNC. Uno di questi framework è CLASSES, un framework metodologico cross-strato che combina l'iniezione di guasti a livello di piattaforma con una simulazione degli errori a livello dell'applicazione. L'interfaccia tra i due livelli di astrazione in CLASSES è costituita da i modelli di errore estratti dai risultati delle iniezioni e utilizzati per generare errori nella simulazione. I modelli di errore sono una parte cruciale della metodologia di CLASSES perciò necessitano di rappresentare con un buon livello di fedeltà gli errori provenienti dalle iniezioni di guasto. Nella prima versione di classes, la struttura dei modelli di errore è rudimentale e perciò limita la fedeltà degli errori generati specialmente nel momento in cui emergono nuovi pattern dalle iniezioni di guasto. Inoltre una buona parte del processo di modellazione degli errori è manuale, rendendo la definizione dei modelli di errore un compito lungo. L'obbiettivo di questo lavoro è quello di migliorare CLASSES risolvendo i problemi

appena elencati.

Il primo contributo di questo lavoro sta nella definizione di una nuova struttura dei modelli di errore, introducendo una rappresentazione rifinita delle distribuzioni spaziali e dei domini dei valori corrotti provenienti dalle iniezioni di guasto. Questi cambiamenti hanno l'obbiettivo di incrementare la fedeltà della simulazione rispetto ai risultati delle iniezioni e allo stesso tempo permettere il riutilizzo dei modelli in analisi di altre applicazioni. Un altro contributo di questo lavoro è un nuovo e sistematico approccio al processo di definizione dei modelli di errore. Abbiamo, infatti, razionalizzato il processo di estrazione degli operatori dalla RNC sotto analisi e abbiamo cambiato il modo in cui le campagne di iniezioni vengono pianificate. Abbiamo anche progettato e implementato uno strumento software che visualizza i pattern di errore che emergono dai risultati delle iniezioni. Iniziando quindi da una prima empirica analisi delle distribuzioni spaziali degli errori, l'utente definisce una classificazione dei vari output provenienti dalle iniezioni di guasto definendo le varie classi con del codice, che verrà utilizzato per generare i modelli di errore. Sebbene questo processo rimane non del tutto automatico, le procedure proposte fanno un passo avanti verso la completa automazione del processo di generazione dei modelli di errore. Il framework metodologico che proponiamo, nella sua interezza, mantiene un buon grado di flessibilità, necessaria per rendere il framework compatibile ad operare con diverse RNC e con diverse architetture hardware.

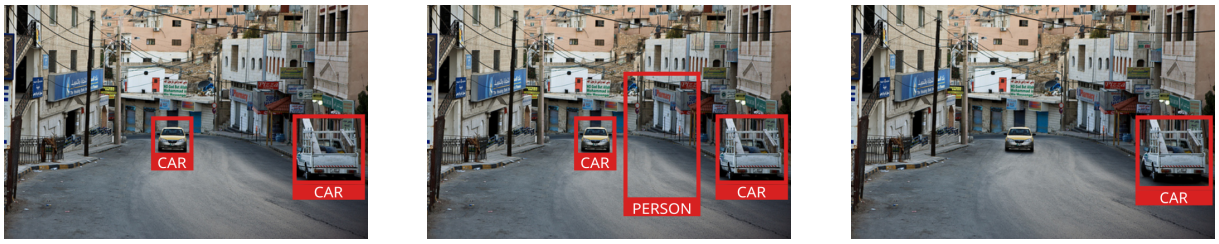
In conclusione, per validare la metodologia proposta e per mostrare il grado di flessibilità del framework nel produrre varie informazioni sull'affidabilità della RNC testata, abbiamo applicato i miglioramenti proposti nel design di una campagna di iniezione guasti e una successiva analisi della applicazione sulla rete YOLOv3, una rete di rilevamento oggetti al livello dello stato dell'arte. Nell'analisi abbiamo studiato la vulnerabilità dei diversi operatori ai guasti e abbiamo studiato come differenti distribuzioni spaziali e di dominio degli errori possono avere effetto sull'affidabilità della rete.

Parole chiave: Appredimento Profondo, Reti Neurali Convolutive, affidabilità, Unità di Elaborazione Grafica, Rilevamento di Oggetti, Sistemi a Guida Autonoma

Contents

1 | Introduction

Over the past decade, there has been a significant increase in the adoption of digital systems across various aspects of our daily lives. Depending on their specific roles and the operational context in which they are utilized, these products can play a crucial role in determining the success of the applications they are integrated into. An important category of digital systems is the one of *Safety-Critical Systems*, composed of all the digital systems that can cause harm to people and objects in case of failure. Examples of critical systems that are seeing their adoption increase are *Automatic Driver Systems* (ADS) in the automotive scenario. These systems are employed to partially, or in the future totally, replace the driver in their functions such as lane tracking, steering, road sign interpretation and obstacle identification. There are various degrees of automation, codified as a standard [40] proposed by the Society of Automotive Engineers (SAE) that classifies the various systems into six levels of automation, with level 0 being the lowest (no driving automation) and with level 5 being the highest (full driving automation, with the steering wheel becoming optional). Since ADS are safety-critical systems, they need to follow strict requirements about their reliability, as codified in the international standard ISO 26262 [26], which specifies the minimum reliability requirements for these systems. One type of event that can thwart the reliability of these systems is a physical fault occurring on the hardware used by the ADS. A fault may propagate from the hardware to the software application currently running on top of it causing a failure in the system, which may cause serious consequences to the safety of the system. These faults can originate from internal causes or environmental causes. Internal causes come from the normal wear and tear of the internal components caused by the day-by-day operation of the system and can be mitigated with self-diagnosis checks and timely replacement of the worn-out components. On the contrary, environmental causes come from punctual or continuative exposure to adverse external conditions such as high temperatures, humidity and radiation damage. Protecting the systems against environmental conditions instead requires designing protections from the various agents that affect the system, such as heat sinks and ingress protection mechanisms. The faults can be classified also based on their duration ranging between transient faults (also known as soft errors) having durations as



(a) Correct Detections (b) False Positive Detection (c) Missed Detection

Figure 1.1: Examples of Corrupted Outputs of an Object Detection Network

low as some nanoseconds and permanent faults that have an indefinite duration and even persist after a device reboot.

An emerging cause of transient hardware faults are the radiations carried by cosmic rays that in contact with the atmosphere generate showers of high-energy neutrons capable of altering the functioning of the integrated circuits that constitute most hardware of the digital systems. In [39] the authors estimated that the number of radiation-caused faults in a device has a frequency of two every thousand billion hours. The occurrence of a radiation fault may seem very rare but we need to factor that in 2021 400 million cars were travelling in Europe meaning that on any car a fault occurs every 3.5 hours. It should also be considered that thanks to the progress of the technology, the process size and the supply voltages of integrated circuits used in digital systems are constantly shrinking making them more vulnerable to faults caused by high-energy neutrons. Effects coming from cosmic rays have usually a transient nature and typically cause single bitflips in the flip-flops and other memory elements of logic circuits altering the course of the computation. This alteration may propagate in the computation affecting the output of the system.

Figure 1.1, shows some ways in which the output of an object detector, usually part of an ADS pipeline, can be corrupted by hardware faults. An example of output corruption is found in Figure 1.1b where a person is falsely detected, causing an inappropriate braking action. It could also happen that objects that in normal conditions would be detected are missed due to faults, like in Figure 1.1c, where a car coming in the opposite direction is not detected, potentially leading to delayed braking actions that can thwart the safety of the ADS. A way to evaluate how hardware faults impact the reliability of ADS systems is to perform error injection campaigns in the system using fault models that adhere as much as possible to the ones happening in the wild. To understand how to carry out these campaigns we need first to understand how the internal system works. Originally the tasks performed by an ADS were mainly implemented using traditional computer vision

algorithms and pipelines applied in serially to the input to obtain the desired output. With the advancement of the research in deep learning, trained Convolutional Neural Networks (CNNs) started to outperform the traditional methods, and nowadays are the preferred solution for most of the ADS tasks already listed before. CNN inferences have very high computational and memory requirements, and for a large part of their extent are embarrassingly parallel applications, making them a good fit to be executed in Graphic Processing Units (GPUs), that are well suited for this type of loads. A typical solution to solve the problem caused by soft errors is to duplicate the system and compare the two outputs, if they are different an SDC is detected. *Duplication with Comparison*, however, is too expensive and violates the strict constraints of power and cost pervasively present in the automotive domain. Other solutions harden only the most vulnerable parts of the network [7] reducing the resources required for the hardening without losing too much reliability. However, these solutions to be properly implemented require the results obtained from a reliability analysis. Normally, in the industry, the system-level reliability analysis is performed once the system is complete, however considering the complex nature of both the latest CNN models and the hardware architecture of GPUs employed, the corrections suggested by the hardening methods would require a lot of effort to be introduced once the various components of the system were already put together. To achieve a faster development cycle a common practice is to anticipate some of the reliability analyses performed on the single components of the system.

In the literature, there are multiple methodologies already in place to perform early reliability analysis of standalone CNNs running on GPUs. These methodologies inject errors at various levels of the software/hardware stack. In order of abstraction levels (high to low) we have software-level error simulators, platform-level fault injectors, hardware simulators and radiation tests. Some methodologies present in the literature are cross-layer, involving injections or simulations happening at multiple levels of abstraction storing the error models obtained in the lower level of abstraction and reusing them to reproduce the errors in the higher level. An example of these methodologies is CLASSES [8], a cross-layer methodological framework that combines platform-level fault injection with an application-level error simulator. The interface between the two abstraction levels in CLASSES is constituted by the error models. The error models are a crucial part of the methodology and so they need to contain accurate information about the spatial and value distributions of the corrupted values in the tensors (the main data structures used internally in the CNN). While studying and analyzing CLASSES we identified some weaknesses in this process that we are going to tackle as a part of the goal of this work.

1.1. Thesis Goal

Following the introduction of the scenario, described in the previous section, we now list the main goals of this work:

- **Goal 1:** *Propose improvements in the error modeling process of CLASSES.* The current error modeling methodology in CLASSES has some weaknesses. We tackle these flaws by proposing changes in the structure of the error models. Additionally, we propose a systematic approach to the planning of the platform-level fault injection campaigns. These improvements come with the goal of making the error models reflect better the spatial and domain distributions of the corrupted values in the tensors, increasing the fidelity of the simulation experiments and making the model easy to reuse for simulations in multiple networks.
- **Goal 2:** *Automate the error modeling process.* The CLASSES methodology requires that the user starting from a first empirical analysis of visualized error patterns, defines a classification of the corrupted outputs in spatial classes. In the current version of CLASSES, this process is mostly manual. We propose and implement a unified tool that allows the user to define the classes using code. The code definition of the classes is then used by the tool to automatically generate the error models.
- **Goal 3:** *Apply the proposed methodology for generating error models.* With the new proposed methodology, which includes the improvements described in the two previous goals, we define and subsequently execute a set of fault injection campaigns in the GPU. With the tool described in Goal 2, we define the spatial classes and we generate the error models of various commonly present in CNNs, including also some alternative implementations of the Convolution operator, the principal building block of CNNs.
- **Goal 4:** *Use the error models in a case study for analyzing the reliability of an object detection network.* We apply the error models generated to simulate errors at the application level in YOLOv3, a state-of-the-art object detection network [42]. From the results of the simulation experiments, we extrapolate multiple metrics about the reliability of the components of the CNN. In particular, we compare the reliability of the different operators and of the alternative convolution implementations.

1.2. Thesis Outcomes

The current outcomes of this thesis are the following:

- A paper entitled “Analyzing the Reliability of Alternative Convolution Implementations for Deep Learning Applications" co-autored by C. Bolchini, L. Cassano, A. Miele, A. Nazzari and D. Passarello has been accepted for presentation at the *36th IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems* (DFT 2023).
- a prototype of the tools automating the proposed methodology and all the obtained results are freely downloadable from `https://github.com/D4De/cnn-error-classifier`.

1.3. Thesis Structure

The organization of this work is the following:

- **Chapter 2** introduces the background knowledge needed for understanding this work. The chapter introduces CNNs by explaining their foundations, showing a couple of examples of real CNNs and describing the high-level frameworks for deep learning. The chapter also introduces one of the most used computation devices for CNNs, the Graphics Processing Unit (GPU). A section is dedicated to introducing the faults that can happen in the GPU during the execution of a CNN. Another section presents the state of the art in the field of error reliability frameworks and the last section presents CLASSES.
- **Chapter 3** proposed an innovative methodology that builds on top of CLASSES analyzing the weaknesses of the framework and from there we try to improve it by rationalizing the planning of the fault injection campaigns and improving the structure of the error models. We then propose a tool for automating part of the error modeling process.
- **Chapter 4** describes the results of the fault injections obtained by applying the presented methodology. The chapter starts by presenting the experimental environment and the choices made for planning the fault injection campaigns. This is followed by a qualitative and quantitative description of the error models obtained from the analysis of the fault injection.
- **Chapter 5** presents a variegated analysis of the results coming from application-level error simulations. Chapter 5 first describes the software-level experiment environment, lists all the different simulation campaigns and explains the metrics measured in the experiments. Then it explains the various analyses that were performed on the experiment results, including analysis of the various operators and layers of the network.

- **Chapter 6** contains the concluding remarks of this work and the possible future improvements and integrations of this work.

2 | Background and Related Works

In this chapter, we introduce the foundational concepts needed for understanding this work. Here we also perform an analysis of the state-of-the-art of already existing methodologies to perform a reliability analysis of Convolutional Neural Networks. This chapter starts with Section 2.1 where we define Convolutional Neural Networks and explain their basic building blocks. In Section 2.2 we introduce Tensorflow and PyTorch, two widely used high-level frameworks for implementing and running CNNs. In Section 2.3 we show two examples of CNNs, one performing image classification and another performing object detection. Section 2.4 explains the architecture of the Graphic Processing Unit (GPU) and why it is good for deep learning applications. Section 2.5 introduces the possible faults that can happen in the hardware during the execution of Neural Networks, and the consequences that they can have in the result. In Section 2.6 we perform an analysis of the existing frameworks for performing reliability analysis. Finally in Section 2.7 we describe CLASSES, a cross-layer reliability analysis framework that will be the object of improvements in this work.

2.1. Convolutional Neural Networks

In this chapter, we delve into the fundamental concepts and components of Convolutional Neural Networks. We begin by enunciating the fundamental definitions of deep learning and neural networks, highlighting the core principles behind these computational models. We will also investigate the key operators and layers that constitute CNNs, such as convolution and activation functions. Understanding these building blocks is essential for understanding the inner workings of CNNs and their applications.

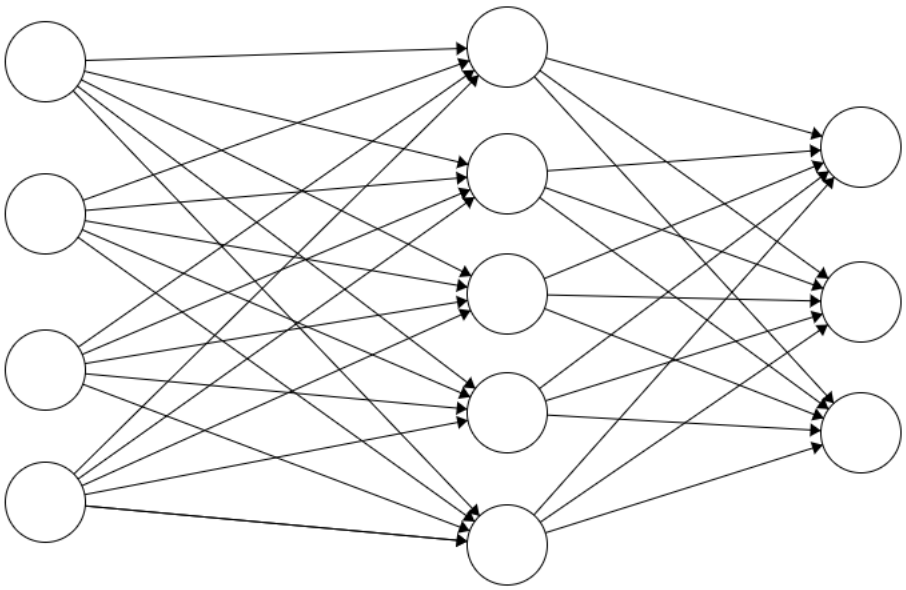


Figure 2.1: A Neural Network topology.

2.1.1. Definitions

Deep Learning (DL) is a field of artificial intelligence focused on creating and training *Neural Networks* (NN), which are computational models inspired by the structure and function of the human brain. These neural networks consist of interconnected nodes or "neurons" organized in layers, and they are capable of autonomously learning and recognizing intricate patterns within data.

In general, an NN takes a vector as its input. The input is processed in sequence by several hidden layers before yielding as its output another vector. Each hidden layer takes the intermediate output of the previous layer and outputs a new intermediate output. The intermediate outputs are called, in literature, *Activations* or *Feature Maps*, and are also vectors, with each activation of the vector called a *Neuron*. Figure 2.1 shows the topology of a simple NN, which in general is a *Directed Acyclic Graph* (DAG), namely a graph without any cyclic path. Each node of the graph is a *Neuron*, each one of the three columns of nodes is a *Feature Map*, and each one of the two sets of edges between two feature maps is a *Layer*.

Formally, the operation executed inside an NN layer is matrix multiplication, followed by the item-wise application of a non-linear function (called *Activation Function*) to the output vector of the matrix multiplication. More specifically, given the input feature map vector $u \in \mathbb{R}^{N\times 1}$, a weight matrix $W \in \mathbb{R}^{M\times N}$, a bias vector $b \in \mathbb{R}^{M\times 1}$ and a (non-linear)

activation function $\sigma \in \mathbb{R} \to \mathbb{R}$, the output feature map $v \in \mathbb{R}^{M \times 1}$ is calculated as:

$$v = \sigma(Wu + b) \tag{2.1}$$

The layer illustrated in Equation 2.1 is called *Fully Connected Layer*, because, for the properties of Matrix Multiplication, each element of v depends on all the elements of u. In Deep Learning, a *Tensor* is defined as a N-dimensional generalization of a matrix. The position of a single value inside a matrix can be identified by two integer coordinates (Row and Column). Similarly, the position of a single value in a N-dimensional tensor is identified by a tuple of N values. The *Shape* is an N-dimensional natural vector that specifies the number of elements contained in each dimension of the tensor. In some NN applications, such as image processing and sound processing, it is easier to see the input and the feature maps as tensors instead of as a one-dimensional vector.

Between the most used operators in DL, there is the *Convolution*, a binary linear operation between an input and a filter. The filter, usually smaller than the input, slides across the input, computing a dot product between the filter values and the corresponding values within an input neighborhood. The goal of the Convolution is to extract local features from the input. This operation will be better illustrated in Section 2.1.4. A *Convolutional Neural Network* (CNN) is a specialization of an NN in which *Convolutional Layers* replace fully connected ones. A *Convolutional Layer* has the same structure as the one specified in Equation 2.1, but the convolution replaces the matrix multiplication. Because of how the convolution works, each element of the output feature map will depend only on the neighboring corresponding input elements, making the layer partially connected. This change is desirable in applications of NNs where the features to be extracted exhibit patterns that are specific to certain regions or neighborhoods within the input.

2.1.2. Structure of CNNs

As explained before NNs and CNNs are structured in layers. The layers are connected together following a graph topology, more specifically they form a DAG since CNNs do not present cycles. Normally the layers are arranged in sequence with one layer following another, where the latter layer takes as its input, the output of the former. Figure 2.2 shows the structure of VGG-16 [49], a CNN able to perform various image recognition tasks. We can observe in the figure that the layers are all connected in a sequence. In more complex networks there are topologies different from the sequence, such as connections that branch, skip some layers and then merge back in the sequence, merging the feature maps with binary operators like the item-wise sum or the concatenation. These kind of

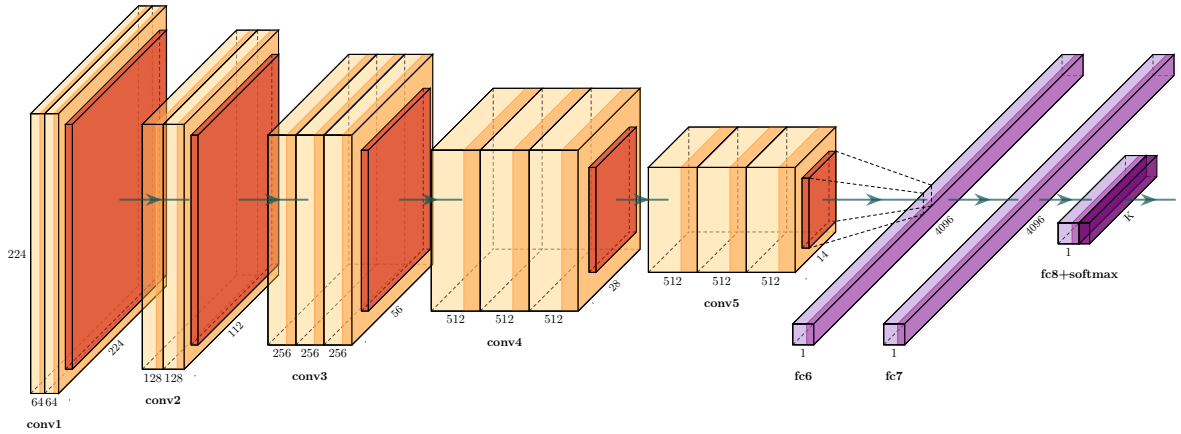


Figure 2.2: Structure of a VGG-16 convolutional network.

connections are called in literature *Shortcuts* or *Skip-Connections*.

All networks have an input layer that processes the input tensor whose shape depends on the purpose of the network. For example, if a CNN is designed to process images, it will usually take a 3-dimensional tensor as its input, with a shape of $C \times H \times W$, where H and W are respectively, the height and width of the image (in pixels) and C is the number of color channels. C is 1 if the image is monochromatic, 3 if the image uses Red, Green and, Blue channels, and 4 if the image also contains an Alpha transparency channel. There is also a single output layer whose output is the output of the entire network. Also in this case the shape of the output feature map depends on the task performed by the network. For example, an image classification network will associate an image to one of k possible classes. In this case, the output of the CNN will be a vector of k probabilities. Some applications of CNNs will be described in more detail in Section 2.3.

In general, the shapes of the feature maps vary through the layers of the network. In most CNNs, as the layer gets closer to the output, the height (H) and width (W) of the output feature maps' channels decrease while the number of channels increases. This phenomenon, called *Dimensionality Reduction*, is necessary to reduce the sensitivity of the model to small movements and minor differences in the input and can be observed also in 2.2.

Various types of hidden layers are used in CNNs besides the convolutional layers:

- Pooling
- Batch Normalization
- Fully Connected

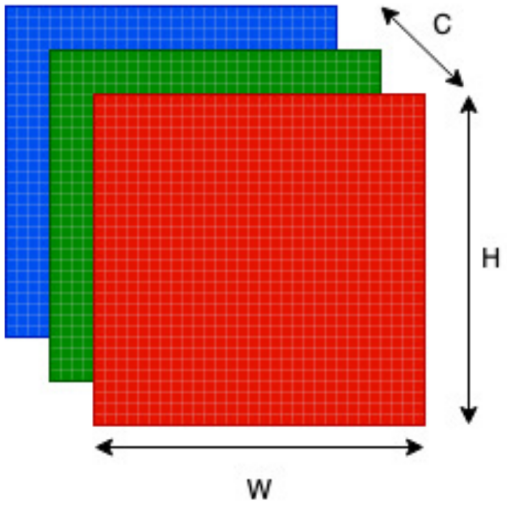


Figure 2.3: A Tensor of 3 channels, storing an RGB image.

- Softmax

Some of these layers can come with an activation function applied to their output. Sometimes, in the literature, the element-wise application of the activation function (named σ in Equation 2.1) is considered as a standalone layer called *Activation Layer*. Similarly, the sum of the output of the layer with the vector b can also be viewed as a standalone layer that is referred to in the literature as the *Bias Add* layer. We will describe part of these layers in the following sections.

2.1.3. Parameters, Training and Inference

In Section 2.1.1, we said that NN was capable of autonomous learning. The learning process, technically called *Training* is an automatic iterative process where the CNN refines its internal parameters referring to a dataset annotated by humans that contains the expected outputs of each item of the dataset.

CNN are highly configurable and have a lot of parameters, but not all can be trained. Based on this distinction, there are two kinds of parameters:

- *Trainable Parameters*, often called *Weights*, are parameters assigned and modified at each iteration of the training process. An example of trainable parameters are the values in the convolution filter or in the bias vectors.
- *Non-Trainable Parameters*, also called *Hyperparameters*, are parameters that are defined when the network is designed and cannot be changed during the training process. All the parameters that determine the topology of the network, such as the

shape of the filter of the convolution, are fixed at the design time of the network, hence non-trainable.

Let $P \in \mathbb{R}^p$ a vector containing all the p trainable parameters of the CNN; the training process tries to minimize a *Loss Function*, $\mathbf{L}(\theta) : \mathbb{R}^p \to \mathbb{R}$. The loss function at a given iteration represents a measure of the error between the actual outputs obtained by feeding the entire dataset to the network and the desired outputs contained in the annotations of the dataset. The minimization process is performed using the backpropagation method, following the graph of the network backward and layer by layer, adjusting the parameters in such a way that the value of the loss function decreases. This adjustment is typically carried out using an optimization algorithm, such as gradient descent. Training is a very computationally intensive process since it combines forward predictions with backpropagation, where a huge number of derivatives have to be calculated in order to update millions of weights. For instance, YOLOv3 [42], an object detection network of medium complexity, has around 65 million trainable parameters.

Once training is complete, the performance of the network is validated using an annotated dataset disjoint from the training dataset to see if the network can generalize what it learned to other inputs. The metrics for evaluating the performance vary based on the task performed by the CNN, for example in the case of image classification CNN, the percentage of images classified in the right class (also called *Accuracy*) is measured. Once the network reaches a satisfactory performance is ready to perform its task normally, by receiving in input any tensor respecting the required input shape and receiving the output; this process is also known as *Inference*.

2.1.4. Convolution

Convolution [21] is the fundamental operation at the base of CNNs. The operation is a readaptation of the discrete convolution operation widely employed in signal theory. Given two discrete sequences, the input $x : \mathbb{Z} \to \mathbb{R}$ and the filter (or kernel) $w : \mathbb{Z} \to \mathbb{R}$, the discrete convolution between these two sequences is calculated as:

$$y[n] = (x * w)[n] = \sum_{t=-\infty}^{\infty} x[n] \cdot w[n-t] \tag{2.2}$$

In real-world applications, the filter $w[k]$ can be assumed to be limited, or more formally, w is 0 when $|k| \leq \bar{k}$. The intuition behind the convolution is that the k-th element of the output ($y[k]$) is obtained from a linear combination of the corresponding element of the input $x[k]$ and its neighbors in the interval $[k - \bar{k}, k + \bar{k}]$. The coefficients of the linear

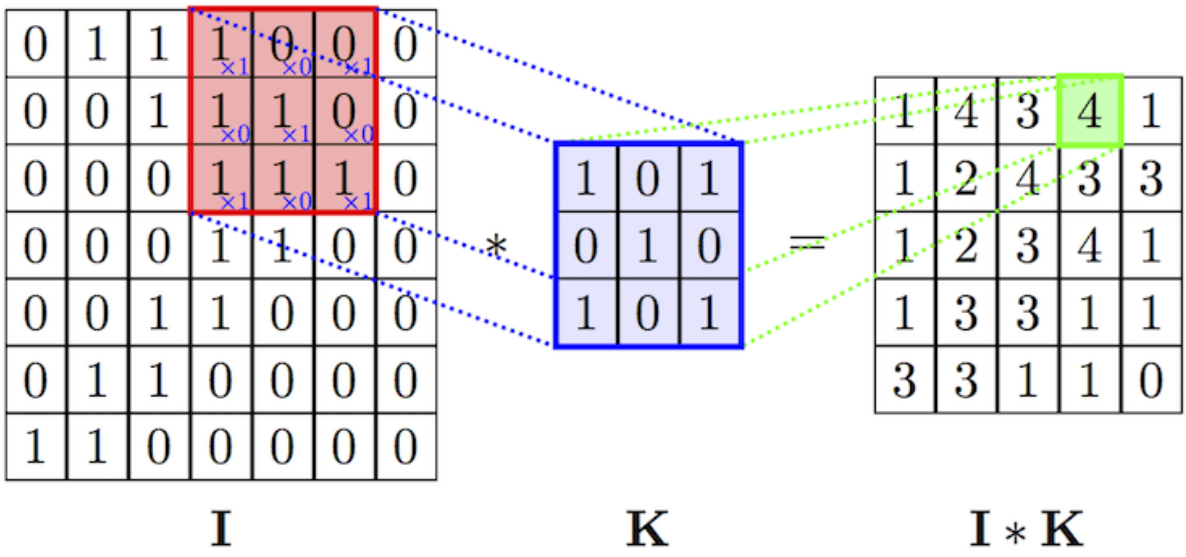


Figure 2.4: A Convolution visualized

combination are stored in the filter w, and they are reused for calculating each $y[k]$.

In CNNs applications, the two operands of a convolution are often two 4-dimensional tensors. The input tensor has shape (N, C, H, W) and represents a batch of N input feature maps, and the filter has shape (S, C, F, F) and contains multiple $F \times F$ filters to apply to the input images. The output tensor has a shape of $(N, S, \bar{H}, \bar{W})$.

For calculating each one of the S output channels, C $F \times F$ filters are applied simultaneously to each one of the C input channels. The application of the filter to a single channel is shown in Figure 2.4 which illustrates the convolution of a channel I with a filter K. After all the convolutions are executed all the resulting channels of size $(\bar{H}, \bar{W})$ are collapsed together with an element-wise sum. As shown in Figure 2.4, the size of the output channels is different from the one the input channels because the filters can slide only on all elements of the input channel in which all the $F \times F$ neighbors are defined, so they cannot be applied near the edges and the angles of the channels, unless some zero-padding is used.

The most used deep learning frameworks employ different strategies for performing the Convolution primitive, choosing dynamically the one that could perform better given the size of the input and the underlying hardware architecture. Here are some of the most known strategies of the convolution:

- The *Direct algorithm* comes directly from the definition of Convolution. It computes every element of the output matrix as the linear combination of the neighbors of the corresponding element of the input. The Direct approach is rarely used in deep

learning frameworks in favor of more complicated but faster algorithms.

- The *General Matrix Multiplication* (GEMM) Convolution algorithm, maps the convolution operation to a GEMM operation by building the *Doubly block circulant matrix* from the input and the filter. Basically, it creates a new matrix positioning the repeating the elements of the output channels in order to make the matrix product equivalent to the direct convolution. This is the most frequently used strategy since there are a lot of libraries for every hardware to perform matrix multiplication fast and efficiently.
- The *Fast Fourier Transform* (FFT) approach [38] exploits the convolution theorem, stating that the convolution operation in the frequency domain becomes a simple element-wise product. Using the FFT algorithm, the input and the filter are both transformed to the frequency domain. The two transformed operands are then multiplied together element-wise. The result is transformed back using the Inverse Fast Fourier Transform (IFFT) algorithm. The FFT approach is more efficient compared to classic convolution algorithms when the output and the size of the filters are big enough or when multiple inputs are batched together in order to minimize the number of transforms because the time to perform FFT and IFFT tends to dominate the time to perform element-wise multiplication.
- The *Winograd* [29] approach minimizes the number of floating point multiplications performed during a convolution by calculating and storing some intermediate values. The pre-computed values are summed together to obtain the correct output. This approach is proven to be faster when the sizes of the filters are small (such as 3x3 filters).

2.1.5. Activation

An *Activation* layer applies to all elements of a tensor a predefined non-linear function. The primary purpose of having an *Activation* layer is to ensure that the model during training is able to capture the highly non-linear nature of the problems solved by the CNN [37]. The layers mentioned so far can be expressed as linear functions, meaning that the output of all the layers can be written as a linear combination of its input. Convolution, for example, is a linear operator since each value of the output tensor is a linear combination of the neighbors' values of the corresponding input value. Without activation, a CNN will be just a very deep linear regression model with limited capacity for learning and performing the complex tasks that would normally be able to perform.

One of the most popular activation functions is the *Rectified Linear Unit* (ReLU) [54] and

its derivatives. The classic ReLU is a piecewise linear function shown in Figure 2.5 and with analytical form $ReLU(x) = max(x, 0)$.

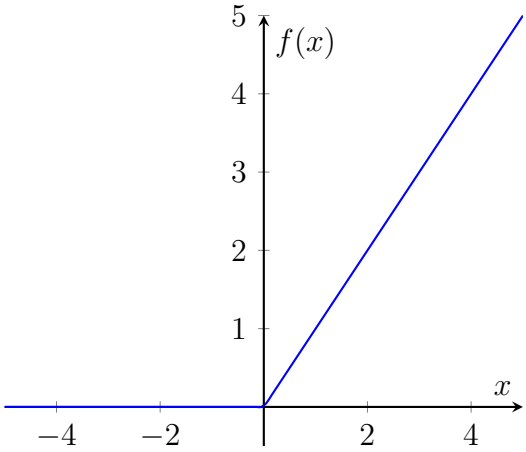


Figure 2.5: ReLU Activation Function

The leaky ReLU, in Figure 2.6, and with analytical form $LeakyReLU(x) = max(x, ax)$ where $a << 1$, crops negative values with a slow slope instead of completely cropping them to 0. Note that a is a non-trainable parameter. The main advantages of the family of ReLU functions are a low execution time, simplicity, and their ability to prevent the vanishing gradient problem.

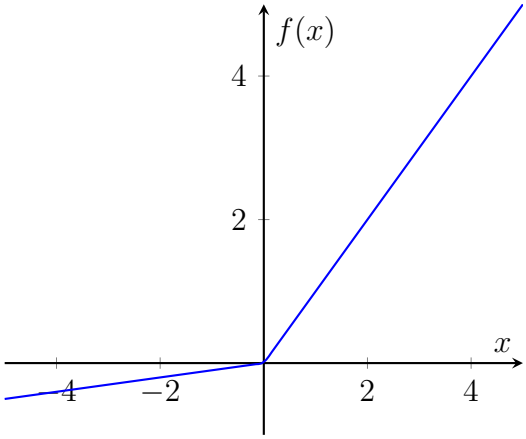


Figure 2.6: Leaky ReLU Activation Function

The ELU activation function, in Figure 2.7, maintains the linear part for positive inputs and has a saturating exponential part for negative inputs. It can be expressed as:

$$\text{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ a \cdot (e^x - 1) & \text{if } x < 0 \end{cases} \tag{2.3}$$

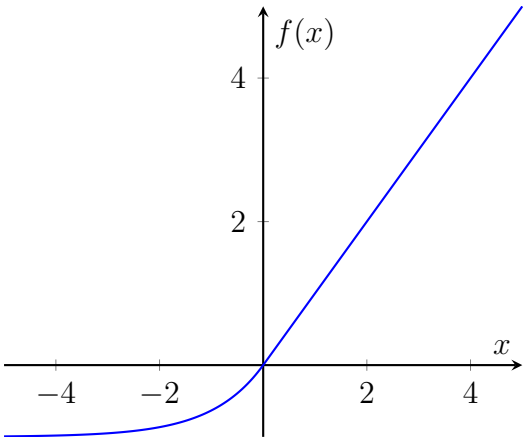


Figure 2.7: ELU Activation Function

The sigmoid function, in Figure 2.8, also known as the logistic function, has its image in the interval $(0, 1)$. For this reason, it is commonly used for representing probabilities. Its analytical form is expressed as:

$$\text{sigm}(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

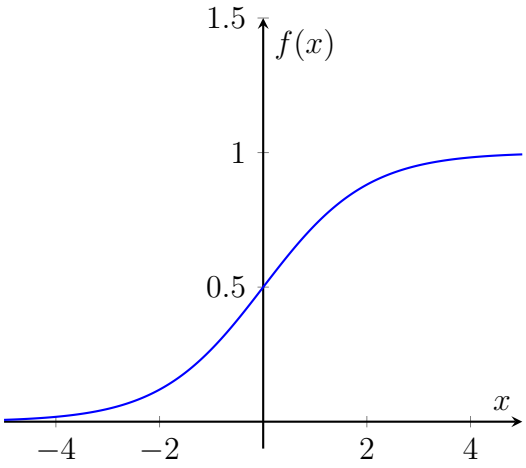


Figure 2.8: Sigmoid Activation Function

2.1.6. Other Operators

Batch Normalization (BN) is a layer used in Convolutional Neural Networks (CNNs) to improve training efficiency and generalization. During training, it normalizes the activations of each layer across a mini-batch, ensuring that they have zero mean and unit variance. This stabilizes and speeds up the convergence of training by reducing internal covariate shifts. Additionally, batch normalization introduces learnable scale and shift

parameters per feature map, allowing the model to adapt the normalization to the data. During inference, the normalization statistics (mean and variance) are fixed to the estimated population statistics from the training data. This ensures consistent behavior during inference, making the model less sensitive to small changes in input distribution. By maintaining the learned statistics, the performance of the model remains stable and reliable when making predictions on unseen data.

A *Pooling* layer has the scope to reduce the width and height of the feature maps in order to make the model more resilient to small movements and small differences in the input, as we explained in Section 2.1.2 when we mentioned the concept of Dimensionality Reduction. To downsample the image, a pooling layer divides an image into tiles of $t \times t$ pixels (usually $t = 3$), and it samples one pixel from each tile. The two most used strategies for sampling from a tile are *Average Pooling*, which takes the average of the tile, and *Max Pooling*, which selects the maximum from each tile.

2.2. High-Level Frameworks for CNNs

Implementing an NN from the ground up poses significant challenges due to the numerous operators to implement and their complexity. Optimizing the code for leveraging the potentiality of specialized hardware requires deep knowledge of the target architectures. To allow programmers to focus only on the high-level aspects of deep learning, over time, multiple libraries were developed with the scope of abstracting all the complex low-level implementation details of CNN. In this section, we will briefly present TensorFlow and PyTorch, two of the most used frameworks in the field of artificial neural networks.

2.2.1. Tensorflow

TensorFlow [3] is a Machine Learning framework developed by Google that can be used along with different programming languages, including Python, C++, and Javascript. Tensorflow programs can be executed on CPUs and GPUs. Tensorflow models are represented logically as static directed data-flow graphs. The nodes of the graph are operators, and each node can have inputs and outputs. Tensorflow offers a wide variety of built-in operation nodes:

- Element-wise mathematical operators
- Matrix operations
- Tensor operations

- NN building blocks (Convolutions, Activations, Pooling)
- Checkpointing nodes (for backing up from the RAM to disk the status of a computation, in case of a failure)
- Stateful nodes (variables and assignments)

Edges represent data flowing from one output of a node to the input of another node. The data type passing through edges is usually a tensor containing numeric data types. Tensorflow also provides a particular edge representing a control dependency that does not transfer data between nodes. A node that is the destination of control dependency edges will only execute after all the source nodes of those edges complete their execution. Tensorflow picks the node to execute from a queue. A node is inserted in the queue when all its data inputs are ready and all control dependency edges have been resolved. *Keras* [11] is a library that works on top of TensorFlow. Its main goal is to provide the programmer with an abstraction layer to simplify the code for complex CNN architectures and comes with various pre-trained and ready-to-use CNN models together with their datasets.

2.2.2. PyTorch

PyTorch [41] is a Deep Learning framework developed by Facebook's AI research group, implemented as a library for Python. PyTorch represents all layers of a neural network as objects that inherit from the `Module` base class, allowing the program to create custom layers by extending `Module` and implementing the `forward` method. At inference time, PyTorch will execute the `forward` method of the modules in their order, passing the input tensor as a function parameter. The value returned from the `forward` will be the output of the layer.

Unlike TensorFlow, the computation graphs in PyTorch are dynamic, allowing the programmer to define and edit graphs on the fly at runtime. Dynamic computation graphs provide more flexibility during development, also easing the debugging process. Their main drawback is the overhead incurred for optimizing the graph at runtime, but in most deep-learning applications, this overhead is negligible.

2.3. Applications of CNNs

There are many image-processing tasks that CNNs are able to perform, in this section we will discuss two image-processing applications for CNNs. For each of the two tasks, we will formally introduce the problem, then we will dissect an example from the literature

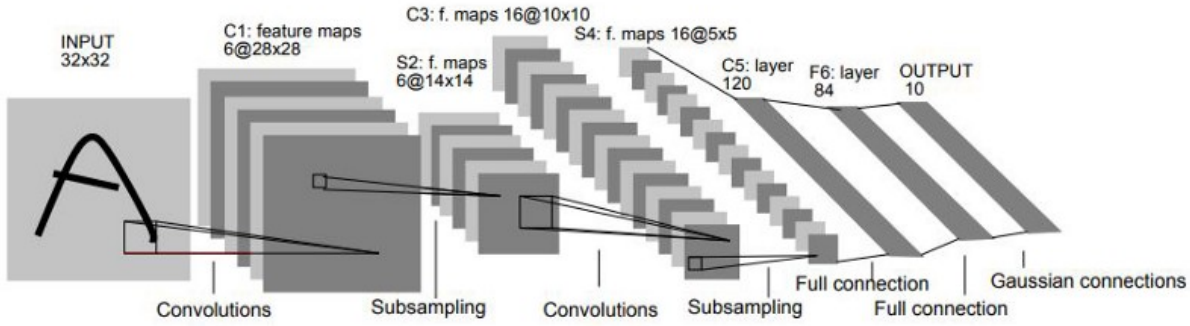


Figure 2.9: LeNet-5 Network Topology

of a CNN performing the given task and finally, we will discuss what metrics can be used to quantify how well the CNN can perform its assigned task.

2.3.1. Classification

A *Classification* network for images has the goal to assign an image the class to which it belongs choosing between k classes predetermined at training time. The output of a classification network is a vector $P \in [0,1]^k$ where $\sum_{i=0}^{k-1} P_i = 1$. The values of P represent the probabilities of the input image belonging to each class.

LeNet-5 [30] is one of the first and most known CNNs for carrying out classification tasks. Figure 2.9 illustrates the architecture of LeNet-5. The network contains, in order, a convolutional layer with a 5×5 filter, an average pooling layer with the scope of subsampling the image, and the application of the sigmoidal activation function. This structure of three layers is repeated two other times and is followed by a linearization and then two fully connected layers and a Radial Basis Function. The original task assigned to LeNet-5 was to classify handwritten digits from the MNIST database [31]. However, with time LeNet was proven to be very flexible, and it was employed for other tasks, like classifying images from CIFAR-10, a dataset with 32×32 images containing objects that belong to 10 classes of ordinary objects.

For evaluating the performance of classification networks like Lenet-5, we can run inferences on a validation dataset where each image is annotated with its actual class. From the results of the inference, we can calculate *Accuracy*, which indicates the percentage of correct predictions of the network under test.

2.3.2. Object Detection

An *Object Detection* network aims to detect, inside an image, the positions of objects belonging to one of the k predetermined classes. The output of an object detection network is a list of rectangular *Bounding Boxes* (BB) that enclose a single object belonging to the k object classes. Each BB comes with a confidence value, a real number in the interval $[0, 1]$ that represents the probability of the enclosed region of the image containing a single object belonging to the specified class. The bounding boxes are usually defined by 4 coordinates, either two opposing vertices of a rectangle or the top left point of the rectangle and its width and height.

Faster R-CNN [44] is an object detection network. It is composed of two different networks. The first is a Region Proposal CNN that scans the input image with the goal of extracting multiple candidate regions that may contain a single object belonging to one of the k classes. All the proposed regions are then passed to a classifier network that determines the class to which the selected region belongs, if any.

YOLOv3 [42] is a real-time capable object detection network. Differently from Faster R-CNN and other prior networks, YOLO applies a single CNN two the whole input image. From here comes the name "You Only Look Once" hidden in the acronym YOLO. The authors of YOLOv3 proved that the proposed approach is faster than the prior ones and allows real-time object detection, showing that in a high-end device, it is possible to perform more than 30 inferences per second, enabling the use of CNN for real-time object detection.

The architecture of YOLOv3 is shown in Figure 2.10. At his beginning, the image goes through a feature extractor, a re-adapted version of *Darknet-53* [42], composed of 53 blocks, each one containing in order a convolution, a batch normalization and a Leaky ReLU activation. The topology of Darknet-53 contains various skip connections and residual layers.

Detection Blocks follow the feature-extracting layers. The goal of a detection block is to detect the positions and classes of each possible BB. As shown in Figure 2.10, three independent detection blocks split from layers 69, 91 and 103, processing feature maps of different scales. This strategy is used to achieve accurate detections on objects that have different sizes inside the image. The outputs of all three detection blocks are combined, in a single vector of BBs, re-scaling the coordinates of each BB to the one relative to the original image.

The output of a detection block is a 3-dimensional tensor containing information about a

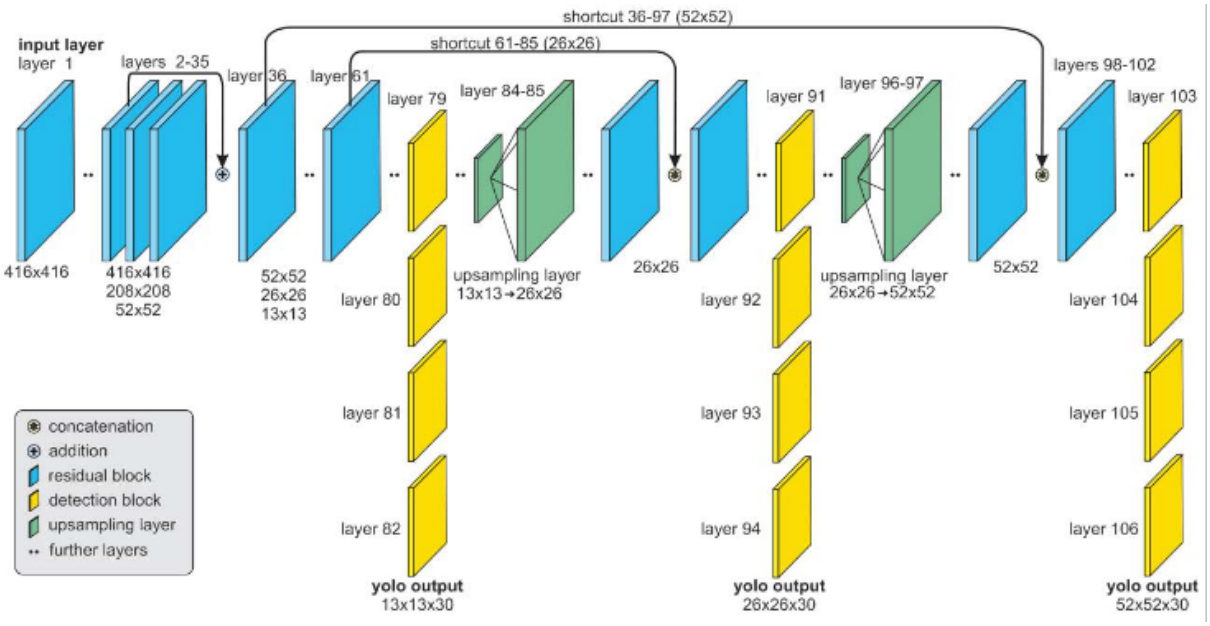


Figure 2.10: YOLOv3 network topology [55]

large number of bounding boxes. The size of each channel depends on the scale chosen. For each pixel of the feature map, YOLO tries to predict B different BBs. To store the information needed, the number of channels of the feature map is $(4+1+k) \times B$. In fact, for storing a single bounding box, we need 4 coordinates for the position of the BB, one additional value for the general objectness score (the probability of having a single object inside the BB), and k values, one for each class, for the values representing the confidence that the BB contains an object belonging to that class. Each class confidence value can be between 0 and 1, and it is calculated independently from the other classes.

After combining the output of the three detection blocks, there will be a lot of candidate BBs. A post-processing layer discards all the BBs with low objectness and/or confidence scores. For each object, there will still be multiple bounding boxes with high scores. To avoid multiple detections for a single object, the network will choose the BB with the best scores from the set of BBs that include the same instance of an object. This procedure is called *Non-Maximum Suppression* (NMS). After NMS, the remaining BBs will be associated with the class having the highest confidence score.

As expected the process of performance evaluation of an object detection task is more complicated than the one in classification networks, because the boundary between a correct and a wrong inference is more blurred. It is not required that the coordinates of the output BBs be exactly equal to the ones in the annotated dataset, and there is a certain tolerance before the detection is not safely usable by a digital system. In the rest

of the section, we will present some widely used metrics to quantify how well an object detection network like YOLOv3 performs.

Intesection Over Union (IoU) measures how spatially similar two bounding boxes are and is calculated as

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.5}$$

Where $|A \cap B|$ is the area of the intersection of the two BBs, and $|A \cup B|$ is the area of the union of the two BBs. The more two objects overlap proportionally to their area, the higher the IoU is. Two perfectly identical BBs will have IoU of 1, while two disjointed BBs will have an IoU equal to 0. We can use the IoU metric to determine how well the CNN detects objects. To do this, we should find a way to compare the annotated images in the dataset to the results of the network inference. In particular, for each BB annotated in the dataset, we need to find a match in the list of output BBs returned from the CNN.

There are two requirements for matching two BBs:

- The two BBs must be spatially similar enough, or more formally, they must have an IoU greater than a certain threshold.
- The two BBs must belong to the same class of objects.

Ideally, we should have each annotated BB paired up with an output BB, with the BBs sharing the same class and having an IoU bigger than a fixed IoU threshold (for example 0.5). Since trained CNNs usually do not have perfect accuracies, we will expect that some BB remains unmatched on both sides. The matched BBs are called *True Positives* because they correctly identify an object annotated in the dataset. The unmatched BBs in the annotation sets are the *False Negatives* since they represent an object that a human detected in the image but that was not detected from the CNN. Conversely, the unmatched BBs in the output set are the *False Positives* because they detect an object not seen and annotated by a human.

Algorithm 2.1 describes how to match the BBs in order to calculate the number of *True Positives*, *False Positives* and *False Negatives*. This algorithm must be launched once for every detection and every class supported by the model. The input lists *BBAnnotated* and *BBDetected* must contain only annotations and detections for the class currently analyzed. From the results of the algorithm, we can calculate two crucial performance metrics *Precision* and *Recall*. The two metrics can assume values between 0 and 1; the higher their value is, the better the network performs detections. Precision and Recall are calculated as:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.6}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.7}$$

Where TP is the number of True Positives, FP is the number of False Positives, and TN is the number of True Negatives. High precision indicates that when the CNN predicts a positive outcome, it is likely to be correct, while a High Recall means that the CNN is good at not missing positive instances. Precision and Recall can be either calculated separately for each class of objects or can be calculated for all the classes by summing the TP, FP and FN calculated for each class.

The *F1 Score* [51] is a metric that combines precision and recall by calculating their harmonic average:

$$\text{F1} = 2\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.8}$$

A widely used metric for benchmarking object recognition networks is the *Mean Average Precision* (mAP) [25], the mean of the *Average Precision* (AP) of all the classes. The AP of a class has values in $[0, 1]$ (the higher the better) and it is an approximation of the area under the precision-recall curve, calculated from the detections of objects of a given class sorted by their confidence value.

Another metric, close to the concept of "right or wrong" prediction, used in evaluating the performance of classification networks, is the *Whole Image Accuracy* (WIA), the percentage of images where all the annotated BB are detected by the network, without having false positives or negatives.

Algorithm 2.1 Bounding Boxes matching algorithm

Inputs
BBAnnotated: List of annotated BBs belonging to single class
BBDetected: List of output BBs belonging to single class
IoUThreshold: Minimum IoU for matching two boxes
Outputs
TPCount: Number of true Positives
FPCount: Number of false Positives
FNCount: Number of false Negatives

1: $TPCount \leftarrow 0$
2: Allocate a Matrix M of size $BBAnnotated$.size $\times$ $BBDetected$.size
3: **for** $i \leftarrow 0$ **to** $BBAnnotated$.size **do**
4: **for** $j \leftarrow 0$ **to** $BBDetected$.size **do**
5: $M_{i,j} \leftarrow IoU(BBAnnotated[i], BBDetected[j])$
6: **end for**
7: **end for**
8: *finished* $\leftarrow$ **false**
9: **while** *finished* is **false do**
10: $m \leftarrow max(M)$
11: **if** $m < IoUThreshold$ **then**
12: *finished* $\leftarrow$ **true**
13: **else**
14: $(i, j) \leftarrow argmax(M)$
15: $TPCount \leftarrow TPCount + 1$
16: Remove i-th element from $BBAnnotated$
17: Remove j-th element from $BBDetected$
18: Remove i-th row from M
19: Remove j-th column form M
20: **end if**
21: $FNCount \leftarrow BBAnnotated.size$
22: $FPCount \leftarrow BBDetected.size$
23: **end while**

2.4. GPU and Parallel Programming

In the world of modern computing, Graphics Processing Units (GPUs) have evolved from dedicated graphic accelerators into versatile general-purpose devices, suitable for running various highly parallelizable tasks. CNNs, introduced in the previous sections, are well-suited to be executed in GPUs. In this section, after a brief historical introduction, we will describe the architecture of a GPU, then we will proceed with explaining their programming model, introducing CUDA. We conclude the section by delving into a practical example of how matrix multiplication, a pervasively used operation in CNNs, can be accelerated in the GPU.

2.4.1. GPU Architecture

In 1999, NVIDIA released the GeForce 256 [36], naming that device, for the first time, a *Graphic Processing Unit* (GPU). The GeForce 256 had only the single purpose of accelerating the graphic pipeline. All the stages of the pipeline were implemented in hardware and had a fixed function that could not be modified by the programmer.

A decade of research and development transformed the GPU from being a single-purpose device to become a general-purpose device. In 2010, NVIDIA released the Fermi Architecture, defined as the "First Complete General Purpose GPU" [20]. The Fermi architecture featured an array of fully programmable unified shader processors with a unified memory architecture. In this new architecture, the graphic pipeline became just a simple software abstraction and no more the only purpose of the GPU. After Fermi other newer architectures were developed like Fermi (2010), Maxwell (2014), Volta (2017), Turing (2018), and Ampere (2020) [12]. While each architecture introduces new features and optimizations, the basic building blocks that constitute modern GPU architecture do not change too much.

A modern GPU contains multiple GPU processing clusters (GPC). Each GPC includes multiple *Streaming Multiprocessors* (SM), which are particular manycore processors. The SM can simultaneously execute a group of threads that share the same instruction stream. The group of threads is called *Warp* and in NVIDIA architectures, is composed of 32 threads. The threads of the warps are executed in lockstep, but they are still able to branch separately. This paradigm is called *Single Instruction Multiple Threads* (SIMT). A Streaming Multiprocessor, as shown in Figure 2.11 has the following units:

- A *Warp Scheduler*, deciding which warp will be executed at the next clock cycle. The scheduler will choose between all the warps that are not stalled due to memory

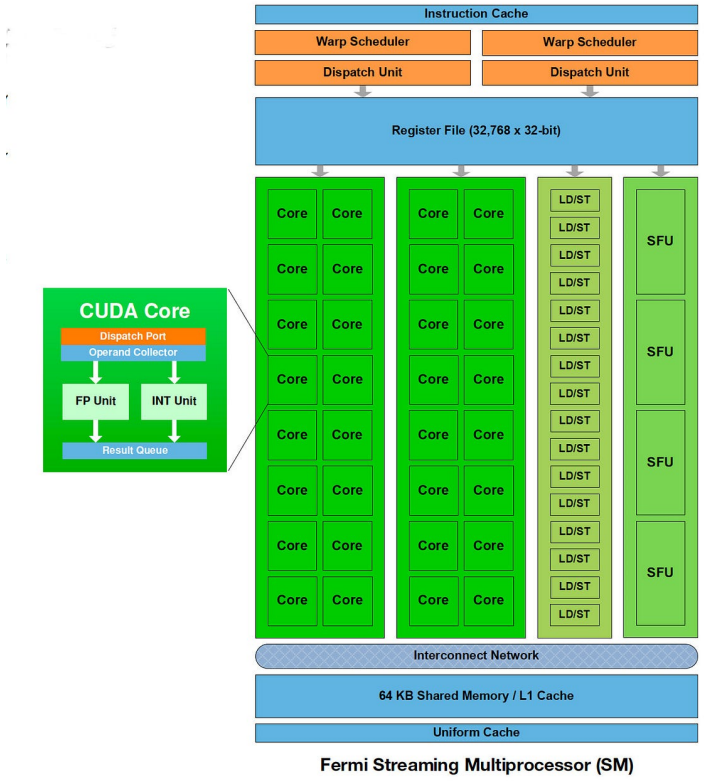


Figure 2.11: Units of a Fermi Streaming Multiprocessor

operations that require waiting for multiple clock cycles to complete.

- A single *Register File* shared by all threads and all warps running in the SM. Each thread can only access its registers.
- A single *Fetch and Decode unit*, since all the threads in the warp execute the same instruction in lockstep.
- A sufficient number of *Arithmetic Cores*, specialized for executing Floating point operations of various precision and integer operations.
- *Load and Store Units*, for writing and reading from and to the memories
- *Special Functional* Units that can calculate transcendent functions (for example *sin*, *log*)
- *On-Chip Memory*, that is partitioned between a simple cache and the *Shared Memory*, a sort of scratchpad memory available to the programmer

Unlike advanced modern CPU cores, the SM does not have fancy Branch Predictors, complex Cache hierarchy, and out-of-order execution units. The rationale behind this choice is that GPUs are throughput-oriented systems, so the SMs are kept simple in order

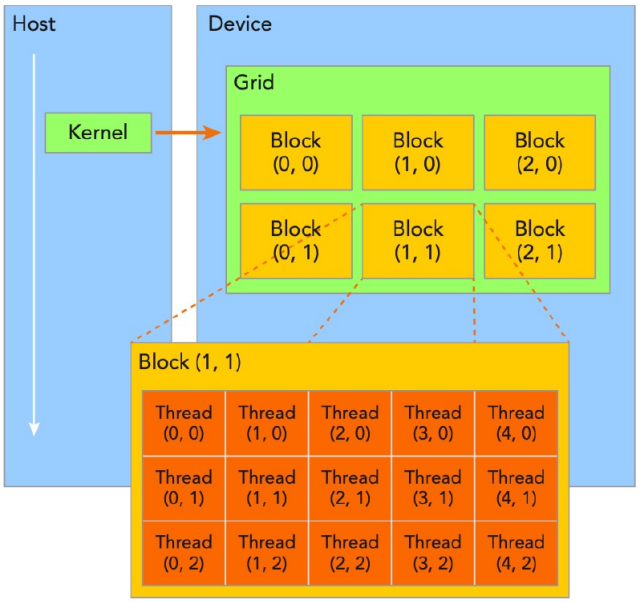


Figure 2.12: CUDA thread hierarchy

to be replicated as much as possible in the limited area available in the chip. Removing the advanced modules will remove the optimizations that keep latency lower, but it will save on the space occupied for the single SM in the chip. In this way, more instructions can be executed in parallel, and even if their latency is higher than in a CPU core, the number of instructions executed in the unit of time, hence the throughput, is bigger than a CPU.

The memory hierarchy of the GPU is composed of On-Chip and Off-Chip memories. The Off-Chip memory is a DRAM that is the memory with the biggest capacity in the GPU (in the order of GigaBytes). On the other hand, the DRAM has the highest access latency of all memories in the GPU. Modern architectures also include an L2-Cache shared by all the SMs. Besides the registers, each SM comes with fast 64 kB on-chip memory divided between the L1 cache and the Shared Memory. The Shared Memory is a low-latency scratchpad memory that can be used by the programmer to manually cache data to reduce the number of high-latency Off-Chip memory accesses. In section 2.4.4, we will see an example of using Shared Memory to increase matrix multiplication performance.

2.4.2. CUDA Programming Model

The *Compute Unified Device Architecture* (CUDA) is a general-purpose programming model that allows programming CUDA-enabled GPUs to exploit their parallel computing capabilities.

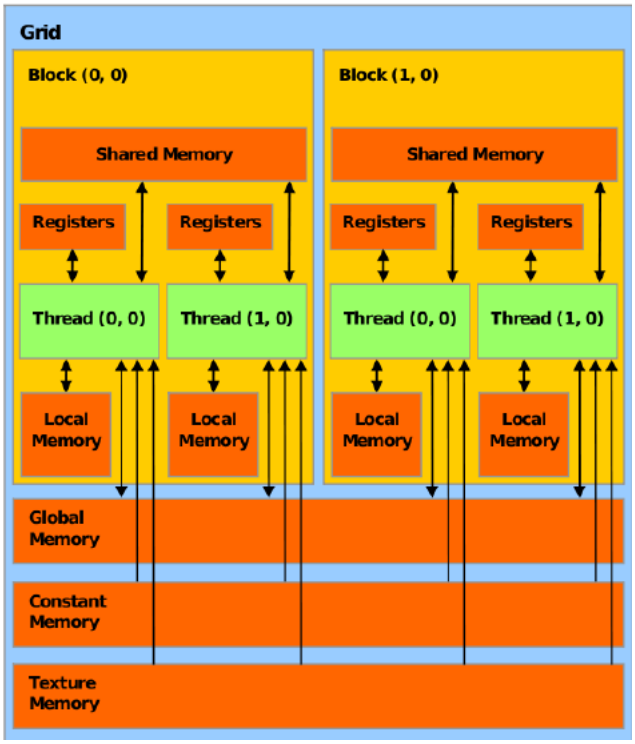


Figure 2.13: Logical view of CUDA memory model

The CUDA programming model is based on CUDA C++, which includes syntactic extensions to C as well as C++ libraries. A CUDA program starts running in the CPU (called *Host*), and from there, it can spawn simultaneously numerous threads running in the GPU, also referred to as the *Device*. As shown in Figure 2.12, threads are organized in multidimensional structures, up to three dimensions, called *Thread Blocks*. Blocks are also organized in another multidimensional super-structure called *Thread Grid*, which is launched from the host. The dimensionality of blocks and grids is customizable by the programmer who has to tune it taking into account the shape of the input, the implementation of the algorithm, and the characteristics of the device, with the objective of achieving the best occupancy rate of the device and so the best throughput and performance from the execution.

The CUDA programming model provides a complex memory hierarchy shown in Figure 2.13. The hierarchy features various logical types of memories each one with a different function. The most relevant for the objective of this work are:

- Registers and *Local memory*. They have the scope and the lifetime of a single thread, and they contain local variables and the actual parameters of the kernel call. Registers have the fastest access time (1 clock cycle), but their space is limited. If a variable can't fit in the registers (due to spilling or because it is a large array), it is

stored in the local memory that is stored in the off-chip DRAM, with big capacity (in the order of GigaBytes) but a very high latency (around 1000 clock cycles).

- *Shared memory*, which has the scope and the lifetime of a Thread Block. Shared memory is physically stored in the on-chip memory. The Shared Memory is used as a programmer-managed cache. It can be used to load from the DRAM data that needs to be accessed multiple times from the same block, saving costly off-chip memory operations.
- *Global Memory*, which has the scope and the lifetime of the entire application. Global memory resides in the device's DRAM, but the host can perform allocations and copy data on it from its own memory, using explicit functions from the CUDA runtime library. For example, the host can move input data coming from the application, and then launch kernel grids in the device; when the device terminates, the host can copy the output data in the global to the host memory and then use it in the application.

2.4.3. CuDNN

CuDNN is a CUDA library implementing the primitives used in CNN. Multiple widely used deep learning frameworks rely on cuDNN as their backend for GPU accelerations in NVIDIA devices. For Convolution, CuDNN provides different implementations for GEMM, FFT, and Winograd implementation described in Section 2.1.4.

The CuDNN implementations of Convolution and Batch Normalization make massive use of Shared Memory. For example, the GEMM implementation of Convolution uses two kernels: the `im2col` kernel prepares the matrices to be multiplied, and the `gemm` kernel performs the actual matrix multiplication.

2.4.4. Accelerating Matrix Multiplication

To show how shared memory can be exploited to increase performance, in this section, we will illustrate some basic GPU optimizations to accelerate the execution of Matrix Multiplications, a pervasive operation in the High-Performance computing world, including the domain of CNN.

Consider the multiplication of two square $N \times N$ matrices A and B, that yields the result matrix C (also a $N \times N$ matrix). The data of all matrices is stored in the global memory. In the naive CUDA implementation [4] of the Matrix Multiplication (MM), the thread with `threadId` (m, n) calculates $C_{m,n}$, where $0 \leq m, n \leq N$. $C_{m,n}$ is calculated by

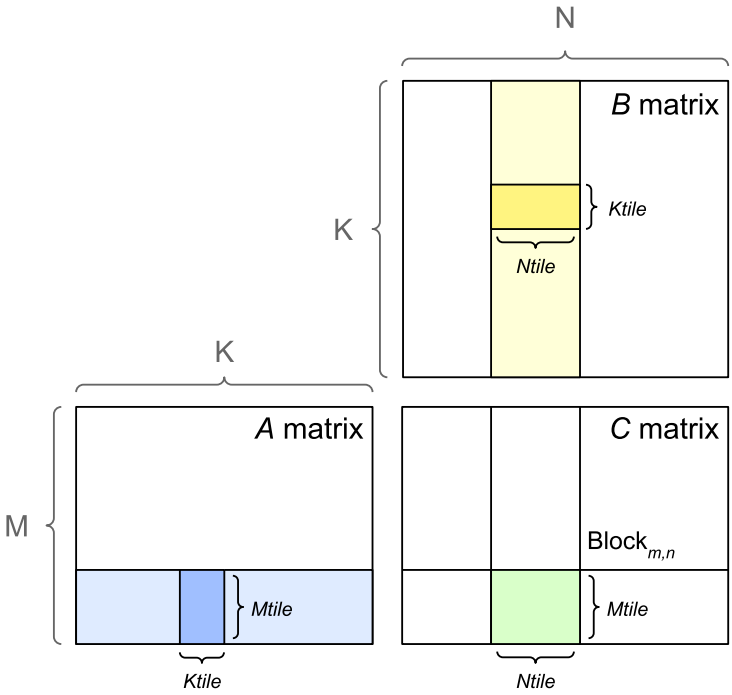


Figure 2.14: Tiled matrix multiplication

performing the scalar product between m-th row of matrix A and n-th row of matrix B. The bottleneck of the proposed implementation stands in the memory bandwidth. Each element of A and B is accessed from N threads, so there will be, in the entire matrix, N^3 reads from the DRAM. To reduce the number of long latency accesses to off-chip memory, the Shared Memory can be used to store parts of the operand matrices. Suppose the kernel is configured to launch using bi-dimensional threads blocks of size $T \times T$ threads. Without loss of generality, assume that $N = ST$ where $S, T \in \mathbb{N}$, to avoid remainder blocks. All the matrices can be partitioned in $S \times S$ tiles of $T \times T$ cells. We will indicate the tile (X, Y) of a matrix M with $M^{X,Y}$.

The tiled matrix multiplication algorithm using the shared memory follows these steps:

1. Launch a bi-dimensional grid of $S \times S$ thread blocks containing $T \times T$ threads.
2. In each thread initialize a local accumulator variable c to 0, and a cycle counter k to 0.
3. Consider now a generic thread block (X, Y). Every thread in the block will cooperatively load in the shared memory the tile $A^{k,Y}$ and the tile $B^{X,k}$.
4. All threads will synchronize using the thread block level barrier primitive, `__syncthreads()`, available in CUDA. A thread encountering a `__syncthreads()` in its execution will wait until all the threads in the block reach the same barrier point.

5. After the barrier, each thread, according to its position, will perform the scalar product between a row of the tile of A and the column of the tile of B, accumulating the result on c.
6. Before continuing to the next value of k and loading another tile from global memory, all the threads in the block will wait for each other again using `__syncthreads()`.
7. Steps 3-6 are repeated for $k = 0, 1, ..., S - 1$.
8. At the end of the loop in k, all threads will write the value of their accumulator of C in their designated location of the output matrix.

Figure 2.14 shows a freeze-frame of the tiled matrix multiplication. Note that a single tile in the output is obtained by multiplying multiple tiles from the operand matrices as described in the algorithm above. Using this algorithm, each cell of the matrix is only read from the global memory S times, and so the number of reads from the global memory, in the whole execution of the program, is $S \times N^2$, where $S << N$.

In this section, we observed that the use of shared memory-based optimizations, such as tiling, is fundamental to speeding up matrix multiplications. On the other hand, we will see in Section 2.5.2 that the same optimizations that make MM fast in the GPU reduce its resiliency to soft errors because Shared Memory is not protected from faults at the hardware level. Furthermore, the values stored in the unprotected shared memory are reused multiple times by multiple threads, meaning that a corrupted value in the shared memory can spread to numerous locations of the output matrix.

2.5. Faults and Reliability of CNN

With the advancement of technology, the density of transistors inside an Integrated Circuit (IC) continued to increase. A the same time, the operating voltage of ICs continued to be reduced. Smaller transistors supplied with lower voltages are more likely to be vulnerable to radiation-induced hardware faults [5] since less energy is needed to alter their functioning.

One of the most common factors that cause malfunctions in IC is the interaction of cosmic rays with the atoms in the atmosphere, generating showers of high-energy neutrons capable of affecting the functioning of integrated circuits. The phenomenon's intensity has a strong altitude dependency [28], reaching its peak between 10 to 40 km above the sea level. Outside of the atmosphere, in space, hardware faults are directly caused by ionizing radiation carried by cosmic rays originating from the Sun or distant galaxies. Unlike in

the Earth's atmosphere, radiation in space is not attenuated, making it more likely to cause *Single Event Upsets* (SEU), and prolonged exposure of non-shielded electronics to ionizing radiation can cause permanent damage.

2.5.1. Effects of Hardware Faults

The existing literature [16] distinguishes hardware faults based on their persistence. *Permanent Faults* (or Hard Faults) are irreversible physical changes in the circuit. Conversely, *Transient Faults* are temporary and reversible faults caused by environmental variations such as radiations, power supply voltage variations, or electromagnetic fields.

Transient Faults lead to temporary alterations in the state of the circuit called *Soft Errors*. One of the most common effects of a Soft Error is a transient bit-flip in a flip-flop that shifts the state of the cell from 0 to 1 or vice-versa. A write to the memory cell restores the normal functioning of the circuit. The effects of a soft error depend on various factors, such as the executed program, the time when the soft error happens, or the location of the targeted flip-flop. These effects are classified based on the difference between the expected output (called *Golden Output*) and the produced output. The three classes are:

- *Silent Data Corruption* (SDC): The program terminates without errors, and the produced output differs from the golden output.
- *Masked*: The program terminates without errors, and the produced output is equal to the golden one.
- *Detected Unrecoverable Error* (DUE): The application hangs forever and requires a restart or throws an exception without returning any input.

Masked errors and DUEs are not very dangerous for the reliability of the system, since Masked errors do not produce any tangible effect and a DUE is detected by the system, which can decide what action to take, such as restarting the program, rebooting the system or issuing an alert to the user. On the contrary, SDCs can be very insidious since when they happen the program terminates with a wrong output without any part of the system knowing that a fault happened. Consider a CNN part of a critical system, such as a collision avoidance system on board an autonomous vehicle. An SDC happening during the execution of the CNN leads to two possible outcomes: the SDC can corrupt the output in a way that makes the system fail, causing a *Critical Fault*. On the contrary, the SDC may corrupt the output in a way that does not create a failure in the system. In this case, we will have a *Non-Critical Fault*.

The proposed classification heavily depends on the use case and the implementation of the

system. For example, consider a very simplified hypothetical collision avoidance system in an autonomous vehicle that employs an object detection CNN for triggering an emergency braking action if it detects an obstacle (such as a pedestrian, another vehicle, or an object) in the collision course of the car. Suppose an SDC causes the CNN to omit the bounding box generation around a pedestrian that is dangerously crossing the road in front of the vehicle (false negative). In that case, the system fails to activate emergency braking, and the fault must be considered critical because it creates a dangerous situation. Conversely, the fault may be considered non-critical if the bounding box is misplaced in a way that the system still triggers an emergency right on time.

This classification of the fault is often referred to as *Usability Based Classification* [6], since it evaluates whether the corrupted result is still safely usable in a critical system.

2.5.2. Faults in the GPU

Modern GPUs' performances are the result of the progress that has been made over the years in the field of ICs. The high number of transistors in the GPU constitutes a large attack surface for neutrons and radiation, motivating all the research around this topic. Error-correcting codes (ECC) are implemented in the DRAM memory in most of the modern GPUs [20], among other things, to try to mitigate the effect of Soft Errors. The simplest protection is a parity bit added in every word of the memory and checked at every memory operation. NVIDIA GPUs implement in the DRAM a more complex ECC scheme called SECDED, which is able to correct errors when one bit is corrupted and to detect and retry the transfer when two or more bits are corrupted. Unfortunately, this mitigation is not applied to On-Chip memories and the control logic.

Soft errors can also happen in registers inside the control units of the GPU [14], such as the warp scheduler. A fault in one of those registers can bring the scheduler into an inconsistent state, causing DUEs, or it can silently corrupt the execution of an entire warp, leading to a severe SDC.

These facts combined motivate a more accurate study of the reliability of CNNs, focusing on the effects of faults in the control logic and the parts of the computation that extensively use the on-chip shared memory, like convolution and batch normalization layers.

2.6. Techniques for Reliability Analysis

In this section, we will analyze the literature about the different techniques for analyzing the reliability and resilience of CNNs running on GPUs. The insights gathered from the

analysis are helpful for understanding which part of the CNN is more vulnerable to faults and how to design hardware and software hardening strategies to increase the reliability of the CNNs. The same analysis tools are then insightful for benchmarking the effectiveness of the proposed hardening designs.

2.6.1. Radiation-Based Methodologies

Radiation-based methodologies use laboratory equipment, such as Neutron Beam emitters, to irradiate the Device Under Test (DUT) while running CNN inferences.

In [17], the authors perform experiments, irradiating three different GPUs models (a K40 Kepler, an embedded Tegra, and a Titan X), running object detection networks, with a Neutron Beam emulating the energy spectrum of a neutron flux present in the atmosphere. Each hour of test performed is equivalent to roughly 100 years of atmospheric exposure. Tests were performed in scenarios with ECC off and on and applying an Algorithm-Based Fault Tolerance (ABFT) method that adds a checksum row and column to all Matrix Operations.

Radiation-based methods achieve the highest level of precision [45] by causing the actual physical phenomenon that causes hardware faults in the DUT and, as discussed before, allow to obtain hundreds of years of data in a few hours of experiments. As contraries, the costs of experiments are very high due to the need to access specialized facilities. Also, the performed tests are destructive because the radiation usually damages the device permanently. Radiation tests also have a low *Controllablity* [47], namely the researcher's ability to decide where and when a fault is injected.

2.6.2. Platform Based Methodologies

Platform Based methodologies [47] assess the reliability of the target CNN by running it in the target physical platform and injecting faults using development tools for the platform.

Fault injectors in the GPU share similar structures and follow common steps. The initial step, called the *Profiling Phase* involves the injector identifying suitable locations for injecting faults, thus generating a list of potential injection sites. Depending on the injector, these sites could be executable traces, assembly instructions, or lines of source code. In the *Injection Phase*, sites are extracted from the list to be injected. For each extracted injection site, the program is executed normally until the fault location is reached, then, the value is replaced with a corrupted one, and the execution is resumed. In the following

paragraphs we will present the most relevant platform-based methodologies for this work.

CUDA-GDB Based Injectors

It is possible to create a fault injector starting from an existing debugger. For instance, CUDA-GDB, a debugger developed by NVIDIA inspired by GDB, can also be utilized as a GPU fault injector. By setting a breakpoint at the desired injection site, the developer can alter the value of the target variable (for example, by flipping a bit) and then resume the execution to see the effect of the fault. A debugger like CUDA-GDB does not feature a built-in profiler, so the programmer needs to write one. For using the debugger, it is necessary to include in the executable the symbols and to turn off compile time optimizations. The execution becomes particularly slow when the breakpoint is reached, and the execution must be single-stepped.

GPU-Qin [19] is a fault injector for GPU based on CUDA-GDB. GPU-Qin performs a first phase where it groups threads with similar behavior (non-divergent threads) and then profiles a representative thread for each group. Then GPU-Qin runs the injection by selecting one instruction from the profile, adding a GDB breakpoint in the position of the instruction. When the break-point is reached, the execution is single-stepped until the program counter reaches the target instruction. At that point, GPU-Qin injects the faults. Available targets for injections are arithmetic instructions (in their output value), memory instructions (in their output value or output address), and Control-flow instructions (in their input operands).

SASSIFI

SASSIFI [24] is a microarchitectural fault injector for GPUs, developed by NVIDIA, supporting *Fermi* (released in 2010), *Kepler* (2012) and *Maxwell* (2014) GPU architectures. The tool is based on SASSI, a low-level compiler base instrumentation tool, executed at the final pass of the backend compiler, that compiles in the *ptxas* intermediate representation into SASS, the low-level assembly for NVIDIA GPUs. In the profiling phase, SASSIFI uses SASSI to insert, after every targettable instruction, a call to a handle function that registers and collects information about the injection site. SASSIFI then randomly samples the list of all collected injection sites. During the injection phase, it injects a single fault per execution by inserting a call to a handler function performing the injection before and/or after the target instruction.

In detail, SASSIFI has three injection modes: Register File (RF) mode alters the content of the register file before the value is used in the instruction, Instruction Output Value

Table 2.1: NVBitFI possible outcomes after an injection

Outcome	Symptom Description
SDC	Standard output is different
DUE	Timeout Process crash Non-zero exit status Application-specific check failed
Masked	No differences detected
Potential DUE	(SDC or Masked) with CUDA error (SDC or Masked) with dmesg error

(IOV) alters the result of an instruction, and Instruction Output Address (IOA) alters the destination address of the output of a load/store instruction. SASSIFI also allows the user to pick which instruction groups to target (arithmetic, load/store, predicate) and to choose how to alter the value, offering different bit-flip models (Single-bit flip, double bit-flip, zero or random value). Since SASSI performs only static instrumentations, the whole codebase must be recompiled from scratch whenever the injection mode has to be changed. Also, the library dependencies of the target need to be recompiled in order to be used in SASSIFI, making SASSIFI unfeasible to use when the codebase includes closed-source libraries such as CuDNN or big libraries that have long compile times.

NVBitFI

NVBitFI [53] is a microarchitectural fault injector designed by NVIDIA; it is the designated successor of SASSIFI supporting the more recent GPU architectures between *Maxwell* (2014) and *Ampere* (2020). NVBitFI is based on a newer instrumentation tool called NVBit. A significant improvement of NVBit, compared to SASSI, is the possibility to perform dynamic code instrumentation, allowing to intercept dynamic kernel calls (kernel calls inside other kernels) and inserting handle functions at runtime without requiring a recompilation. This improvement enables us to perform fault injections in codebases that include closed source and/or large libraries such as CuDNN, differently from SASSIFI.

NVBitFI profiler and injector are implemented as two separate modules, compiled into two different shared libraries. The profiler dynamically detects the kernels and the injectable instructions inside them and saves their information in text files. The injector randomly samples some sites and performs one injection per program run. After the execution, NVBitFI checks the outcome of the injection. In Table 2.1, all the possible outcomes of an injection are illustrated.

At the writing date of this work, NVBitFI supports only the IOV injection mode and allows to inject only in three instruction groups (arithmetic, loads, and register operations). The fault models offered by NVBitFI are single-bit flips, two-bit flips, zeros, and a random value generated by randomly extracting bit by bit. The authors of [18] proposed and implemented a patch to NVBitFI that allowed to inject in the output value of 32 threads of a warp, simulating a soft error that happened inside the control logic.

2.6.3. Hardware Simulation Methodologies

Hardware Simulation methodologies aim to analyze the resilience of a CNN by injecting faults in a simulated version of the HW architecture. The target hardware architecture is usually specified with a Hardware Description Language (HDL), or it can be simulated with code written in a programming language. Simulations can be performed at the gate level or at the Register Transfer Level (RTL).

With a hardware simulation, it is possible to inject faults of various types (soft errors, stuck-at) with a very high level of controllability and the possibility to inject faults in hardware modules that cannot be accessed by the programmer, such as the control units. The results of the injection campaigns reach a high level of accuracy.

The main drawbacks of this methodology are:

- Simulations require a huge amount of computational resources, especially for GPUs that have a lot of replicated modules.
- GPU cores hardware descriptions are rarely released to the public to protect business-sensitive information

The first drawback can be mitigated by simulating only the faulty component, restricting the simulation to only the time necessary for the injection, and running the rest of the computation at the application level. This strategy is used in multiple cross-layer methodologies, as we will see in Section 2.6.5.

One of the few available open-source GPU models for hardware simulation and reliability analysis is FlexGripPlus [13], an implementation of the NVIDIA-G80 microarchitecture described in VHDL. The model is compatible with CUDA Streaming Multiprocessor architecture 1.0.

2.6.4. Application-Level Methodologies

Application-level methodologies analyze the resilience of the deep learning framework under test, ignoring the underlying hardware platform. These methods inject fault at the dataflow graph level and study the impact of errors corrupting either the weights of the model or the output of the operators.

The main advantages of this approach are: the ease of developing directly on a high-level framework and the small time required to perform an injection, compared to the other class of injectors. However, these methodologies may lack accuracy due to the aggressive abstractions used from high-level frameworks.

TensorFI [9] is a high-level error simulator designed specifically for TensorFlow. TensorFI operates at the application level abstracting the underlying architecture.

TensorFI provides a function that takes in input a tensor flow graph and some configuration parameters read from a configuration file. The method then replicates the graph and inserts after the targeted operators a special node that is able to corrupt its output.

The fault injection is performed in the instrumented graph returned from the function that can be executed as a normal TensorFlow graph, either in the CPU or in the GPU. Its output can be compared with the golden one to check if an SDC happened or not.

From the configuration file, the user can customize the injection by choosing the fault model (single-bit, random value, zero), the operators and the layers to target, and how many injections per run to perform.

PyTorchFI [35] is a similar fault injector but implemented for the PyTorch library. Due to the dynamic nature of the PyTorch computation graphs, instead of instrumenting the graph directly with error-injecting nodes, PyTorchFI performs a profiling run to get the structure of the graph and then adds the node to the target operators.

2.6.5. Cross Layer Methodologies

Cross-layer methodologies try to combine the advantages of the previous methodologies by splitting the analysis into two steps. In the first step, the hardware platform-specific fault injection or radiation testing activity is carried out on the part of the whole CNN, collecting all the data needed for executing the second step, where the observed results are used in an application-level analysis and simulations.

Cross-layer methodologies basically assign the right task to the right tool. The costly low-level hardware-bound injectors or radiation tests are good at capturing accurate error

models, while the high-level frameworks are the best choice for running complex CNNs efficiently.

In this section, we will discuss some Cross-Layer methodologies for analyzing the reliability of CNNs running in the GPU. CLASSES [8] is a Cross-Layer framework, but since this thesis work is devoted to proposing an improvement to it, we will discuss it in a dedicated section in the next chapter.

A Multi-level Approach to Evaluate the Impact of GPU Permanent Faults on CNN's Reliability

In [15], the authors propose a three-level methodology to evaluate the effects of hardware permanent faults in the functional units of a GPU. The proposed methodology involves platform-level profiling followed by a hardware-level simulation combined with a final application-level analysis.

In the first step, the application is profiled, using NVBit, to trace all the executed instructions inside GPU kernels, generating a report containing, for each instruction, data about the operands and the functional units utilized. As the second step, the data in the report is then used to carry out gate-level simulations of the functional units inside the GPU. The simulations are performed on a synthesized version of the FlexGripPlus model, injecting stuck-at faults in various parts of the circuit. The output of those simulations is post-processed to identify the visible effect at the application level.

Finally, the observed erroneous behaviors are used in an application-level fault simulation to analyze how faults propagate among the layers of the CNN. This enables a detailed analysis of the vulnerability of every layer in the considered CNN.

Combining Architectural Simulation and Software Fault Injection for a Fast and Accurate CNNs Reliability Evaluation on GPUs

The authors of [14] noted that the state-of-the-art platform-based fault injectors (SASSIFI and NVBitFI) are not able to inject faults in the user-hidden modules of the GPU, such as the warp scheduler or the pipeline register, even if these modules are also vulnerable to soft errors. Furthermore, as we discussed in Section 2.5.2, a single fault in the control logic is able to corrupt the output of entire warps at once.

As a response to these issues, the authors proposed a two-step reliability analysis framework that is able to take into consideration also faults in the control. The first step consists in running fault injections at the Register Transfer Level (RTL) using the FlexGripPlus

model in order to have full visibility of the architecture.

Running an RTL simulation for an entire GEMM convolutional layer is computationally unfeasible, even for networks with relatively small feature maps such as LeNet-5. As a workaround, the authors took advantage of the fact that tiles in a matrix can be computed independently from each other (as seen in Section 2.4.4) and performed the simulation only in a single 8×8 tile, using different value distributions for the input values.

In the second step, a classification CNN (in this case LeNet) is executed on a GPU normally until the desired faulty layer is reached. The corrupt tile is merged together with the golden matrix at its correct position, and the inference is executed to completion. In the end, the output is compared with the golden, where three outcomes are possible: Masked, SDC (different output but no misclassification), and critical SDC (misclassification).

The results of the experiments concluded that few cells in the warp scheduler were responsible for the vast majority of critical SDCs.

2.7. CLASSES

CLASSES [8] is a Cross-Layer framework for analyzing the reliability of a CNN running in GPUs. The framework combines platform-level fault injection with an application-level error simulator, interfacing the two layers with error models extracted from the fault injection experiments and reused for generating similar errors in the application-level simulation. The input of the framework is the network graph of a CNN application to be executed in a hardware platform. The framework's output is a reliability report containing metrics about the network's resiliency to faults in the given device. The user establishes which metrics want to obtain in the report also based on what aspects of the network they want to analyze considering the nature of the task performed by the CNN, since different networks have different performance metrics as discussed in Section 2.3.

The strong motivation behind adopting a potentially more complex cross-layer structure in a reliability evaluation framework stands in the possibility of combining the advantages of two frameworks that act at two different layers, hiding their disadvantages. Platform-level fault injection frameworks represent with high fidelity the effects of the emulated faults, but every single injection is slow compared to the normal execution time of a network, especially if the network is complex and composed of a large number of layers. On the contrary application-level error simulators alone are not able to simulate realistic error models, but their simulations are faster and more controllable. In synthesis, the main idea behind the CLASSES methodological framework is to perform fault injection

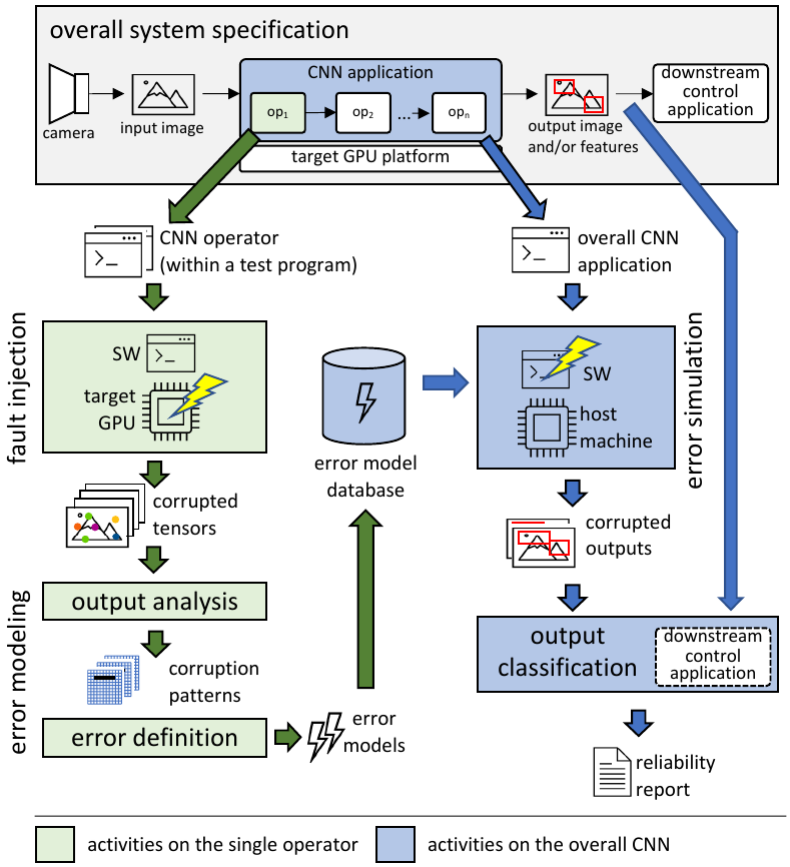


Figure 2.15: CLASSES cross-layer framework architecture

campaigns at the platform level, injecting one operator at a time and collecting each campaign's corrupted outputs. All the faulty tensors are then analyzed using a semi-automatic procedure to produce error models that will be used to simulate errors at the application level.

Figure 2.15 illustrates the architecture of the CLASSES framework, showing the three important phases of this methodology:

- Architectural fault injection
- Error modeling
- Application-Level error simulation

In the next three subsections, we will make a short introduction to the fundamental concepts of these three phases, explaining how they worked in the original version of CLASSES.

2.7.1. Fault Injection

The centerpiece of the fault injection phase is a fault injector, which is an external module to CLASSES. Since this methodology is fault-injection agnostic, it is compatible with multiple fault injectors that must respect the following requirements to be compatible with CLASSES:

- It must be compatible with the target architecture, the object of the reliability analysis
- It should not be a software-only error simulator that completely abstracts from the hardware architecture.
- It is able to perform injections in single isolated operators of the CNN
- The outcome of a fault injection can be reconducted to one of these three outcomes:
  - Masked: The program terminates without errors, and its output is a tensor equal to the golden one.
  - DUE: The program terminates with an error or without returning any input.
  - SDC: A silent data corruption happened and so a corrupted tensor is returned
- Optionally the injector may have internal configuration parameters such as:
  - *Fault models*: Specifies how the targeted values are changed. Some examples of models are single bit-flip, double bit-flip, and a single random value. Error models can also target multiple values, such as all values written by a warp in an instruction.
  - *Target Locations*: Specifies which instructions or functional units are targeted by the fault.
  - *Injection Mode*: Allows to select one of the different possible ways to inject an error (if more than one injection method is possible.)

Once an injector is chosen, the fault injection phase begins by extracting the types of operators present in the network graph of the CNN to be analyzed. The central part of this phase consists of planning and executing fault injection campaigns for each type of operator present in the CNN and collecting the results where an SDC happens to be analyzed. The corrupted tensors, compared to the golden tensors coming from a faultless execution, will have one or more erroneous locations, where the values differ from the golden ones with an absolute value bigger than ε, the maximum level of tolerance for

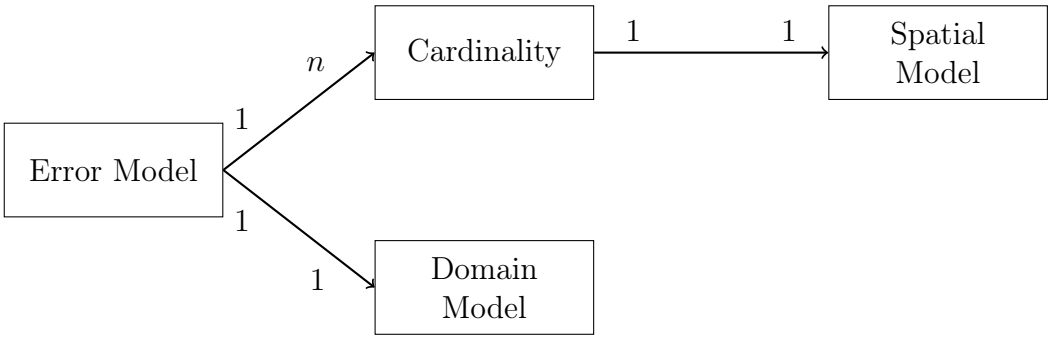


Figure 2.16: High-level structure of a legacy CLASSES error model

which two values are considered equal (usually is 10^{-3}). The location and the domain distributions of these erroneous values will be the object of the analysis of the error modeling phase, which aims to build an error model for each one of the operators of the CNN. In Section 3.1 we will better explain how this phase is structured and the improvements made to the method in order to tackle the limitations of the current CLASSES version.

2.7.2. Error Modeling

The role of the error models in the CLASSES framework is to store, in a structured way, the information on how the faulty tensors coming out from a fault injection campaign are corrupted. The application-level error simulator, to generate the errors reads and interprets the error models that contain information about the spatial and domain distributions of the erroneous values present in the faulty tensors obtained in the fault injection phase.

As shown in Figure 2.16, in the original CLASSES version the error models are structured in two sub-models as follows:

- *Domain Models* contain information about the values present in the faulty locations that are classifiable in five different value classes:
  1. *NaN*: The floating point representation of the value, according to the IEEE 754 standard [2], represents a non-valid number, or infinity.
  2. *Zero*: The erroneous is zero, differently from the golden value
  3. *Bitflip*: The floating point binary representation of the erroneous value and the golden value differ by a single
  4. $[-1; 1]$: The absolute value of the difference between the golden and the erroneous value is less or equal to 1.

Algorithm 2.2 Tensor Spatial Classification

Inputs

CorruptedTensor: A corrupted tensor *GoldenTensor*: A golden tensor

SpatialClassesNames: List of the names of the spatial classes sorted by priority

IndicatorFunctions: List of the indicator function of the spatial classes sorted by priority

ParameterFunctions: List of the parameter function of the spatial classes sorted by priority

ε: Tolerance level for erroneous values

Outputs

SpatialClass: The name of the spatial class

SpatialParameters: Key-value map containing the parameters of the class

```
1: ErrorsTensor ← |CorruptedTensor − GoldenTensor| ≥ ε
2: ErrorList ← CoordinatesWhereTrue(ErrorsTensor)
3: for i ← 0 to SpatialClassesNames.length − 1 do
4:     Match ← IndicatorFunctions[i](CorruptedTensor.shape, ErrorList)
5:     if Match then
6:         SpatialClass ← SpatialClassesNames[i]
7:         SpatialClassParameters ←
           ← SpatialParameters[i](CorruptedTensor.shape, ErrorList)
8:         break
9:     end if
10: end for
11: SpatialClass ← "Uncategorized"
12: SpatialClassParameters ← nil
```

5. *Uncategorized*: Any other case that does not fall in the precedent classes

- *Spatial models* contain information on how the erroneous locations are distributed in the tensor. The spatial models group together faulty tensors based on three characteristics of their error spatial distribution, storing for each group the percentage of all faulty tensors belonging to the group. The three characteristics that define a tensor group are:

  1. Error cardinality: The count of the erroneous values in the tensor
  2. Spatial Class: One of the user-defined classes that describe the kind of spatial pattern observed in the tensor
  3. Spatial Parameters (Error Vectors): A vector that indicates the linearized offsets of the errors relative to the first error present in the tensor

While the definition of the domain model can be performed fully automatically by a simple

script, the definition of the spatial model is a more articulated iterative task that requires the human in the loop. The manual part requires the user to visually observe the faulty tensors coming from the campaign and at each iteration define a new spatial class with its parameters. At the beginning of the process, all tensors fall in the default *Uncategorized* class and then, iteration by iteration most of the tensors will fall into a spatial class.

The original version of CLASSES does not provide a unified tool to perform the classification but still provides an algorithm, similar to Algorithm 2.2 for performing the spatial classification. The algorithm if implemented in useful for speeding up the spatial model definition, takes in input a corrupted tensor, its golden counterpart and a list of class definitions and returns the class where the tensor belongs with his parameters. The classes are characterized as follows:

- A name to identify the class
- An *Indicator Function* taking in input the coordinates of the erroneous values and the shape of the tensor and returning true if the tensor belongs to the class or false otherwise.
- A *Parameter Function* taking in input the same parameters as the indicator function returning the error vector associated with the tensor if it belongs to the class
- A unique non-negative *Priority Value* useful if a tensor matches the description of two or more classes. The class with the lowest priority will be the one where the tensor belongs

In this section, we gave an introduction to the fundamental concepts of error modeling and how the models are structured in the original version of CLASSES. In Section 3.2 we will examine the limitations of the current error models and we will propose some improvements explaining them in detail, while in Section 3.3 we introduce a unified tool for partially automating the error model definition process, based on Algorithm 2.2.

2.7.3. Error Simulation

The error simulation phase takes place at the application level by modifying the programs that run inferences of the CNNs using high-level libraries, such as TensorFlow and PyTorch. The architecture of the application-level error simulator is composed of two main highly decoupled modules:

- The *Error Generator* imports the error models and, given the specifics of a target golden feature map (such as its shape and the layer where it comes from), generates

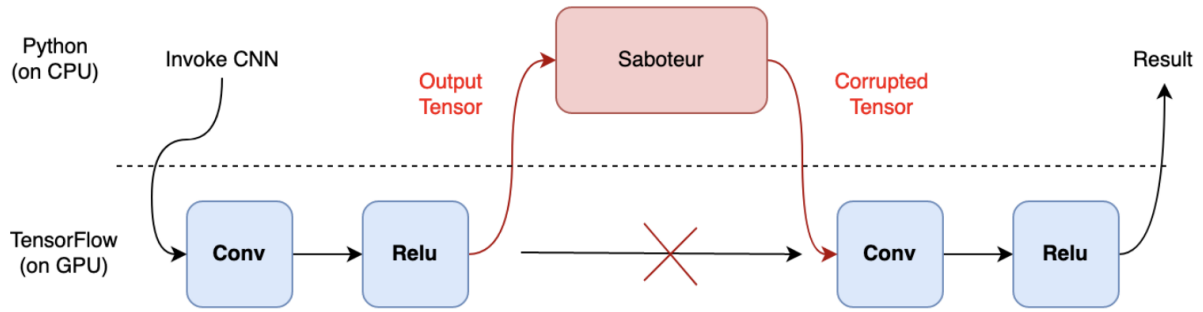


Figure 2.17: Example of the action of a Saboteur

a list of locations (tensor coordinates) to corrupt and for each corrupted location generate a value to corrupt according to the domain models. For generating the locations the user needs to specify one generator function per spatial class that extracts the corrupted locations respecting the spatial class definition and its parameters. These functions are conceptually the inverse of the spatial class indicator functions defined in the error modeling phase, which instead return the spatial class from a pattern.

- The *Simulator* applies the output of the error generator to the target feature map. This module interacts with the framework internals, inserting a *Saboteur* just after the output of the target layer. Practically speaking, as shown in Figure 2.17, the saboteur is code that runs in the CPU that calls the error generator module supplying to it the error models relative to the operator of the target layer and then applies to the output feature map the generated errors.

While the Simulator is strictly coupled to the DL framework for which it was designed, the error generator is not aware and completely decoupled from that framework. In the current version of CLASSES simulators for both Tensorflow and PyTorch are available. With the architecture explained above, a single error simulation works as follows:

1. The CNN runs the inference normally, using the high-level framework used for implementing the network.
2. The inference is paused once the output tensor of the target layer is computed.
3. The error generator module of the simulator extracts randomly an error cardinality, a spatial class (depending on the cardinality) and one of its related parameters following the probabilities specified in the error models.
4. The error generator extracts a list of erroneous locations, with a length equal to the cardinality. The location must be generated according to the spatial class and

parameters extracted, using the code pre-written by the user to generate the pattern.

5. For each one of the erroneous locations the error generator picks a value based on the old value and an extracted value class following the probabilities specified by the error model.
6. The errors are applied to the output feature map of the target layer extracted by the simulator.
7. The inference resumes until the output is reached.

At the end of the simulation, the corrupted output is compared with the golden one. Making a simple bitwise comparison between the two outputs is not effective for the sake of a reliability study, since there could be outputs that, while being different from the golden output would not cause any harm in a real-world application. So, as discussed in Section 2.5, we will adopt the critical/not-critical classification (also as Usability-Based classification).

In conclusion, this chapter has laid the foundational groundwork for understanding the contribution of this work. We started by formally introducing CNNs, ways to implement them and we examined a couple of real world networls. Then we introduced the key hardware component in the world of CNN, the Graphic Processing Unit (GPU) noticing that equally to all computing devices, if not more, can be subject to faults caused by radiation. After defining when the faults can have critical effects on digital systems we surveyed existing reliability analysis frameworks, setting the stage for our in-depth exploration of the improvements that we will introduce in the following chapter.

3 | Methodology

In this chapter, we will propose a methodology for modeling the errors caused by soft errors. The proposed methodology builds on to the one presented in CLASSES [8], introducing some innovations and improvements by proposing a new systematic approach to the planning of the platform-level fault injection campaigns, a new structure of the error models that tackle the limitation of the current version of classes and a unified tool that helps speed up the error modeling phase which up to now is performed manually by the user. These changes have the goal of making the process more efficient and streamlined, making CLASSES more flexible and re-adaptable to new scenarios and improving the fidelity of the generated errors in the application-level error simulation.

This chapter begins with Section 3.1 where we will discuss in detail the new, improved, architectural fault injection phase. In Section 4.2 we discuss the novel error model structure, improving both the spatial and domain side of the error models with respect to CLASSES. Finally in Section 3.3 we formulate the goals and requirements of a unified tool that the user can employ to streamline the error modeling process.

3.1. Improvements to Architectural Fault Injetion

In this thesis, we propose an innovative, formalized approach for planning and executing fault injection campaigns with the goal of generating corrupted tensors that will be used to generate an error model for each one of the operators of the CNN. This section begins with a description of the limitations encountered in the fault injection phase and from there describes, one per subsection, the new, revised planning and execution stages of the fault injection phase. The section concludes with some remarks about the size of the experimental campaigns.

3.1.1. Limitations in CLASSES Fault Injection phase

For what concerns the architectural fault injection phase of CLASSES, explained in Section 2.7.1, we noticed some limitations that we are going to discuss in this section. In

the original work, the formalization of the process of operator extraction and planning was quite rudimentary and not always easy to extend to a general case different from the experimental environment proposed in the original work. Secondly, in the original version of CLASSES, the experimental campaigns that were planned for each operator injected faults only on the instance of the operator with the smallest output tensor size, intending to shorten the injection time. This choice, of course, produces very precise error models at the application layer, when simulating errors in tensors of that specific size, but the error models may not correctly represent the errors for bigger feature maps, since no experiments were carried out for this size, and scaling the error models to the bigger channels maps may not be realistic. A cause of this problem comes from how low-level DL frameworks apply optimizations transparently to the user, often deciding which optimization to apply depending on inputs, parameters (such as the operand sizes) and the architecture. For example, a common optimization used by GPU libraries like CuDNN is kernel fusion. When a library function has to call multiple CUDA Kernels, like in the case of a GEMM or FFT convolution, some of the kernels can be fused together to reduce the number of global memory operations [1] [27]. Consequentially, kernel fusion alters the memory access patterns and the spatial corruption patterns in the event of a fault. Since the effectiveness of the optimization depends on the size of the operands and the other parameters, the function has some internal logic where it dynamically decides whether to apply the kernel fusion or not, meaning that the error patterns may be radically different depending on the internal optimization that cannot be controlled by the user. In the following sections, we will define how to structure the fault injection phase tackling the limitations discussed above.

3.1.2. Operator Extraction

The first step of the methodology consists of the user extracting a list of all the types of operators present in the CNN graph, this step can be easily executed using the library functions available in all frameworks to inspect the network graph. Within the list, the user decides for which operators they want to generate the error modeling performing a fault injection campaign on it. The user is free to decide to not inject all the operators, or to perform injection campaigns on multiple versions of the operators, such as in the case of the convolution where multiple strategies are available. The final product of the operator extraction phase is the plan of the fault injection campaigns to execute with each campaign characterized by:

- The operator targeted by the fault injection.

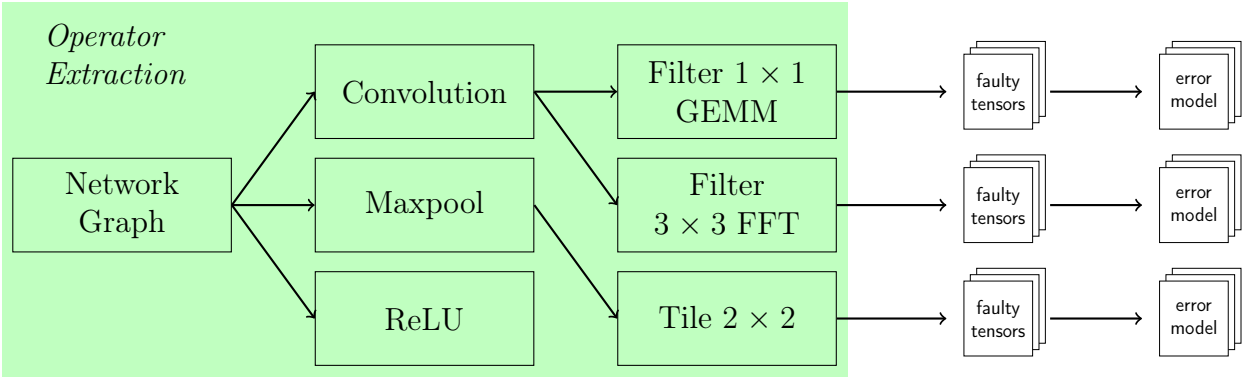


Figure 3.1: An example of the Operator Extraction step

- The eventual high-level strategy used for the operator (if applicable).
- A list of *Fixed Variables*, the variables that remain constant in all the injections that will be performed. These can be hyperparameters and other experimental configuration variables that the user wants to be constant in all experiments.
- A list of *Free Variables* that will vary through the experiments.

Figure 3.1 shows an example of the operator extraction step. From the network graph, three different types of operators are extracted. For the convolution, the user decides to split the operator into two sub-operators having a convolution with 1×1 kernel using the GEMM strategy and another one with 3×3 filter using the FFT strategy. In this case, the fixed variables are the hyperparameters determining kernel size. Analogously, the user performs one campaign for the max pooling operator, setting the tile size to 2×2. For the ReLU, no experiments are conducted. The final list of operators to inject has three elements, so the user will perform three injection campaigns, each leading to generating one error model each.

We introduced the concept of *fixed* and *free variables*, which was not present in the original version of classes, with the goal of better rationalizing the operator extraction and experiment campaign planning. Previously all the hyperparameters were fixed especially the shapes of the operand tensors and consequentially the shape of the output tensors. As discussed in Section 3.1.1, this caused problems with the scaling and reusability of the error models. To solve this problem we introduced the possibility of making some hyperparameters vary through a single campaign, hence we decided to introduce and formalize the concept of free variables. We will delve deeper into this aspect in the next section where we are going to discuss de design of a single injection campaign.

3.1.3. Campaign Design

As discussed in the previous section, a campaign is defined by the operator, its high-level strategy, and its fixed variables. This section will explain the design of a single experiment campaign, discussing which choices should be made for the free variables, a choice that the user has to make at the stage of the single campaign planning. Here is a non-exhaustive list of some of the possible free variables of the experiments:

- *Hyperparameters*: The output shape, the operands shape, and the values of other configurable parameters of the operator that are not fixed variables of the experiment.
- *Input Values*: The values that populate the input tensors
- *Injector Configuration*: The different configurations available in the fault injector

Differently from the original version of CLASSES, in this new version of the methodology, we propose to consider the output feature map size as a free variable, trying to inject faults in output tensors of multiple sizes, in this way we can mitigate the issue discussed in Section 3.1.1. The actual choice of which output sizes to select is purely empirical and is left to the designer of the experiments, who decides the feature map sizes to be picked for the test by observing the network topology. Given that most network topologies follow the principle of *Dimensionality Reduction* (discussed in Section 2.1.2), it is advised to choose tensors with few channels and big images (that are usually at the beginning of the network) and a tensor with more channels and smaller images (usually present at the end of the network). Regarding the input values, we propose to use in each campaign a few (2 to 4) different input tensors. The input tensors may be generated from a known distribution of the values populating the feature maps and the weights or by extracting the feature maps during an inference of the CNN under test. Performing experiments one more than one combination of inputs reduces the dependency of the error models from a particular input, alleviating the chances of having skewed error models.

Figure 3.2 shows an example of how a campaign can be designed. The combination of the choices made on the free variables forms a tree. Each level of the tree is a free variable, and the nodes on that level are all the choices for that free variable made by the user. A walk leading from the tree's root to a leaf is a particular choice of free variables. We call each leaf of the tree an *Batch of experiments* inside a campaign. The small example shows a campaign that, as fixed variables, has the kernel size (3×3), the strategy (GEMM), and the stride (1). The two free variables are the output tensor size and the input values. The user first decides on three tensor sizes to test, and then for each one, it picks two sets

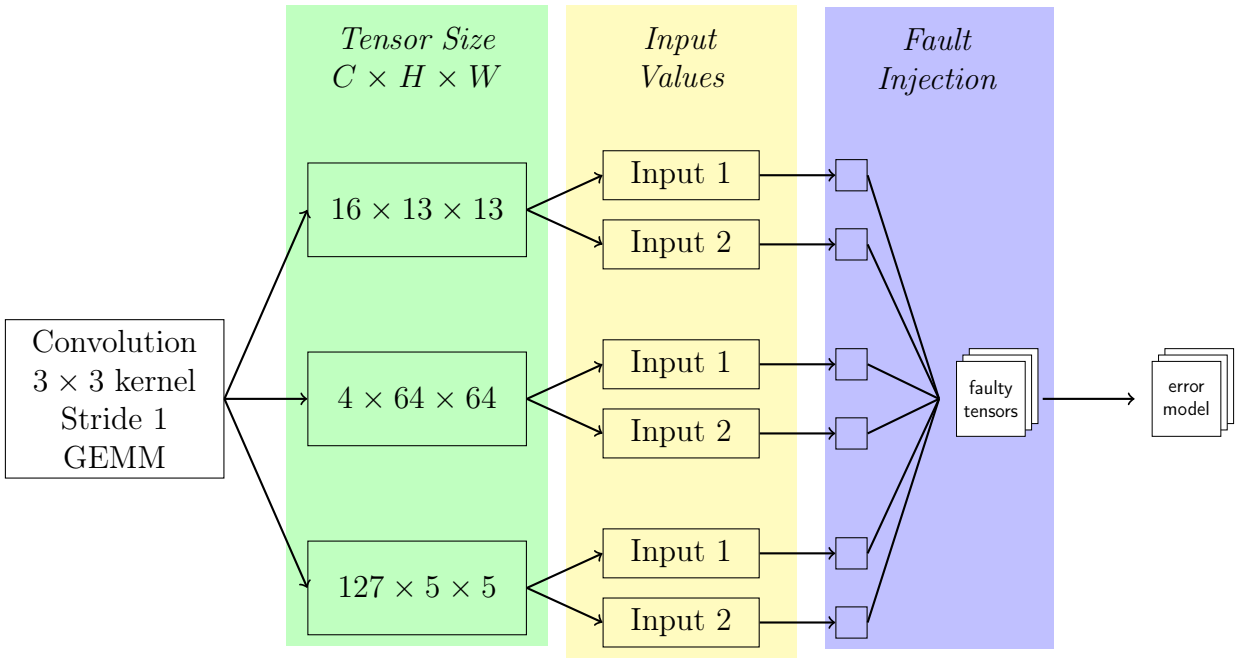


Figure 3.2: An example of the free variable choice for an experimental campaign

of input operands by extracting them from a real execution of the CNN or by generating them randomly. Note that the order of the two variables cannot be changed since the input tensor shapes depend on the output shape chosen before. This design of the experiment leads to 6 batches of the experiment. Each batch is injected multiple times and leads to multiple corrupted tensors. The tensors of all batches are combined together and are used for creating an error model.

3.1.4. Test Program

As explained, in CLASSES, the injections are made on each operator separated by a test program that must be compatible with the requirements of the fault injector. In this section, we want to better rationalize w.r.t. to the original methodology the role and the structure of the test program.

Once all the details of the campaign are refined, the user needs to write a test program that executes a single target operator on the device under test. Before starting the campaign, a script runs the test program once for each batch in the campaign, collecting one golden tensor for each batch. During the injection, the injector runs the test program with the same inputs used during the golden profiling phase. At the end of the injection, the test program will compare its output with the golden one. If the two differ, an SDC happens, and the program saves the corrupted tensor in a folder. Algorithm 3.1 contains an abstract

Algorithm 3.1 Abstract Test Program

Inputs

InputOperands: List of tensors containing the operands

Hyperparameters: List of configuration parameters for the operator

Operator: A function that runs the target operator of the campaign in the device (GPU). It takes in input a list of tensor operands and hyperparameters stored in the device memory and returns the output tensor, also stored in the device memory.

GoldenOutput: The golden tensor if the program is running in injection mode or **nil** if the program is running in normal mode

Outputs

ProgramOutput: The output of the program

SDCHappened: A boolean, **true** if a Silent Data Corruption happened

```
1: Assert that the device is available
2: Execute Device preliminary configuration
3: Allocate host memory for OperatorOutput
4: Allocate device memory for InputOperands, Hyperparamters and OperatorOutput
5: Transfer InputOperands and Hyperparamters in the device memory
6: Execute in the device ProgramOutput ← Operator(InputOperands, Hyperparamters)
7: Wait the termination of the Operator function in the device
8: Transfer from the device to the host OperatorOutput
9: if GoldenOutput is not nil then
10:    SDCHappened ← ProgramOutput ≠ GoldenOutput
11:    if SDCHappened then
12:       Save ProgramOutput to file in corrupted tensors folder
13:    end if
14: else
15:    SDCHappened ← false
16:    Save ProgramOutput to file as golden tensor
17: end if
18: Deallocate host and device memory
```

description of a unified test program that can generate a golden tensor when executed generally in the device. Instead, when it receives the golden tensor as an argument, it compares its output with the golden one if they are different. It saves the corrupted tensor in a folder.

3.1.5. Considerations on the injection campaign size

Once the user has defined the structure of every campaign, they need to establish how many experiments to perform in order to have enough data to build a statistically significant error model. Just in this section, we consider a simplified version of an error model

that contains C categories describing different corruption patterns of the faulty tensors. Each tensor belongs to one of the C categories. In Section 3.2, we will explain in detail the actual structure of the error models.

Each category in the model is associated with a value p_i, indicating the proportion of corrupted tensors belonging to the category i. The exact value of p_i can be calculated by performing an exhaustive fault injection campaign, namely a campaign where the fault injector emulates all the faults in all locations that can access within the campaign definition. Exhaustive campaigns require a huge number of architectural injections. In fact a soft error can happen at each unprotected location in the architecture at every instant of the execution, leading to an explosion in the number of possible faults. To make the injection phase feasible, we can randomly and uniformly select a subset of all possible faults to sample the population. Most fault injection frameworks natively perform the sampling, offering the possibility of configuring the sample size. The proportions p_i obtained using the results of experiments performed on a sample of the population of size N are not as exact as the one obtained from the exhaustive campaign and will have some statistical uncertainty. The user may want to use only the models that have an uncertainty under a certain threshold.

Some works, such as [32], use statistical inference to estimate the minimum sample size for injection experiments in order to calculate the reliability of a CNN network with a determined uncertainty. In particular, [46] proposes a method for determining the sample size needed to estimate the proportion of critical faults s over all possible stuck-at faults in the weights of a CNN with a precision h and a confidence level α. More in detail, the authors find a formula that calculates the minimum sample size $\bar{N}$ for which it holds

$$P(\bar{s} \in (s-h, s+h)) > 1-\alpha \tag{3.1}$$

where $\bar{s}$ is the estimated proportion obtained from the sample and $(s-h, s+h)$ is the confidence interval of level α. The proposed method applies to experiments that have two outcomes, such as "critical" and "non-critical" fault, but it is possible to extend this method to our error model, which has C possible outcomes, by using the multinomial proportion estimation methods proposed in [22] and [52]. However, in order to be able to apply statistical inference for estimating Bernoulli or multinomial proportions, some assumptions should be made

1. The trials must have two (or in general C) outcomes
2. The trials must be independent of the others

3. The number of trials must be fixed
4. The probability of success must be the same for each trial

While the first three assumptions are reasonable in the discussed setting, the fourth assumption is too strong for a fault injection scenario [46]. Without that assumption, we cannot apply statistical inference methodologies based on Bernoulli or multinomial proportions. Without having the possibility of using a statistical inference method, we have to assume that no quantity of experiments can guarantee any precision with any confidence in all the category's proportions. As a consequence, the number of injections to be performed in each campaign should be as high as possible, considering the computational resources that the user has at his disposal and the execution time of each injection.

3.2. Improving the Error Models

The original CLASSES error models present some flaws in their structure. In the first two sub-sections, we will expose some issues in the structure of the spatial models of CLASSES and subsequently propose some improvements to them. Then we will do the same for the domain models, with a section that exposes the flaws and a section that proposes the enhancements.

3.2.1. Spatial Models Limitations

In this section, we will discuss the structural limitations of the error models proposed in the original version of CLASSES focusing on the spatial models. In particular, in the structure of the error models defined in CLASSES we identified two issues.

The first issue is caused by the use of the error cardinality as a way of classifying the tensors. As discussed in Section 2.7.2, the faulty tensors are classified based on their error cardinality, the spatial class and the related spatial parameters. Using cardinality as a structural element in the spatial models creates a strong dependency of the models on the size of the tensors used for the test, especially in the case where only a single kind of output tensor is injected. For example, consider a "Full Channels" spatial class that includes all the feature maps with at least a channel completely corrupted (100% of faulty values in a single channel). If we plan an injection campaign only in instances of the operators that yield an output tensor of shape $C \times H \times H$ (as done in CLASSES), where H is the smallest in all the network, the occurrences of a "Full Channels" pattern with a single channel corrupted, will all have in the model a cardinality of H^2. Following the simulation procedure described in Section 2.7.3 we would generate "Full Channels"

with only a cardinality of H^2. The problem arises when we want to simulate a "Full Channels" in a tensor with square channels of size $2H$. In that case, following the error models collected we will inject error patterns of cardinality H^2, while the single-channel has an area of $4H^2$ missing the point of the "Full Channel" spatial class.

A second issue comes from the noisy behaviors of the spatial locations. We refer to these patterns as *Noisy Patterns*, which are essentially similar to the most frequent patterns in the models, with a few random erroneous values missing. This situation arises when a value is expected to be corrupted according to the expectation of the spatial class and the memory access patterns, but it turns out to be correct unexpectedly. Several factors can lead to noise in the pattern, such as newly generated values being close enough to the golden value, making them not classified as erroneous since their difference is bigger than ε defined in Section 2.7.1.

To illustrate this further, let's consider a "Same Row" spatial class containing all its errors in a single row of single channels, characterized by an error vector. Imagine that the most frequently encountered pattern consists of 32 consecutive errors in the same row, with no errors occurring elsewhere in the tensor. The error vector representing the linearized offsets relative to the first error would be $(0, 1, 2, ..., 31)$, During a fault that produces this described pattern, one of the 32 corrupted values may remain uncorrupted due to some noisy behavior. This means that for each different position that remains uncorrupted, we end up with a separate error vector, leading to potentially 32 additional entries in the spatial model, despite the pattern being almost identical except for a single masked value. If, in the example above, there are two noisy values, there could be up to 32×31 possible error vectors. This phenomenon is illustrated in Figure 3.4, where four very similar kinds of error lead to four different parameters. This phenomenon worsens if more than one value is affected by noise. The main takeaway from this example is that error vectors can be very sensible to noise phenomena caused by random masking. While the error vectors accurately represent noise phenomenons, we may not be interested in having so many details, such as the exact position where noise is located, leading to many entries in the spatial module that represent the same fault syndrome.

3.2.2. Improving the Spatial Models

To address the issues discussed in the previous section, we propose a new structure for the spatial models. The spatial models still remain a set of different groups of faulty tensors with an associated relative frequency used at simulation time for reproducing the proportion in which the various groups of tensors appeared during the fault injection.

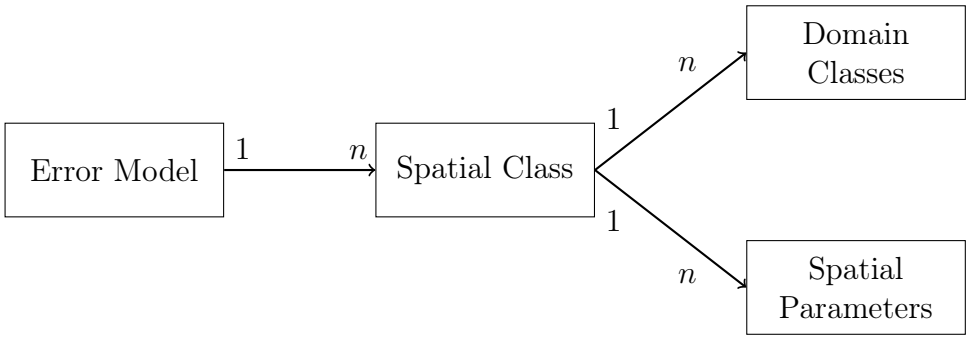


Figure 3.3: High-level structure of the proposed error model structure

The major change in the models stands in how the groups are constituted. In the spatial models of CLASSES, the tensors were grouped by error cardinality, spatial class and spatial parameters. In this revision of the models, we removed the cardinality as one of the characteristics of the groups, leaving only the spatial classes and their parameters. Figure 3.3 shows the high-level structure of the revisited error model structure. Notice how the spatial class stays at the top of the hierarchy having under them the related parameters. Also, the spatial models were moved under the spatial class, but this will be discussed further in Section 3.2.4.

Another improvement was made in the structure of the spatial parameters. Instead of forcing the parameters to be error vectors, we made it possible for the user who defines the error models to adopt a flexible key-value data structure. For each spatial class, the user defines the names of the parameters that wants to capture depending on the spatial class and then writes a parameter function that returns a key value map with the name of the parameter and the value calculated on the tensor. For simplicity, all the tensors of the same spatial class must always have the same keys. The user can define two types of parameters:

- *Key Parameters* are used for creating the groups of tensors of the spatial models. If two tensors are in the same spatial class and have the same values for the key parameters then they belong in the same tensor group.
- *Aggregated Parameters* have a key, a value and an aggreator function (e.g., max, min, sum, or avg). After the groups of tensors are created following the values "Key Parameters" the aggregated group value of this kind of parameters is computed by applying the aggregator function to the list of parameter values of all tensors within the group.

In this way, it is possible to define new types of parameters that are different from the error vectors. For example, it is possible to define integer or float parameters (like "percentage

of corrupted channels" or "maximum number of errors per channel"). In the case a key parameter varies a lot in all the tensors, like floating point values or percentages, to avoid too many classes, caused by the many values assumed for that parameter by all the tensors, we introduced the possibility of using ranges as the values of the key parameters, creating larger and more significative tensor groupings and avoiding to scatter the tensors in too many small groups. This possibility turns out to be particularly useful for percentages, creating tensor groups with values of key parameters such as "10 to 20% of corrupted channels".

If we take as an example the "Full Channels" spatial class, introduced in the previous section, one of the possible ways of defining the parameters is:

- **Key Parameters:**
  - *Percentage of corrupted channels*: The percentage of corrupted channels over the total number of channels, expressed as a range of multiples of 10%.
  - *Average corruption percentage*: The number of erroneous values divided by a channel's area ($H \times W$) and by the number of corrupted channels. This measure is also expressed as a range of percentages, with a step of 10%.
- **Aggreated Parameters:**
  - *Max Corrupted Channels*: The maximum number of corrupted channels in the group (Aggregatator function: max). This parameter can be used to limit at simulation time the number of corrupted channels.

Listing 3.1 shows an example of the structure of the spatial model. The top-level classification criterion is the spatial class and under it there are the spatial parameters. Each spatial class is described in the error models with the following fields:

- Relative frequency: The proportion of faulty tensors in that class over the number of faulty tensors obtained in the campaign
- Associated domain model (better discussed in Section 3.2.4), and a
- List of groups in the models, with each group described as follows:
  - Key Parameters: A dictionary with the names of the key parameters associated with their value
  - Aggregate Parameters: A dictionary with the names of the aggregate parameters associated with their value

Listing 3.1: Structure of new spatial model

```
{
    "full_channels": {
        "frequency": 0.23,
        "domain_model": ...,
        "parameters": [
            {
                "keys": {
                    "corrupted_chan_pct": (10%, 20%),
                    "avg_corruption_pct": (40%, 50%)
                },
                "aggregate": {
                    "max_corr_channels": 6
                },
                "frequency": 0.2,
            },
            {
                "keys": {
                    "corrupted_chan_pct": (50%, 60%),
                    "avg_corruption_pct": (40%, 50%)
                },
                "aggregate": {
                    "max_corr_channels": 8
                },
                "frequency": 0.8
            }
        ]
    }
}
```

- Frequency: The proportion of faulty tensors in the group over the number of faulty tensors in the spatial class

We will spend now the last part of this section discussing how the changes proposed here address the limitations discussed in the previous section. Regarding the issues created by the presence of the cardinality in the model, removing it from the models solved the issue. However, there is a need for some other parameters that determine how many errors are generated by the simulator. With the new structure of the spatial models, this aspect is left to the user, who has the flexibility of defining the parameters that need to be used at generation time to decide the cardinality of the tensor with the goal of replicating with good fidelity the patterns. The user for example can use a percentage of corruption in the parameters, if they want to make the cardinality scale with the size of the tensor, or it

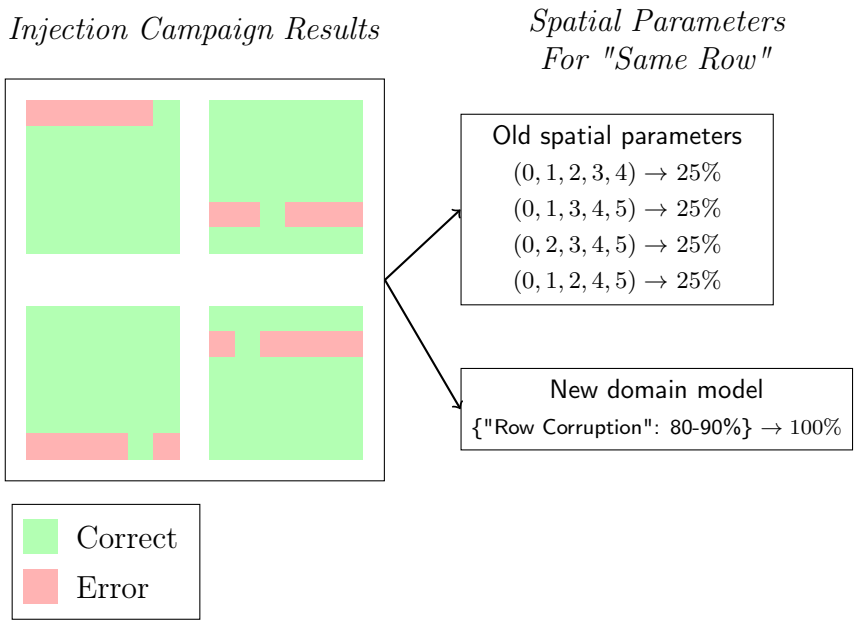


Figure 3.4: Example illustrating how the new spatial parameters handle noisy patterns

can put a fixed value if the size of the error is constant, and then they enforce this choices by implementing the proper error generator for the spatial class.

The problem of the noisy patterns is solved also with the new definition of the parameters, because instead of using the error vectors the user may choose to use percentage ranges in the values of the key parameters large enough to absorb the error, reducing the number of groups entries created in the error models. Figure 3.4 shows an example of how the new parameters can mitigate the problem, using the "Same Row" pattern introduced before as an example. Notice how a single noise element creates four different error vectors and consequentially four error groups, while in the new parameters definition constituted by a single ranged key parameter called "Row Corruption Percentage", the noisy patterns fall in a single group. This modeling of the "Same Row" pattern gives also the benefit of easily reproducing the noise phenomenon during simulation, automatically scaling it with the size of the row, in bigger tensors.

3.2.3. Limitation of CLASSES Domain Models

In this section, we will discuss the limitations of the error models on the domain side. We will address these limitations in the following section. The first limitation of the domain models stands in the existing definition of the value classes, specified in Section 2.7.2. The value classes $[-1; 1]$ and Uncategorized both include valid floating point numbers representing erroneous values that are respectively a minor error that is very close to the

original one and a bigger error distant more than 1 from the golden value. However, having a fixed distance threshold to determine the severity of the error simplifies too much the problem. Depending on the network and on the position of the layer, the golden tensor may have a distribution where all the values are very close together and span in a range way smaller than 1, then an error of 1 results in being very disruptive. On the opposite side the tensor may have a very broad distribution and a difference of 1 may not be so relevant relative to the values of that tensor. So we think that the value classes must be revisited to take into consideration the actual distribution of values in the golden tensor.

The second limitation of the error models in CLASSES stands in how the domain models are built. The occurrences of each value class are counted from all the corrupted tensors in the campaign, and in the error models the five value classes are stored with their relative frequency. If we calculate the domain models this way, we lose track of how the domain models were distributed inside each tensor. For example, consider the example shown in figure 3.5, where half of the tensors have almost all erroneous values, in the $[-1, 1]$ class and the other half has almost uncategorized corrupted values. Consider for now only the results of the simulations using the old (current) domain models. For how the domain model is structured now they would store the information that about half of the values were $[-1, 1]$, and the other half were uncategorized, without knowing the internal distribution of the value classes considering the tensors in isolation. Using the error generator described in Section 2.7.3, with the old model, we would obtain tensors with about half of their values in $[-1, 1]$ and the other half of the values uncategorized, even though we do not ever encounter this type of domain distribution in faulty tensors coming from the injections.

3.2.4. Improvements to the Domain Error Model

In this section, we address the issues raised in the previous sections about the CLASSES error models, redesigning the structure of the models to address them. The first change that we performed is to make the domain models dependent on the spatial class, having for each spatial class a separate domain model calculated only on the faulty tensors belonging to that spatial class. This change on one side increments the number of domain models stored, and makes the structure of the error models more complicated, but it allows us to better capture the behaviors of the faulty values of each spatial class which was not possible before because the domain model was one for the whole fault injection campaign.

Following the conclusions of previous works stating that errors way out of the range of values contained in the golden feature map are more likely to cause critical errors [10],

we decided to modify the definitions of the value classes eliminating the $[-1, +1]$ and Uncategorized value classes and replacing respectively with two new classes:

- *In-range* value class includes all the values that are different from the golden counterparts and fit into the range of values of the golden feature map F, calculated as $[min(F), max(F)]$. This value class replaces the old $[-1, +1]$ value class.
- *Out-of-range* value class includes all the values that are different from the golden counterparts and that are outside of the range $[min(F), max(F)]$. This value class replaces the old uncategorized value class.

The new domain model presents also changes in its structure. Instead of having only a list of value classes associated with their relative frequency calculated over all erroneous values in all faulty tensors obtained from the fault injection campaign, the new error model structure is a list of *Domain Classes*, with each a relative frequency. A Domain class is a characteristic assigned to an entire tensor, while a value class is assigned a single erroneous value. The domain class tells what is the distribution of the value classes of the erroneous values inside a tensor. The most simple domain classes are the ones that represent tensors that have only a single type of value class, such as only zeros or only in-range. There are also domain classes that include all the tensors that have two types of value classes. In this case, the description of the domain class, other than containing the two value classes present in the feature map, also contains information about the proportion in which the two value classes are present in the tensor. The proportions are expressed as intervals of a fixed length L, decided by the user. For example, if $L = 10\%$, a domain class can be "In-range $(10\% - 20\%)$ and Out-of-Range $(80\% - 90\%)$". If the tensor has more than two domain classes then it belongs to the catch-all "random" domain class that stores the distribution of the value class as done in the old domain models.

In Listing 3.2, we can see an example of the new domain models. 32% of the tensors has only In-Range values, while 18% of the tensors has between 40% to 50% of their erroneous values in the In-Range value class and the remaining in the Out-of-Range value class. Finally, 9% of the tensors have three or more value classes, having on average 14.3% NaNs, 27.7% In-Range values, and the remaining are Out-of-Rangevalues.

The changes made in the error models have the goal of addressing the limitations and improving the fidelity of the error simulation. The definition of the two new values classes, In-range and Out-of-Range replacing respectively $[-1, +1]$ and Uncategorized addressed the first issue listed in Section 3.2.3, by creating two value classes that are aware of the distribution of the tensor analyzed and that can be easily reproduced in the error simulation keeping in mind the distribution of the target tensor.

Listing 3.2: Structure of new domain model

```
[
    {
        "in_range": 100%,
        "frequency": 0.32,
    },
    {
        "in_range": (40%, 50%),
        "out_of_range": (50%, 60%),
        "frequency": 0.18,
    },
    {
        "in_range": (10%, 20%),
        "out_of_range": (80%, 90%),
        "frequency": 0.11,
    },
    {
        "NaN": (0%, 10%),
        "zero": (90%, 100%),
        "frequency": 0.30,
    },
    {
        "random": {
            "NaN": 14.3%,
            "in_range": 27.7%,
            "out_of_range": 58.0%
        },
        "frequency": 0.09,
    }
]
```

To tackle the second limitation reported in the previous section we introduced the concept of Domain class, which summarizes how the values are distributed in the tensor. The example in Figure 3.5 shows how the new domain class concept can help to better reproduce the domain patterns in the application-level error simulation. Consider an ideal fault injection campaign where half of the tensors have almost all in-range erroneous values, and the other half has almost all out-of-range corrupted values. Using the old legacy model, we would have only the information that about half of the values were in range, and the other half were out of range, independently from the other values in the same tensor. During the application level error simulation with the old model, the great majority of the generated tensors will have about half of their values in the range and

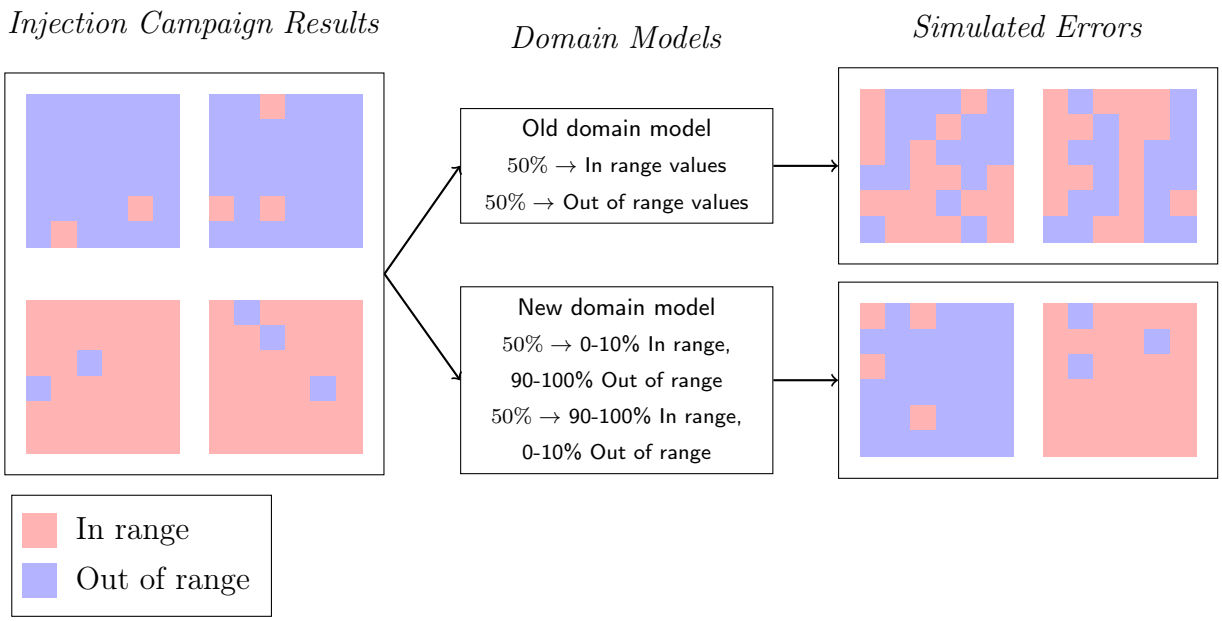


Figure 3.5: Example of simulation using the old and the new domain models

the remaining values out-of-range, even though we do not encounter this type of domain pattern in the tensors coming from the injection campaign. The new models can better capture this behavior since they take the distributions of value classes inside each tensor instead of evaluating the distribution globally.

3.3. Automation of the error modeling process

As discussed in Section 2.7.2, the error modeling in CLASSES requires the manual intervention of the user in the definition of the error models. The original work of CLASSES proposed a simple algorithm (Algorithm 2.2) to aid the user in performing the classification automatically requiring to the user to specify using code the functions that define a spatial class. The implementation of that algorithm is not provided in the original work but it is easy to implement. However, even if the user has the implementation of the classification available, they still need to implement various tools that help them to visualize the corruption in the tensor, and get insightful statistics about them. Once the spatial classification is completed the user also needs a tool that generates the error models in a compatible format with the simulator. One of the innovative contributions of this work is the definition and the implementation of a unified tool that performs the visualization and the automatic classification of the corrupted tensors, with the possibility of providing insights and statistics that help the user with the error modeling. The tool also, once

the classification work is over, should be able to generate automatically the error models, ready to be read from the simulator, without any further modification. We will present the tool in question by first defining the goals and requirements in Section 3.3.1 and then by describing an implementation strategy that respects those goals, in Section 3.3.2.

3.3.1. Design goals and Requirements

The *Analyzer* program has the goal of helping the user to define the spatial classes used in the error models by providing statistics on the corrupted tensors and visual aids. After the models are entirely defined, its goal is to create the error models automatically file so they can be used directly from the application-level error simulator. The process of the model definition is iterative. The user will observe the statistics and the visual patterns of the errors, and from there, they will notice an emergent spatial class. The user then will define, by code, a spatial class by specifying its characteristics already described in Section 2.7.2. The user then will re-run the program and they will check the result of the classification. Suppose the newly introduced class matches the user expectation regarding which tensors are included in the class and which are not. In that case, the user continues the process until a few tensors are in the uncategorized class. This process should be done considering all the tensors from all the performed injection campaigns to have spatial models that have the same classes for all the operators.

From the goals described above, we derive a numbered list of requirements of the application:

- R1: The main task of this program is, given a list of batches of experiments, each one containing a golden tensor and one or more folders of corrupted tensors, to classify the tensors using the user-defined spatial class and to generate and save to file the error models described in Section 3.2, in a format that is compatible with the application level simulator.
- R2: Visualize, for each tensor, an image of the spatial distribution of the errors across the corrupted channels. Besides the location, the user should be able to see the value class of each error in the image. The images must be grouped by spatial class in different folders.
- R3: Save a synthesis of the results in a human-readable report containing actionable data that the user can easily interpret and re-use to improve the models in another iteration of the model design process.
- R4: Offer the option to the user to generate a structured database containing the

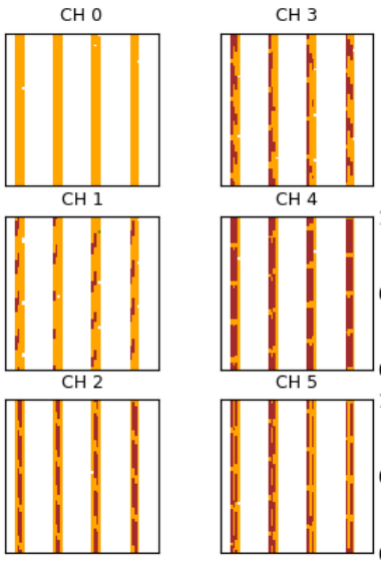


Figure 3.6: Example of visual output coming from the analyzer

analysis data for each tensor. In this way, the user can use a structured query language to derive custom statistics about the tensor.

- R5: Allow the user to add new spatial classes of error models, describing the indicator function and the parameter function with code and specifying for each class a priority, without having to change other modules of the application
- R6: The execution time of the tool should take a reasonable amount of time, even when there are large quantities of tensors to be analyzed

In the next section, we will describe a possible implementation compliant with the requirements presented above.

3.3.2. Implementation

The classification software is a command line application written in Python 3, a scripting language known by most people working in the domain of deep learning since it is the language used for mainstream machine learning frameworks such as PyTorch and TensorFlow.

- R1: The program takes in input the relative path of the golden tensor and the path of a folder containing all the batches to analyze. All the tensors are saved as a .npy file, a binary file format used by numpy for storing tensors. The program loads each batch's golden tensor and corrupted tensors, detects the position of the erroneous value and performs the spatial and value classification. When the classification is

complete, it generates a JSON file containing the error model as described in Section 3.2.2, with a structure compatible with the application-level simulator.

- R2: The program generates an image for each corrupted tensor that contains a number of 2D sub-plots equal to the number of corrupted channels in the tensor. Each sub-plot is composed of $H \times W$ cells, colored if the corresponding location in the tensor is corrupted. The color of the cells varies on the value class. Figure 3.6 shows an example of a visual output of the analyzer.
- R3: The program, at the end of the analysis, generates a brief human-readable JSON report containing the number of tensors in each class, the cardinality, and various other information.
- R4: The program exports a SQLite 3 database in the output folder. SQLite3 is a simple relational Database Management System (DBMS) in a standalone file. The database contains a single table with various columns containing statistics about the analyzed tensors, such as the error cardinality, spatial class, and domain class. The user can easily open the file using a compatible client and perform SQL queries to gather data about the experiment.
- R5: To define a new spatial class, the user needs to create a file in a specific package, where they implement a function that is both the spatial class indicator function and the parameter function. The function takes in input the list of the coordinates of the erroneous values and returns `None` if the spatial does not belong to that spatial class and a dictionary of parameters otherwise. Then the user needs to add the function in a list of all spatial classes in the package init file, specifying a class name. The order of insertion in the list defines the priority since the spatial class functions are evaluated in order until one of the matches. The user does not need to modify any other part of the application to add a new spatial class.
- R6: To speed up the runtime of the analysis, we execute in parallel the processing of each batch of tensors since the classification of a tensor batch can be run independently from the others. For parallelization, we use the Python 3 multiprocessing library. The user specifies the number, p, of parallel processes from the command line. The program creates p worker processes and a work queue of batches and divides the batches between the workers.

In this chapter, we have presented a methodology for assessing the reliability of soft errors in system-level fault injection campaigns built upon the foundation laid out in CLASSES. We added to the existing methodology several innovative enhancements aimed at signif-

icantly improving the error modeling process. We proposed a rationalized process for planning platform-level fault injection campaigns, making it more formal and adaptable to diverse scenarios. Additionally, the restructuring of error models addresses the limitations of the previous CLASSES version, enhancing the fidelity of generated errors in application-level simulations. Moreover, the introduction of a unified tool to automate the error modeling phase represents a major advancement, reducing the manual workload on users and accelerating the overall process. These improvements make CLASSES more flexible and re-adaptable to new challenges. In the next two chapters, we are going to apply this methodology to a real scenario, showing in practice the improvement introduced here.

4 | Experimental Results: Error Models Definition

In this chapter, we will apply the methodology defined in Chapter 3 to build the experiment campaigns and to extract the error models that we will use in Chapter 5 to perform an application-level analysis of the reliability of CNNs running on the GPU. First, we introduce the experimental environment and the instance of the problem for which we want to generate the model, and then we will apply step-by-step the procedures defined in the methodology to define the experimental campaigns, perform the injection experiments and obtain the error models.

4.1. Design of the Fault Injection Campaign

This section describes how we applied the method proposed in Section 3.1 given the goals of the analysis. Starting from the experimental environment at our disposal and keeping in mind the goal of the model we design the campaign by extracting the operators from the target CNNs and by specifying the details of each campaign associated with an operator.

4.1.1. Experimental Environment

The machine used for running fault injections is equipped with an NVIDIA RTX3060, with an Ampere architecture and a *Compute Capability* (CC) of 8.6. The installed GPU is used as the target device for the architectural fault injection. The OS installed on the computer is Ubuntu Linux 20.04 LTS. The architectural fault injector that we will use in the experimental campaigns is NVBitFI [53], already described in Section 2.6.2. NVBitFI has some system requirements inherited from the instrumentation tool from which is derived, NVBit, which is compatible with architectures between Kepler (CC 3.5) and Ampere (CC 8.6) and requires 11.0 as the minimum version of CUDA. NVBitFI is the successor of SASSIFI [24], already used in the original CLASSES work [8].

To simulate the effects of faults in the control units of the GPU we applied to NVBitFI

the patch proposed in [18] that includes the support for warp errors in NVBitFI. A warp random error corrupts all 32 threads of a warp, replacing the *Instruction Output Value* (IOV) of each thread with a bitwise generated random value. This error simulates the effect of a fault that makes the warp scheduler skip the execution of the warp, leaving in the destination registers random invalid values remaining from previous instructions. NVBitFI can inject faults in the following instruction groups:

- `fp32`: All the floating point instructions using 32-bit operands
- `ld`: Load from memory instructions
- `gp`: General purpose register instructions

In its original version, NVBitFI splits the total number of injections to perform in equal parts between the instruction groups. We modified the injector scripts to make NVBitFI split the injections between instruction groups proportionally to the number of executed instructions within each group. For implementing this change we added a preliminary step in the injection script that parses the profiler output files and calculates the number of injections of each group.

4.1.2. Fault Models

Between the numerous fault models offered by NVBitFI, we use the following models:

- *Warp Random* [18] fault model replaces the instruction output value of all the threads of the warp with a random value generated bit by bit. This fault model represents the effect of a soft error that caused a bitflip in the scheduler of the GPU, and consequentially caused the scheduler to skip or misplace the execution of an instruction of a warp [14], leaving random values in the destination registers of the warp.
- The *Single Random* fault model replaces the instruction output value of a single thread with a random value representing the effect of a soft error happening on an intermediate pipeline register of the ALU of a functional unit.

Differently to the *Single bit-flip* , another fault model available in NVBitFI, which flips a single bit in a single thread, the two fault models above do not represent directly the effect of a soft error but they are approximated second-hand fault models caused by a soft error in a part of the architecture that is not observable and controllable by the programmer and so it cannot be affected directly from an instrumentation tool like NVBit. Unfortunately, comparing the original CLASSES error models with new ones is unfeasible since the error

models are different and the previous CLASSES experimental setup employed a Kepler GPU [8] using SASSIFI as the injector, very different from the current setup described in Section 4.1.1. So we decided to not use single bit-flip fault models since it would require an additional campaign for each operator, and we would not be able to compare the two results. Also, the analysis of the effects of single-bit faults was already done in the previous version of CLASSES, albeit it was done using the previous structure of the error models.

4.1.3. Operator Extraction

In this section, we perform the operator extraction phase described in Section 3.1.2. We chose to inject in the most common operators present in the CNN, keeping in mind that we are going to use some of these results for application-level simulation in YOLOv3, introduced in Chapter 5, and that we can inject only on the operators for which there is an implementation in CuDNN. Counting these considerations we decided to perform injection campaigns in the following operators:

1. Convolution (GEMM Strategy)
2. Convolution (FFT Strategy)
3. Convolution (Winograd Strategy)
4. Batch Normalization
5. Max Pooling
6. Average Pooling
7. Sigmoid
8. ReLU
9. Bias Add

For each operator, we perform two campaigns: one injecting a warp random error and another one injecting single random errors. In this way, using the error simulator, we can compare the resilience of each operator to the two fault models. We decided, for the convolution, to study the three different strategies discussed in Section 2.1.4.

Table 4.1: Fixed parameters of each experimental campaign performed.

Campaing Id	Operator	Fault Model	Hyperparameters
conv_gemm_wr	Convolution (GEMM)	Warp Random	Kernel: 3×3 Stride: 1 Dilation: 1 Zero Padding: 1
conv_gemm_sr		Single Random	
conv_fft_wr	Convolution (FFT)	Warp Random	
conv_fft_sr		Single Random	
conv_win_wr	Convolution (Winograd)	Warp Random	
conv_win_sr		Single Random	
batchnorm_wr	Batch Normalization	Warp Random	-
batchnorm_sr		Single Random	
maxpool_wr	Max Pooling	Warp Random	Kernel: 2×2
maxpool_sr		Single Random	
avgpool_wr	Average Pooling	Warp Random	Kernel: 2×2
avgpool_sr		Single Random	
relu_wr	ReLU	Warp Random	-
relu_sr		Single Random	
sigmoid_wr	Sigmoid	Warp Random	-
sigmoid_sr		Single Random	
biasadd_wr	Bias Add	Warp Random	-
biasadd_sr		Single Random	

4.1.4. Campaign Design

After extracting all the operators we now need to define the fixed and the free variables of each campaign, as described in the proposed methodology. We first start defining the fixed parameters.

Table 4.1 shows the fixed parameters chosen for each campaign. For some operators, the table lists the hyperparameters that remain fixed during the whole campaign. For example for the convolution and the pooling we keep fixed the kernel size to values that are commonly used in the CNNs that we are going to use. Other operators, such as the activation, due to their simplicity do not have any parameter to be fixed.

After picking the values of the fixed variables, we decided which are the free variables and what values they assume. For all the campaigns we decided to structure the free variables as follows:

- *Tensor Output Shape*: The shape of the output feature map. Figure 4.1 shows the 16 different shapes we chose in a plot, where each label in the plot contains in order the number of channels and the size of each square channel. The sizes were chosen following the dimensionality reduction principle, discussed in Section 2.1.2. As shown in the plot we picked shapes with a large number of small channels, shapes

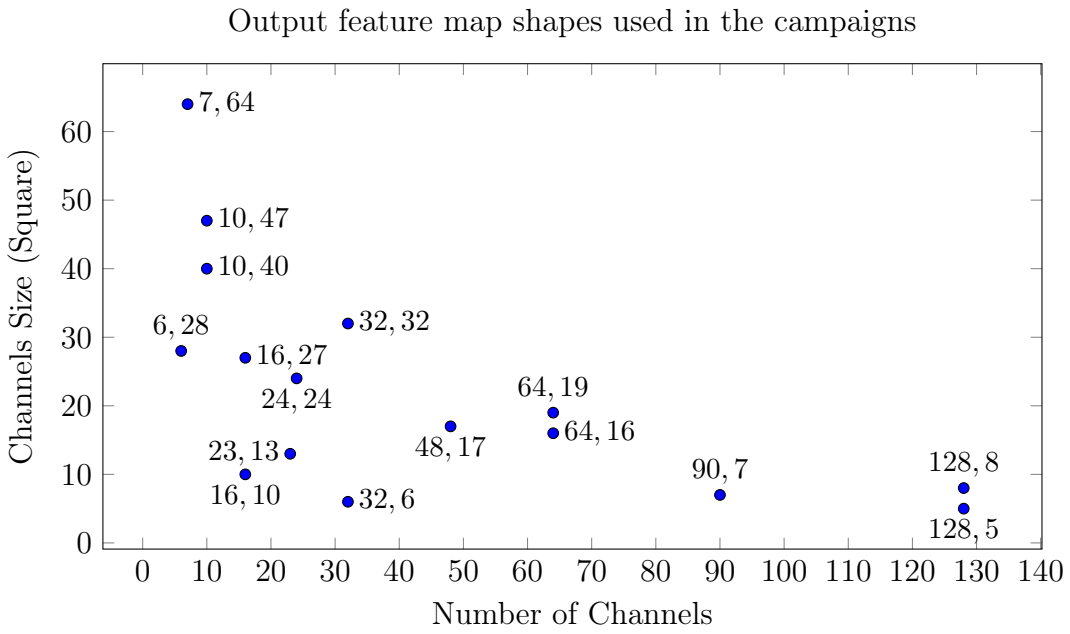


Figure 4.1: Plots of all the shapes of the feature maps used in all the campaigns.

with a lot of big channels and shapes in the middle of the two extremes. Some of these shapes were extracted from YOLOv3 and LeNet5, introduced in Section 2.3, while others were chosen empirically by us.

- *Input Values*: For each tensor shape we generate at least two different input values to avoid data dependencies from a single output.
- *Instruction Groups*: We inject in all instruction groups available in NVBitFI: `fp32`, `ld`, `gp`. Following the modifications we did to NVBitFI, we distributed the number of faults proportionally to the number of instructions profiled in each of the three groups.

The injections of each campaign are equally divided into batches. For each feature map output shape listed in Figure 4.1 we create at least two batches, each one having different input values. The number of experiments of the batch is equal to the number of injections for the whole campaign divided by the number of batches. The injections of each batch are divided between the three instruction groups proportionally to the number of instructions profiled for each group.

4.1.5. Campaign Sizing

Section 3.1.5 concludes that no choice of campaign size can give us statistical guarantees on the statistical significance of the relative frequency values contained in the error models.

Table 4.2: Fixed parameters of each experime ntal campaign performed

Campaing Id	Campaign Size	Corrupted Tensors	SDC Rate
conv_gemm_wr	20040	11716	58.4 %
conv_gemm_sr	16020	3817	23.8 %
conv_fft_wr	16052	8438	52.2 %
conv_fft_sr	16020	7471	46.7%
conv_win_wr	16020	5869	36.6%
conv_win_sr	16020	4643	29.0%
batchnorm_wr	10336	4216	40.8%
batchnorm_sr	10080	4119	40.8%
maxpool_wr	10294	4140	40.2%
maxpool_sr	10028	2855	28.4%
avgpool_wr	10182	4576	44.9%
avgpool_sr	10026	4074	40.6%
relu_wr	37862	2697	7.1%
relu_sr	10062	446	4.4%
sigmoid_wr	10292	1864	18.1%
sigmoid_sr	10080	1678	16.6%
biasadd_wr	19126	3621	18.9%
biasadd_sr	10034	1786	17.8%

So we opted to choose a minimum number of experiments based on the resources that we have at our disposal. We run some preliminary experiments for each operator to get a grasp on how long it takes to perform an injection. We observed empirically that each injection takes around 5 seconds for the convolutions (independently from strategy) and around 3.5 seconds for all the other operators. We noticed that these times are not significantly influenced by the size of the operands that we use, probably because the NVBitFI uses a big part of the time for setting up the instrumentation and for transferring data to the device. Sometimes, experiments can run for up to 20 seconds before getting interrupted by NVBitFI for a time-out, these experiments are classified as Detected Unrecoverable Error and do not yield a corrupted tensor. We observed that this behavior does not penalize too much the execution time of the campaign because it happens very rarely, so we decided its effect in estimating the execution times of the campaigns.

From these observations, we decided to set a minimum of at least 10000 experiments for each campaign, with an estimated runtime using the prposed size of about 14 hours for convolutions and around 10 hours for the other operations. Table 4.2 lists the count of injections and the number of corrupted tensors obtained from those experiments in each campaign. It also includes the SDC Rate for each campaign, namely the percentage of

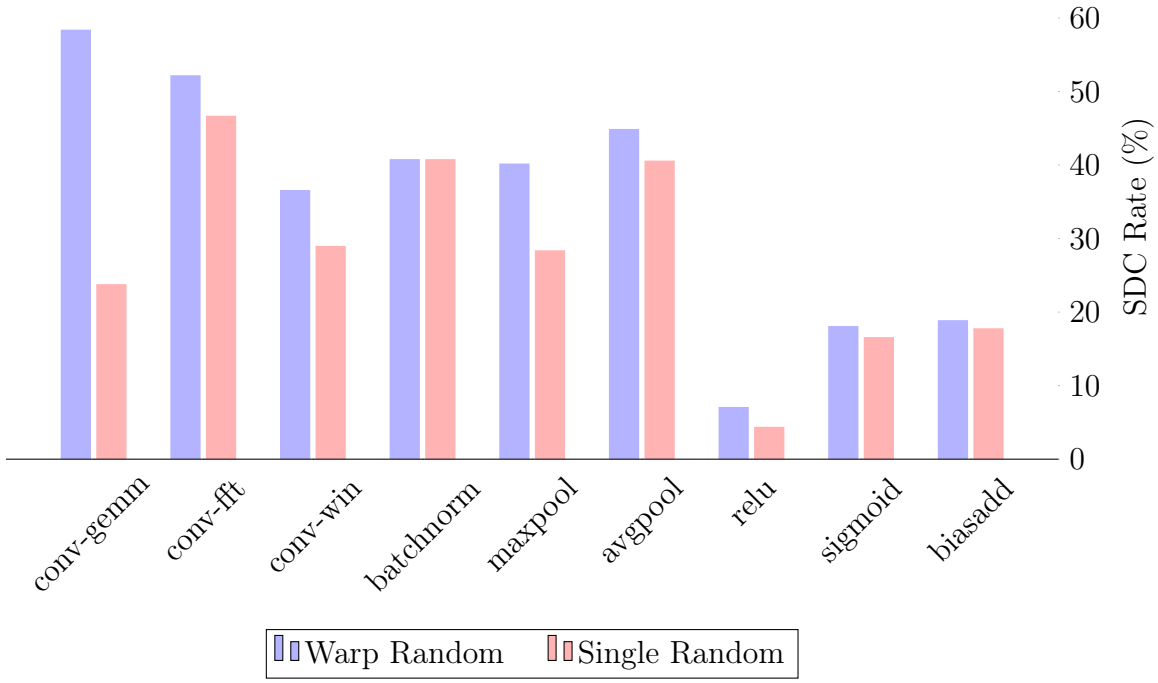


Figure 4.2: SDC Rate for each injection campaign

SDC (and so corrupted tensors) obtained from all the experiments.

Figure 4.2 plots the SDC rate for all the campaigns. We observe that the SDC rate is slightly higher for warp-random campaigns with respect to single-random ones. Warp random errors inject errors in 32 threads of a warp at a time, while single random corrupts only a single thread. Corrupting more threads reduces the possibility of masking the fault leading to a higher SDC rate. We observe also that operators using the shared memory such as convolution and batch normalizations have higher SDC rates compared to simpler operators like the bias add and the activations. This effect is caused by the use of the shared memory, which increments the fan out of an incorrect value, spreading the error in multiple locations, and consequentially making it more difficult to mask the error before the execution of the program terminates.

4.2. Error Models

Once we gathered at least some partial results from the various operators we started performing the error modeling procedure with the aid of the tensor analyzer script described in Section 3.3. From the analysis and the classification of the observed patterns using the classifier application, we found 12 new classes on top of the Uncategorized default class. Before listing all the classes with their parameters we need to introduce some concepts

used widely in the definitions of the classes and their parameters.

We define two contiguous values in a channel as two locations that are contiguous in the linearized version of the channel following the row-major order.The parameters of the spatial classes we defined can be of two data types: *Integer parameters* store simply an integer value. *Interval parameters* use two float numbers to represent the two ends of an interval. They are used to group together similar parameter values, in order to reduce the number of parameter groups (see Section 3.2.2). In the classes, we used interval parameters to represent percentages, so we decided to adopt ten, equally sized ranges of percentages $(0-10, 10-20, ..., 90-100\%)$ as possible values for the parameters.

Here we describe all the spatial classes by defining their indicator function with a natural language description. Each class also comes with a list of its key and aggregate parameters.

1. **Single**: A single erroneous value in the whole tensor
   - Channel Count: 1
   - Parameters: None
2. **Skip 4**: All erroneous values have a linearized distance multiple of 4 from the first erroneous location in the channel. Corresponding erroneous locations may be present in other channels
   - Channel Count: 1
   - Key Parameters:
     - *Unique Channel Indexes* (Integer): Counts the number of unique faulty channel coordinates
     - *Affected Channels %* (Interval): See Table 4.3
   - Aggregate Parameters:
     - *Min/Max Channel Skip*, *Max Corrupted Channels*: See Table 4.3
3. **Single Block**: All the erroneous values are contained in a block of 8, 16, 32 or 64 contiguous values, that may spill over the next channels. More than 50% of that block must be corrupted.
   - Channel Count: ≥ 1
   - Key Parameters:

  - *Block Corruption* % (Interval): Percentage of the block containing erroneous values.

- Aggregate Parameters: None

4. **Single Channel Alternated Blocks** A Single Channel containing repeated pattern composed of a block of 32 contiguous erroneous values alternated by one or more blocks of 32 correct values

   - Channel Count: 1
   - Key Parameters:
     - *Min/Max Block Skip* : Minimum and Maximum number of empty block between two corrupted blocks
   - Aggregate Parameters:
     - *Max Channel Size*: Maximum channel size for which the related key parameters are encountered

5. **Multi Channel Block**: One channel (the leader) has all his erroneous values contained in a block of 8, 16, 32 or 64 contiguous values. More than 50% of that block must be corrupted. Other channels may have any number of erroneous values, but they should be located inside the corresponding block of the leader channel.

   - Channel Count: > 1
   - Key Parameters:
     - *Block Size* (Integer)
     - *Average Block Corruption %* (Interval): Calculated as *Error Cardinality*/(*Block Size* $\times$ *Corrupted Channels*).
     - Affected Channels % (Interval): See Table 4.3
   - Aggregate Parameters:
     - *Min/Max Channel Skip*, *Max Corrupted Channels*: See Table 4.3

6. **Bullet Wake**: Each corrupted channel has a single erroneous value. The erroneous values are in the same position in all channels.

   - Channel Count: > 1
   - Key Parameters:

  - *Affected Channels %* (Interval): See Table 4.3
- Aggregate Parameters:
  - *Min/Max Channel Skip*, *Max Corrupted Channels*: See Table 4.3

7. **Same Row**: A single row of a single channel contains all the erroneous values
   - Channel Count: 1
   - Key Parameters:
     - *Row Corruption %* (Interval): Percentage of erroneous values over the row length
   - Aggregate Parameters:
     - *Min/Max Value Skip* (Integer): Minimum and Maximum number of correct values between two erroneous values
     - *Max Cardinality* (Integer): Maximum number of corrupted values in the parameter group
8. **Full Channels**: Each corrupted channel has more than 50% of erroneous values
   - Channel Count: ≥ 1
   - Key Parameters:
     - *Average Channel Corruption %* (Interval): Calculated as *Error Cardinality*/($H \times W \times$ *Corrupted Channels*).
     - Affected Channels % (Interval): See Table 4.3
   - Aggregate Parameters:
     - *Min/Max Channel Skip*, *Max Corrupted Channels*: See Table 4.3
9. **Rectangles**: The corrupted values in a channel form the shape of a rectangle. The same rectangle may be repeated on other channels in the corresponding location.
   - Channel Count: ≥ 1
   - Key Parameters:
     - Rectangle Height (Integer)
     - Rectangle Width (Integer)
   - Aggregate Parameters:

  - *Min/Max Channel Skip*, *Max Corrupted Channels*: See Table 4.3

10. **Shattered Channel**: All corrupted channels have at least a common corrupted location.
    - Channel Count: > 1
    - Key Parameters:
      - *Average Span Corrupted %* (Interval): The percentage of the corruption span constituted by erroneous values. The corruption span includes all the indices of all corrupted channels in between the minimum and the maximum corrupted index in all channels.
    - Aggregate Parameters:
      - *Min/Max Channel Skip*, *Max Corrupted Channels*: See Table 4.3
      - *Min/Max Span Width*: The minimum and the maximum span of the width between all the tensors of the parameter group
11. **Quasi Shattered Channel**: Each pair of corrupted channels have at least a common corrupted location.
    - Channel Count: > 1
    - Key Parameters: As in Shattered Channel
12. **Single Channel Random**: A single channel is corrupted, with no particular pattern emerging.
    - Channel Count: 1
    - Key Parameters:
      - *Channel Corruption %* (Interval): The percentage of the channel span constituted by erroneous values.
    - Aggregate Parameters:
      - *Min/Max Value Skip* (Integer): Minimum and Maximum number of correct values between two erroneous values in the parameter group.
      - *Max Cardinality*: The maximum error cardinality in the parameter group.
13. **Uncategorized**: Default spatial class. No condition is required to be in this class.
    - Channel Count: ≥ 1

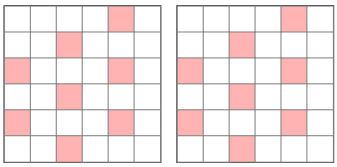
Table 4.3: Description of the parameters used by multiple classes

Name	Data Type	Description
Min Channel Skip	Integer	Minimum offset between two non-corrupted channels in the parameter group.
Max Channel Skip	Integer	Maximum offset between two non-corrupted channels in the parameter group.
Max Corrupted Channels	Integer	Maximum number of corrupted channels in the parameter group
Affected Channels %	Interval	Percentage of corrupted channels over the total number of channels in the feature map

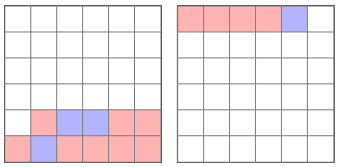
- Key Parameters:
  - *Channel Corruption %* (Interval): The percentage of the channel span constituted by erroneous values.
  - *Affected Channels %* (Interval): See Table 4.3
- Aggregate Parameters:
  - *Min/Max Errors Per Channel* (Integer): The minimum and the maximum erroneous values present in a channel
  - *Min/Max Channel Skip*, *Max Corrupted Channels*: See Table 4.3

Figure 4.3 shows a visual example of every spatial class defined above. All the erroneous locations are marked in red. For the block errors (Figures 4.3b and 4.3d) we marked in blue the locations that are part of the block but not corrupted and for the *Shattered Channel* (Figure 4.3i) block we marked in gray a common erroneous location in all the corrupted channels.

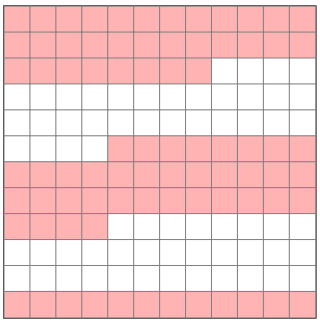
After having the definitions of the spatial classes and their parameters configured into the analyzer script, we re-run the script on all the campaign folders containing the corrupted tensors to get the final error models, one per campaign. The models are saved in a JSON file with a structure similar to the one shown in Listings 3.1 and 3.2.



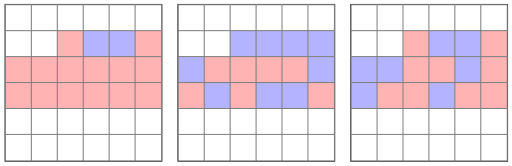
(a) Skip 4



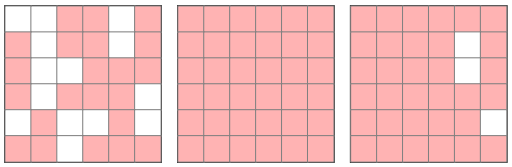
(b) Single Block (over two contiguous channels)



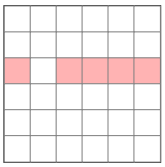
(c) Single Channel Alternated Blocks



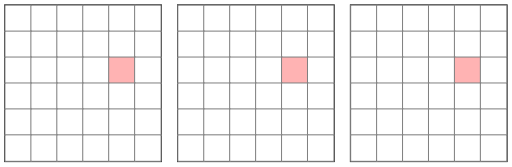
(d) Multi Channel Multi Blocks



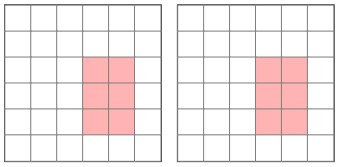
(e) Full Channels



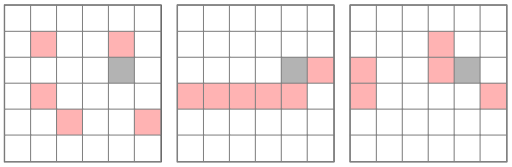
(f) Same Row



(g) Bullet Wake



(h) Rectangles



(i) Shattered Channel

Figure 4.3: Illustrations of the spatial classes

4.3. Outcome of the classification

Table 4.4 and 4.5 show the spatial class distribution in the various experimental campaigns, showing the classes that appeared with a relative frequency of at least 1%. From the data gathered at this stage, we cannot yet make firm conclusions about the overall reliability of a network affected by the faults we injected, but we can observe the results and compare the various campaigns against themself to speculate the results of the

application level simulation.

For example, if we want to make a comparative analysis of the three different strategies of the convolution we can observe the data in the tables described above, for getting some preliminary insights. In the FFT convolution error models, we observe a spike in the incidence of the *Full Channels* spatial class in both warp and single random experiments. This may be caused by errors affecting the computation in the frequency domain, that spread over entire channels when the tensor is converted back in the time domain. We can speculate that such *Full Channels* errors lead to a high incidence of critical errors at the application level, given the fact that they corrupt a big part of one or more channels, according to their definition. On the contrary, the GEMM convolution presents a majority of *Single* errors in using the single warp experiments and *Skip 4* for the warp random experiments. Those spatial classes contain usually less invasive errors, having limited error cardinalities, and we expect that faults falling in these classes cause less critical errors than the *Full Channels* errors. From this observation, we expect that the FFT convolution may be more vulnerable to warp and single error than the GEMM convolution. The Winograd convolution instead presents a variegated distribution of spatial classes so it is more complicated to make predictions only from this data. In every case, we will be able to make stronger conclusions for this comparative analysis, and in all the other possible analyses that we will execute, after using the results coming from the application-level error simulations.

In this chapter, we planned and executed the fault injection experiments using the improvements to the methodology proposed in Section 3.1. Then using the erroneous tensors obtained from the fault injection campaigns we generated error models for the various CNN operators, following the new improved structure discussed in Section 3.2. Those error modules, are the main products of this chapter and were generated with the aid of the unified error modeling tool proposed in Section 3.3. In the next chapter, we are going to use the produced models in the application-level error simulations that we will then use to produce an analysis of the reliability of an object detection network.

Table 4.4: Spatial Classes distributions for the operators using the shared memory.

Spatial Class	`conv_gemm`		`conv_fft`		`conv_win`		`batchnorm`	
	sr	wr	sr	wr	sr	wr	sr	wr
Single	76%	-	1%	-	22%	-	78%	-
Skip 4	-	45%	3%	1%	17%	-	22%	-
Single Block	1%	5%	-	-	-	3%	-	71%
Single Ch. Alt. Blocks	-	-	1%	-	-	2%	-	24%
Multi Channel Block	-	20%	-	-	-	4%	-	-
Bullet Wake	21%	2%	-	-	6%	-	-	-
Same Row	1%	-	-	-	15%	-	-	1%
Full Channels	-	15%	86%	89%	5%	12%	-	4%
Rectangles	-	1%	3%	4%	-	4%	-	-
Shattered Channel	-	12%	1%	3%	15%	62%	-	-
Quasi Shattered Ch.	-	-	-	-	-	-	-	-
Single Ch. Random	1%	-	4%	2%	20%	13%	-	-
Uncategorized	-	-	-	-	-	-	-	-

Table 4.5: Spatial Classes distributions for the operators not using the shared memory.

Spatial Class	`maxpool`		`avgpool`		`relu`		`sigmoid`		`bias_add`	
	sr	wr	sr	wr	sr	wr	sr	wr	sr	wr
Single	64%	2%	61%	-	100%	-	100%	1%	87%	-
Skip 4	-	-	-	-	-	-	-	-	6%	-
Single Block	-	38%	-	53%	-	-	-	-	-	80%
Single Ch. Alt. Blocks	-	-	-	-	-	-	-	-	-	-
Multi Channel Block	-	27%	-	37%	-	-	-	-	-	11%
Bullet Wake	36%	-	39%	-	-	44%	-	44%	3%	-
Same Row	-	12%	-	1%	-	-	-	-	-	1%
Full Channels	-	-	-	-	-	-	-	-	-	-
Rectangles	-	-	-	-	-	-	-	-	-	-
Shattered Channel	-	2%	-	1%	-	15%	-	43%	-	2%
Quasi Shattered Ch.	-	4%	-	3%	-	41%	-	12%	-	2%
Single Ch. Random	-	11%	-	2%	-	-	-	-	-	4%
Uncategorized	-	-	-	-	-	-	-	-	-	-

5 | Experimental Results: Application-level Analysis

In this chapter, we present an application-level analysis of a CNN performing its task inside a real-world application. In particular, we evaluate the reliability of YOLOv3 [42], an advanced object detection network, already introduced in Section 2.3.2. The goal of this section is twofold: as first, we analyze the reliability of YOLOv3 dissecting the network in its layers and operators to study what are its weak points according to the error model we gathered in Chapter 4; as the second goal we show that CLASSES with the proposed improvements is a flexible framework that enables the developer to perform various analysis, and producing detailed reliability reports.

Section 5.1 of this chapter re-introduces the network under test and the experimental environment around it and discusses the planning of the experiments and the various observed metrics. In Section 5.2 we illustrate and discuss the results of the experiments, planned in the previous section.

5.1. Simulation Experiments Planning

In this section we fully describe the experimental environment, re-introducing YOLOv3, the CNN under test described in Section 2.3.2, and illustrating the environment needed for performing the simulations using CLASSES. Then, in Section 5.1.2 we plan the experiments by defining the characteristics (size, variable) of the various simulation campaigns that we carried out. Finally, in section 5.1.3 we define some relevant metrics that we are going to use for evaluating the reliability of YOLOv3 in our reliability report, in Section 5.2.

5.1.1. Experimental Environment

The graph of the network under test, YOLOv3, contains 106 layers, 72 of which are elementary blocks constituted by a sequence of convolution, batch normalization and

Leaky ReLU. These blocks are connected together in sequence with the addition of various skip connections and residual layers. As shown in Figure 2.10, the last half of the network gradually forks into three detection blocks having the goal of detecting different sizes of objects. The detection blocks are composed of two elementary blocks, an additional convolutional layer (without batch normalization and activation) and a final processing layer that outputs the detected bounding boxes.

We performed the CLASSES simulation over an existing PyTorch implementation of YOLOv3 [34], employing the same computer used for the fault injections in Section 4.1.1. Even if, in this case, we used the same GPU for the injection and the simulation, there is no obligation to do that in the proposed methodology. As test inputs we use images from COCO [33], a state-of-the-art dataset that contains over 200K labeled images, including objects of 91 common object categories (people, animals, objects of various sizes, vehicles and road sizes). For the error simulation, we randomly sampled 300 images, fixed for all the simulation campaigns, chosen from the validation set composed of 5000 images. We used the pre-trained weights for COCO, provided by the authors of YOLOv3 [43].

The error simulation is performed using the CLASSES simulator implemented for PyTorch, to which we made some changes to make the experiment easy to automate in all layers, adding support for PyTorch hooks. A Hook is a function that can be registered to run after a layer is completed. The hook receives as parameters the input, the output, and the context of the executed layer. If the hook returns `None`, the output is not changed. Otherwise, the output is replaced with the return value of the hook. The main advantage of using the hooks is that it does not require changing the graph of the network by adding simulator layers. For example, the user can at first execute the network without simulation to obtain the golden. Then it registers a hook after the target layer. After it finishes with that layer, the user can de-register the hook, and the network returns to normal. To avoid the user forgetting to de-register a simulator hook before registering another, we also provide a Python context manager that automatically deregisters a hook when it goes out of scope.

Using hooks a single run of an error simulation follows these steps:

1. Choose an operator and a related error model that will be used by the simulator for generating the errors.
2. Instantiate the PyTorch module object that represents the whole network, loading the weights.
3. Call the hook builder higher-order function to obtain the hook function that simu-

lates the errors. The hook builder accepts configuration parameters for customizing the experiments.

4. Register the obtained hook in the desired target layer, save the handle obtained from the registration, to later deregister the hook.
5. Run the PyTorch network as usual by calling the network object and collecting the results (bounding boxes coordinates and simulation details). Perform a fixed number of simulations for each one of the 300 images.
6. De-register the simulation hook using its handle. De-registering the hook is important especially if the network has to be reused for an injection in another layer. Failing to do so leads to have multiple hooks and so multiple simulations in different layers.
7. Save the collected results in a JSON file.
8. Repeat steps 3 to 7 for each injectable layer.

All the experiments that we are going to describe in the next section follow the structure described above.

5.1.2. Simulation Campaigns

After setting up the experimental environment, we planned the various experiment campaigns we performed. Using the error models obtained from the experiments that we described in Chapter 4, we can simulate errors in the three operators (Convolution, Batch Normalization, and Leaky ReLU) that constitute the majority of the layers of YOLOv3. For the Convolutional layers, we performed experiments using the three different strategies discussed in Section 2.1.4 and for which we have generated the error models. For each experiment we want to perform we are going to use both Single Random and Warp Random error models, with the Single Random error models representing faults that happened inside register or single memory elements and Warp random error models representing the effects of soft errors in the control logic (i.e. warp scheduler) that corrupted the output of an entire warp. Unfortunately, without having access least the RTL architecture of the GPU, it is not possible to estimate how often the two fault models happen relatively to each other, consequentially we will keep the two models separated in all the experiments and analyses.

Table 5.1 lists and characterizes the experiment campaigns that we performed at the application level. There are three types of experiments, classified by the first letter in its

Table 5.1: Characteristics of the simulation campaigns performed

Id	Operator	Fault Model	Domains	Spatial Classes	Time
A1	conv_gemm	Warp Random	All	All	2h05m
A2	conv_fft	Warp Random	All	All	5h20m
A3	conv_win	Warp Random	All	All	2h00m
A4	batchnorm	Warp Random	All	All	1h58m
A5	relu	Warp Random	All	All	1h39m
A6	conv_gemm	Single Random	All	All	37m
A7	conv_fft	Single Random	All	All	2h44m
A8	conv_win	Single Random	All	All	52m
A9	batchnorm	Single Random	All	All	33m
A10	relu	Single Random	All	All	28m
R1	conv_gemm	Warp Random	In-Range	All	1h30m
R2	conv_fft	Warp Random	In-Range	All	1h29m
R3	conv_win	Warp Random	In-Range	All	1h20m
R4	batchnorm	Warp Random	In-Range	All	1h05m
R5	relu	Warp Random	In-Range	All	1h06m
R6	conv_gemm	Single Random	In-Range	All	33m
R7	conv_fft	Single Random	In-Range	All	1h42m
R8	conv_win	Single Random	In-Range	All	45m
R9	batchnorm	Single Random	In-Range	All	30m
R10	relu	Single Random	In-Range	All	26m
S1	conv_gemm	-	In-Range	Single	33m
S2	conv_gemm	-	Out-Of-Range	Single	29m
S3	conv_gemm	-	NaN	Single	30m
S4	conv_gemm	-	Zero	Single	29m

identifier:

- In the "A" experiments we simulate the errors using the full-fledged error models without any change.
- In the "S" experiments we simulate errors only for single value errors of the various value classes. The most relevant experiment in this group is the one performed with Out-Of-Range value classes where we simulate the effect of a single error spike, to asses its impact on the reliability of the network.
- In the "R" experiments we simulate the errors with a modified version of the error models, where all the domain classes were artificially restricted to be all made of 100% In-Range values. In this way, all the generated erroneous values are inside the range of values present in the golden tensor, simulating the effect clipping layer proposed in [10]. This kind of experiment does not replicate exactly what was done

Table 5.2: Baseline metrics for YOLOv3

Metrics	Original Work [42]	Validation Set	Random Sample
Image Count	-	5000	300
Precision	-	80.8%	81.8%
Recall	-	44.9%	44.4%
F1 Score	-	57.8%	57.6%
Whole Image Accuracy (WIA)	-	23.5%	29.7%
Mean Average Precision (mAP)	57.9%	53.9%	59.4%

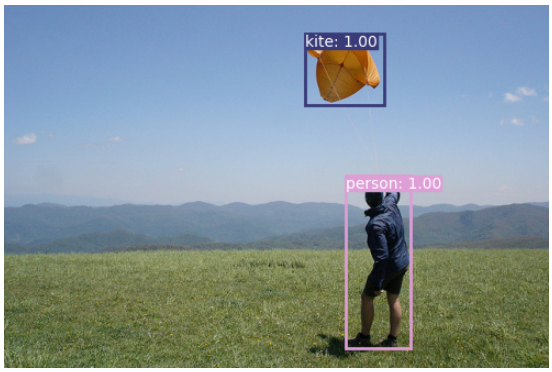
in the work cited above, since it generates uniformly random values in the range instead of clipping the out-of-range values, but we reputed that the effect is close enough.

For some of the campaigns involving Winograd and FFT Convolutions, due to the limitations of the implemented simulator, errors in the first layers took a very long time to be generated, so we skipped the first 10 layers to have a feasible execution time. The reason behind this limitation stands in the fact that the first layers in YOLOv3 have the feature maps with the biggest channels, as big as 416×416, and it may take longer to times to simulate spatial classes that generate big amounts of errors, like the Full Channels class. To perform fair comparisons between operators in Section 5.2, when we calculate metrics that aggregate all the layers of the network, we will consider only the layers from 13 onwards, unless stated differently.

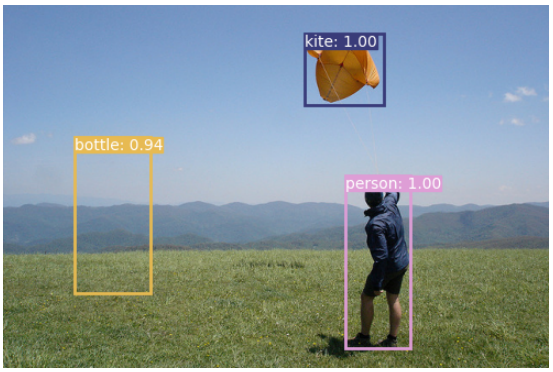
5.1.3. Metrics

Before starting with the application-level simulations we need to be sure that the network that we use for the experiments performs similarly to the network described in the paper, otherwise the simulations may be not valid. We describe the baseline metrics that we evaluated here in Section 2.3.2. Table 5.2 shows the values of the baseline metrics. We can see that the mAP of the validation set we used is not too far from the one claimed in the original work. The mAP of the sample we use in the simulation is even higher than the original one, probably due to some statistical fluctuation caused by the smallness of the sample.

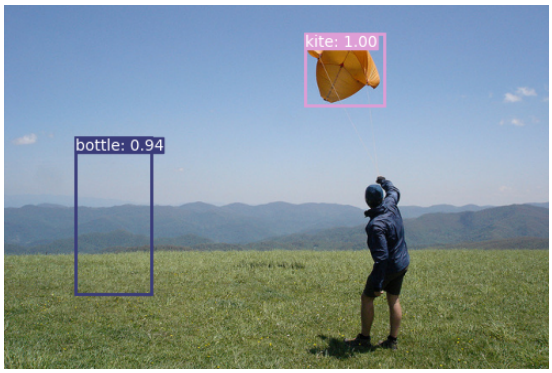
To evaluate the resilience of the network to soft errors we will employ some of the metrics already described in Section 2.3.2, introducing also some modifications of those metrics in order to make them better suited for the reliability evaluation. The metrics above are all calculated by comparing the results of the inference performed by the golden network on each image of the dataset against the annotated dataset, taken as the ground truth.



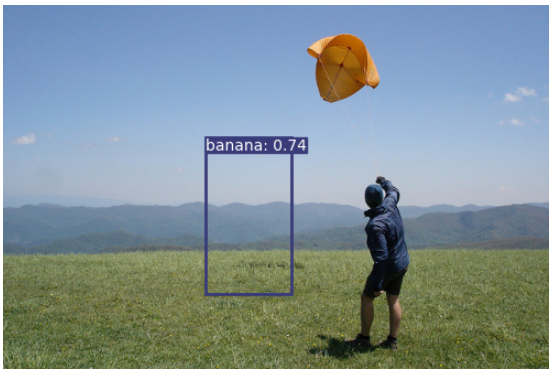
(a) Annotated image



(b) Golden detection



(c) Corrupted detection 1



(d) Corrupted detection 2

Figure 5.1: Example of golden and corrupted detections.

On top of the described metrics, we use a new set of relative metrics, similar to the ones proposed above but calculated by comparing the results from the error simulations with the results of the inference of the golden network taken as the ground truth instead of the annotated dataset. We call these new metrics *Relative Precision*, *Relative Recall*, *Relative WIA* (RWIA) and *Relative mAP* (RmAP) to distinguish them from their absolute counterpart.

Consider the example in Figure 5.1, which uses an image from COCO. The absolute metrics are calculated by comparing the BBs in the corrupted figures (5.1c and 5.1d) with the annotations on the dataset (5.1a). In this case, we will have one true positive (kite in 5.1c), three false negatives (person in 5.1c and 5.1d and kite in 5.1d) and two false positives (the bottle and the banana), scoring 33% in absolute precision and 25% absolute recall. The relative metrics, calculated using the golden inference as the ground truth, are different since the hypothetical network wrongfully detected a false bottle. So the bottle in Figure 5.1c is considered a true positive and its absence in Figure 5.1c is counted as a false negative. The relative precision and recall are respectively 66% and 40%. Note that the absolute and relative WIA are both 0% in this example since no corrupted image has

only true positives.

A metric that evaluates the resilience of a CNN to a soft error, using the Usability-Based criterion (Section 2.5) is the *Program Vulnerability Factor* (PVF) [50], that measures the percentage of faults in a program execution that results in critical errors. In the context of object detection, we define critical error every detection with at least a false negative or a false positive. In this case, PVF is the dual of the relative WIA (RWIA) can simply be calculated as:

$$\text{PVF} = 1 - \text{RWIA} \tag{5.1}$$

There are other kinds of *Vulnerability Factors* (VFs) that can be calculated by simulating errors only on some parts of the network, such as single layers or operators. For example, the *Layer Vulnerability Factor* (LVF), measures the percentage of faults in a given layer that lead to critical errors, and similarly, the *Operator Vulnerability Factor* (OVF), measures the percentage of faults simulated only in a given kind of operators that leads to critical errors and it is calculated by averaging all the LVFs of the operator. In the following section, we will use the relative metrics obtained from the results of the experiments to make considerations about the reliability of the network under test.

5.2. Experiment Results

In this section, we show and discuss the results obtained from the simulation campaigns planned in the previous section. We will analyze different levels of the networks, starting from the layers that constitute the network, up to the five different operators for which we performed fault injections. We will try to put the results that we obtained in the context of the network, and we will analyze the connections between the different kind of analysis we are going to perform.

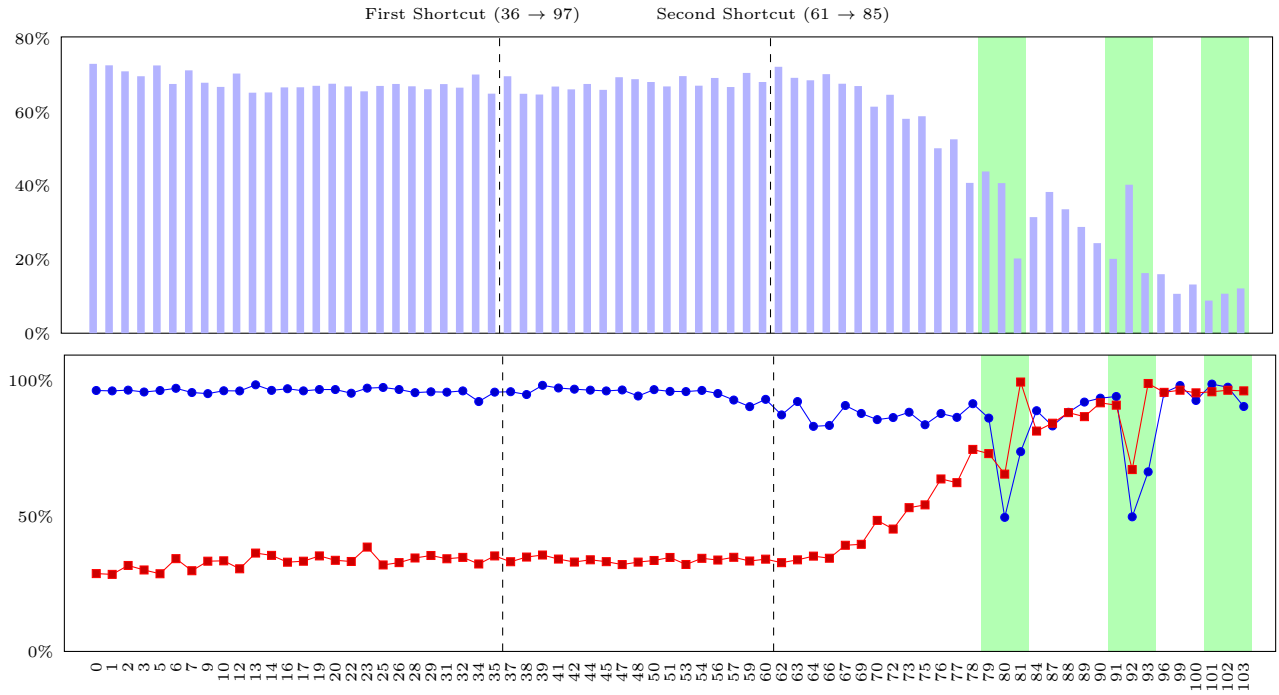
5.2.1. Layer Vulnerability Analysis

In this section, we analyze and compare the resilience of the different layers to faults. Since the results experiments are collected layer by layer, performing layer Vulnerability analysis is straightforward. A motivation behind this kind of analysis is the possibility of using the data to propose hardening designs that increase the reliability of the network. A trivial way to radically improve the resilience of CNNs to soft errors is to duplicate the entire network and then compare the two results. If they are different then an SDC

event is detected and the computation has to be repeated. However, the duplication of an entire network can be unfeasible in terms of hardware resources and energy consumption. A cheaper way to increase reliability can be to selectively harden the most vulnerable layers [7], hence motivating the need of layer vulnerability analysis.

Figure 5.2 shows some plots of the LVF, Relative Precision and Relative Recall of the YOLOv3 GEMM convolution layers for both Warp Random and Single Random error models. For the sake of space, we omitted the LVF plots of other operators, also because they follow similar patterns to the GEMM convolution plot, differing mainly on the maximum and average LVFs. The x-axis is annotated with the ID of the layers using the original YOLOv3 numeration. In both error models, the LVF remains almost constant to high values (around 75% for Warp Random and around 33% for Single Random) until the second shortcut layer, then it starts decreasing almost monotonically. The Relative Precision and Recall plots can tell whether the errors are caused mostly by the presence of false positives or false negatives. In all layers excluding the ones inside the detection blocks the recall is remarkably lower than the the precision, meaning that the errors are mostly caused by false negatives (missed detections). In the detection blocks the situation changes with the precision, that otherwise is close to 95%, dropping suddenly. The effect of this drop in precision can be particularly seen, in layer 92 in the graph 5.2a causing an abnormal resurgence of the LVF of that layer. From comparing the plots of the LVFs of Warp Random and Single Random, we notice that Warp Random errors have a greater impact on the reliability of the CNNs w.r.t. Single Random errors, with the most vulnerable layer having respectively an LVF of 78% and 37%. This result is expected since Warp Random errors are generated corrupting 32 threads at once while Single Random errors are generated corrupting only a single thread, and it implies that faults in the control units such as the warp scheduler lead more easily to critical error than faults on the memory elements. In the next section, when we analyze the various OVFs we will continue to observe this difference between the two error models.

(a) Convolution GEMM, Warp Random Errors (Experiment A1)



(b) Convolution GEMM, Single Random Errors (Experiment A6)

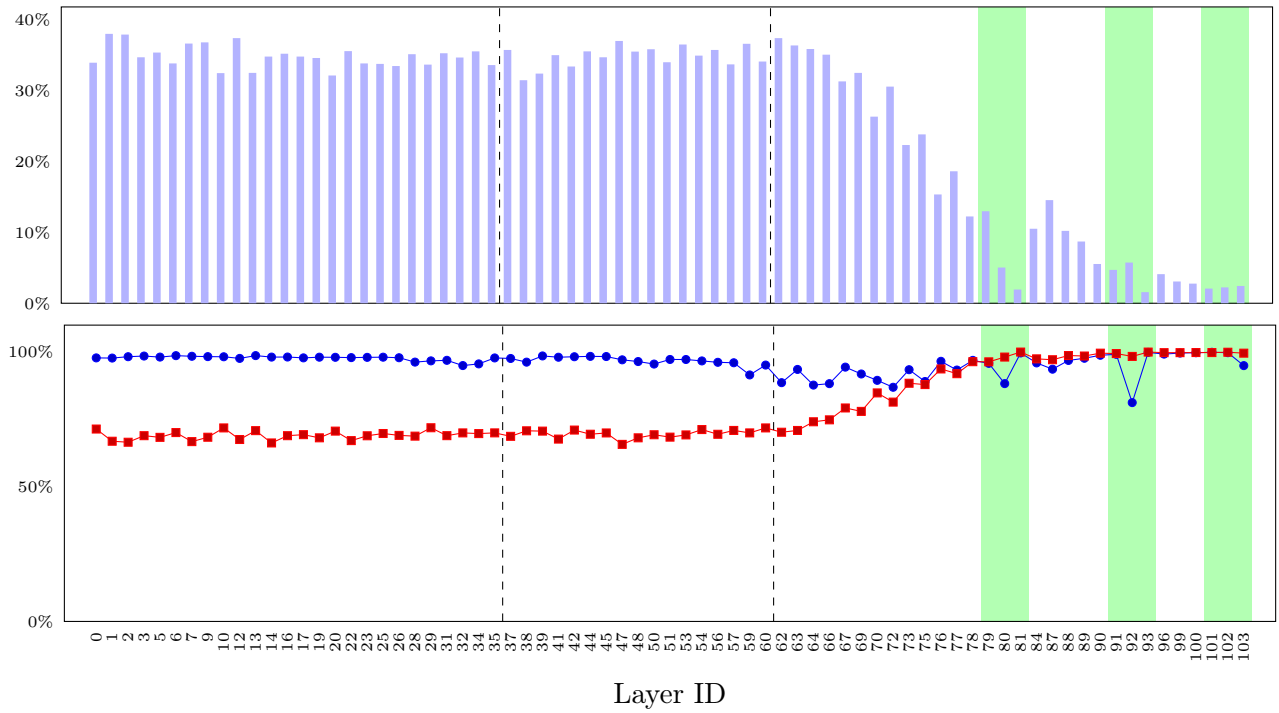


Figure 5.2: LVF, Relative Precision and Relative Recall for each GEMM Convolution layer

5.2.2. Operator Vulnerability Analysis

In this section we will analyze the reliability of the network at the level of the operator, to understand which operators are more vulnerable to faults, limiting ourselves to the scenario where we consider the original domain distributions. We will also make a comparison between the three strategies of the convolution. The data illustrated in this section is obtained by analyzing experiments A1 to A10, with all the metrics obtained calculated as averaging the ones of the layers (starting from layer 13 in all scenarios, as we explained before in Section 5.1.2).

Figure 5.3 compares the OVFs of the operators constituting YOLOv3 using both the Warp Random and Single Random models. Coherently on what we see in Figure 5.2 and what we discussed in the previous section, the OVFs of the experiments using Warp Random models are consistently higher than the Single Random ones. From comparing the OVFs of the operators we notice that the convolutions are the most vulnerable to Warp Random fault with OVFs going from 60 to 80% depending on the strategy while Batch Normalization and ReLU have OVFs respectively at 52% and 41%. A little bit different is the case for the Single Random error where the GEMM convolution has the lowest OVF (24%) compared to the slightly higher OVFs occurring to Batch Normalization (25%) and Relu (29%), and the very high OVF of the FFT convolution (59%) which will be better analyzed in the comparative analysis of the convolutions in Section 5.2.4.

To better understand whether the errors were caused mostly by false positives or false negatives, in Figure 5.4 we plotted relative precision and relative recall of the various operators and experiments. We can see that precision is barely affected in most of the cases, meaning that the faults cause few false positives. On the contrary, recall is consistently low, especially in the Warp Random experiments where it reaches the minimum of 26% in the Winograd convolution, meaning that, on average, almost three-quarters of the detections made by the network are missed (false negatives) in the event of a fault in the control scheduler.

5.2.3. Range Restriction Analysis

In this section, we analyze the data coming from experiments R1 to R10, where the domains of the erroneous values were restricted to the range of the values of the golden feature map, as previously discussed in Section 5.1.2. We will also analyze the results of experiments S1 to S4 studying how single erroneous values of different value classes can affect the reliability of the network.

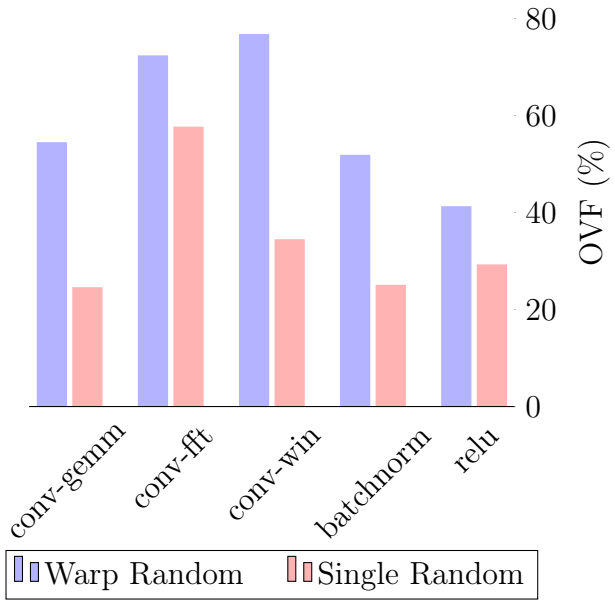


Figure 5.3: OVF for Warp Random and Single Random simulations

Similarly to 5.3, Figure 5.5 represents the OVFs computed from the range-restricted experiments. We can easily observe a pronounced reduction in all the OVFs with respect to the unrestricted experiments, especially for the GEMM convolution, the batch normalization, and the ReLU, which in all cases have an OVF less than 10%. This fact shows that range restriction can be an effective strategy for increasing the reliability of CNNs, in accordance with the conclusions present in the literature [10]. However, range restriction is less effective in FFT and Winograd convolution, where the reduction in OVF was less pronounced; we will better investigate the reason in the next section.

Table 5.3 collects the results of experiments S1 to S4, showing the effects of single error spikes. In each one of these experiments, we simulated single-value errors happening in convolutional layers restricted to one of the four value classes. For each one of these experiment campaigns, we report Relative Precision, Relative Recall and OVF. Without any surprise, Out-Of-Range values generate critical way more often than In-Range values. The OVF of the Out-Of-Range values is very high for being just a single corrupted value inside feature maps that contain up to thousands of values. A reason why just a single value out of range can cause so many critical errors can be found in how the information is represented in the feature maps of a CNN. What contains information in a feature map is not just the absolute value of every element of the feature map, but more importantly, the difference between all the elements in the feature map, especially those that are spatially close or neighbors. This means that a single value out of range can easily cancel the information contained in its neighborhood, since the difference between its uncorrupted

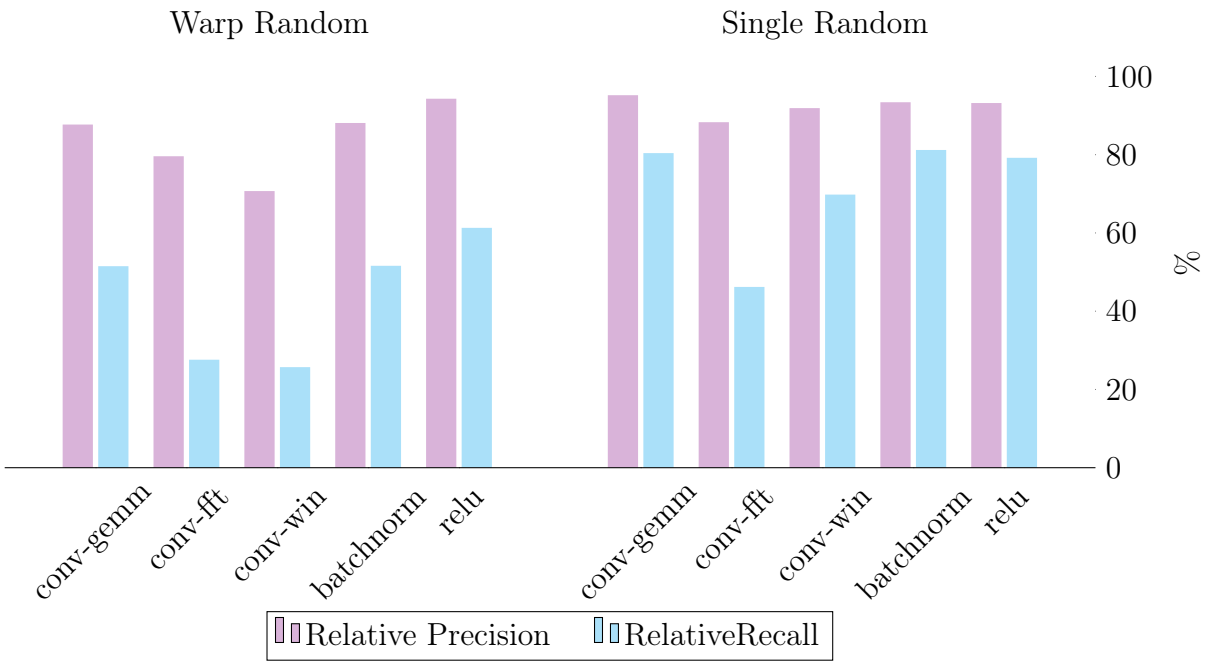


Figure 5.4: Precision and Recall for each operator

Table 5.3: Metrics obtained from Single value experiments

Metrics	In-Range	Zero	Out-Of-Range	NaN
Relative Precision	99.6%	96.8%	86.6%	-
Relative Recall	99.6%	98.1%	64.3%	0%
OVF	1.5%	9.3%	47.9%	100%

neighbor values, which are in the range by default, becomes way less significative compared to the absolute value of the spike.

From Table 5.3, we can also observe that a single NaN value is always completely destructive, spreading in convolutional layers until all the values of the feature maps are NaN because every operation with a NaN operand has NaN as a result. This causes the application to completely miss all the detections every time. To mitigate this problem, we can easily detect NaN and replace them with zeros, or even a random In-Range value.

5.2.4. Comparative Analysis of Alternative Convolutions

In this section, we will use the data gathered in the full-fledged and range-restricted experiments to make a comparative analysis of the three strategies for the convolutional layers (GEMM, FFT and Winograd). It must be noted that in all the experiments we tested one convolution at a time, never mixing the convolution in the same inference.

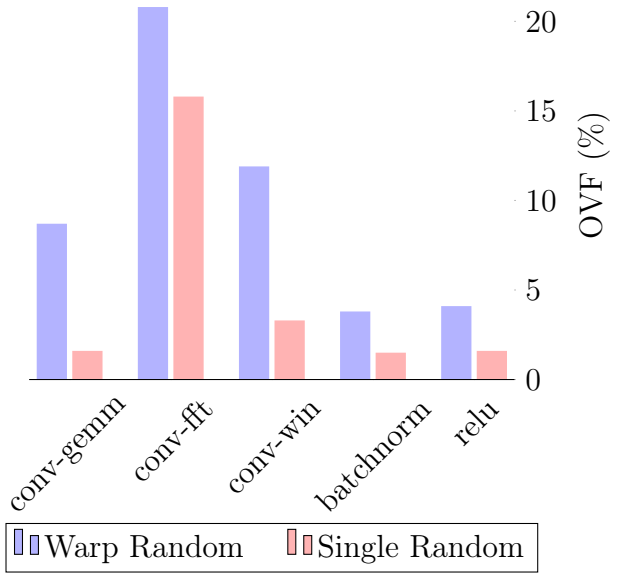


Figure 5.5: OVF for range-restricted operators

We begin the comparison by looking at the OVFs in the full-fledged models, plotted in Figure 5.3. From this data, we can compare the vulnerability of the three different strategies for convolution without any clipping mitigations. We observe that FFT and Winograd strategies have way higher OVFs than the classic GEMM convolution, with the Winograd convolution having the highest OVF (77%) in the Warp Random experiments and the FFT Single Random ones (57%), compared to the GEMM that has OVFs of 55% and 26% respectively in the Warp Random and Single Random experiments.

For the range-restricted scenario experiments, plotted in Figure 5.5, we notice a drop in the OVFs of all the implementations. Even in this scenario, the GEMM convolution has the highest resilience to the faults of the three strategies. The range restriction is way less effective for the FFT convolution where the OVFs are 21% and 16% respectively for the Warp and Single random fault models, compared to the Winograd convolution that has the same metrics equal to 12% and 3.3%. This confirms the hypothesis made in Section 4.3 where we expected to see a high OVF in the FFT convolution, caused by the high incidence of Full Channel errors in its error models. In the next section, we will have another confirmation of this hypothesis by looking at the vulnerability factor of the Full Channel spatial class.

Table 5.4: Vulnerability Factors of the spatial classes using the Single Random error models

Spatial Class	conv_gemm		conv_fft		conv_win		batchnorm		relu	
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
Single	24%	1.5%	-	-	24%	1.6%	25%	1.5%	29%	1.6%
Skip 4	-	-	-	-	-	-	25%	1.6%	-	-
Bullet Wake	29%	1.9%	-	-	35%	1.9%	-	-	-	-
Same Row	-	-	-	-	35%	2.0%	-	-	-	-
Full Channels	-	-	59%	17%	51%	18%	-	-	-	-
Shattered Channel	-	-	-	-	42%	4.7%	-	-	-	-
Single Ch. Random	-	-	-	-	38%	2.9%	-	-	-	-

Note: (1) Unrestricted Range (2) Only In-Range Domain Classes

Table 5.5: Vulnerability Factors of the spatial classes using the Warp Random error models

Spatial Class	conv_gemm		conv_fft		conv_win		batchnorm		relu	
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
Skip 4	66%	2.7%	-	-	-	-	-	-	-	-
Single Block	45%	3.0%	-	-	-	-	56%	2.6%	-	-
Single Ch. Alt. Block	-	-	-	-	-	-	37%	6.3%	-	-
Multi Channel Block	31%	9.1%	-	-	73%	4.7%	-	-	-	-
Bullet Wake	-	-	-	-	-	-	-	-	35%	2.7%
Full Channels	43%	28%	71%	22%	74%	29%	-	-	-	-
Rectangles	-	-	74%	14%	75%	8.6%	-	-	-	-
Shattered Chan.	67%	9.9%	-	-	78%	11%	-	-	82%	6.3%
Quasi Shattered Chan.	-	-	-	-	-	-	-	-	33%	4.3%

Note: (1) Unrestricted Range (2) Only In-Range Domain Classes

5.2.5. Spatial Classes Analysis

In this section, we will perform an analysis operator by operator of the criticality of the errors generated by the various spatial classes. For getting the results reported in Tables 5.4 and 5.5 for each operator we extracted all the spatial classes with a relative frequency greater or equal to 4%, as reported in Tables 4.4 and 4.5. The spatial classes with a lower frequency than 4% were not considered since there were not enough data points for a good estimation of the VF. For each one of the extracted spatial classes, we filtered out all the simulations not using that class and we computed the VF using Equation 5.1.

Table 5.4 contains the spatial classes VFs obtained using the Single Random error models in both range-unrestricted (column 1) and range-restricted (column 2) simulations. For

the range unrestricted simulations, most of the classes have VFs between 20%-40% except for the Full Channels class, which as we anticipated in the previous section can cause critical errors in the application. Restricting the range causes the VF of all the classes to drop between 1%-5%, with the notable exception of the Full Channels class for which the VF is 18%-19%. Values of VF this high in the FFT convolution can be explained by looking at the relative frequency of the spatial classes in this operator (shown in Table 4.4). The Full Channel patterns appear more than 80% of the time in Single and Warp Random fault injections, and as we previously stated the Full Channel pattern is very prone to critical errors even after range restriction.

Table 5.5 has the same structure as Table 5.4 but contains the spatial classes VFs for Warp Random experiments. The VF obtained from Warp Random is naturally higher, reaching peaks of more than 80% for the unrestricted shattered channel class in the ReLU operator. Similarly to the Single Random table, the range limitation is effective in lowering the VF factor in all the operators, but the Full Channels pattern has still a relatively high VF compared to the other classes. An explanation of this behavior comes from the definition of the Full Channel pattern requiring that each corrupted channel has more than 50% of corrupted values (see Section 4.2), meaning that full channel errors corrupt a large number of values, especially in feature maps with the biggest channel and even if there is a range restriction in place if an error has a huge number of corrupted values it has a bigger chance to lead to a critical fault.

In this chapter, we conducted a comprehensive analysis of the reliability of YOLOv3, in order to gain in-depth insights into the reliability of the network and to show how flexibly the results of CLASSES simulations can be used to produce reliability reports. After describing the experimental environment we planned three types of campaigns to asses the resilience to soft errors of the network in a normal operation, with range restriction layers and in a hypothetical scenario with only single values. After describing the metrics we analyzed the results of the experiments making conclusions about the reliability of the network valid in the operating scenario described in chapter 4. The main takeaways of the analysis are:

- The most critical layers in YOLOv3 are positioned before the second shortcut. In the final layers of the network, the LVF tends to decrease.
- While FFT and Winograd convolutions can sometimes be faster than GEMM convolutions, in critical applications we need to be careful before using them, since in our scenario they generated way more critical errors than the GEMM convolutions.
- A single erroneous value out of the range of the golden feature map can create more

critical errors than several errors in the range of the feature map.

- Range restriction methods have the potential of bringing down the vulnerability factors by an order of magnitude, especially with layers that generate lots of Out-Of-Range values in the event of a fault. However, their effectiveness drops with spatial classes that generate errors with a big cardinality, like in the case of the Full Channel pattern.

In the next chapter, we will wrap up the content described in all the chapters and we will propose possible improvements to the current work.

6 | Conclusions and future developments

Since Automatic Driving Systems (ADS) are safety-critical systems, it is essential to thoroughly assess their reliability before deploying them. This evaluation must also align with international standards like ISO 26262, which impose stringent safety requirements for protecting the safety of both occupants of the vehicle and other road users. However, due to the intricate nature of the hardware (such as GPUs) and software (like CNNs) used in ADS, waiting until the entire system is complete for a comprehensive reliability analysis, followed by fixing emerging vulnerabilities, is impractical. To address this challenge, there is a need to accelerate the development cycle by gaining insights into the reliability of individual components before assembling them into the complete system and conducting a full-scale analysis.

Various methodologies in the literature offer solutions for this advanced reliability analysis. One of these is CLASSES [8], a cross-layer error modeling methodological framework that combines platform-level fault injection with software-level error simulation, using a set of error models as the interface between the two layers, summarizing the information of the fault injection experiments and using that for generating the errors in the simulation. We noticed that the error models are a crucial part of the framework and we found some weak points in their definition in the previous work. In particular, we noticed that there was no systematized way to plan and execute the experiments and the error model structure was not flexible for summarising the information of some new patterns emerging from the fault injection experiments. Moreover, the existing error modeling process was for its greater part manual and long to perform.

In this work, we proposed some improvements on all these fronts. We rationalized the process of planning the fault injection campaigns by formalizing a way to define how numerous experimental variables change in the different experimental batches with the result of having a new planning methodology that can be easily adapted in various experimental environments. We suggested a new error model structure where we revisited how

the spatial and domain distributions are stored with the goal of creating error models that can reproduce, during simulation, the errors coming from fault injections with acceptable fidelity. For the spatial distributions, we limited the use of absolute parameters for the distributions in favor of more flexible relative parameters that can be scaled to different tensor sizes. The domain part of the error models now characterizes the distributions of the value classes in the context of the entire tensors instead of considering all the erroneous values independently. Additionally, we provided a tool that can be used to automate some parts of the error modeling step, for instance by helping the user to visualize the errors in the tensors so that they can quickly infer a classification in spatial classes, necessary for automatically generating the error models, using the same tool.

For testing and validating the approach proposed in this work, we planned and run fault injection campaigns in various standalone operators extracted from various networks, such as LeNet5 and YOLOv3, and implemented in CuDNN, using an Ampere NVIDIA GPU. We used the results of the campaigns in combination with the processing tool implemented for classifying the tensors in spatial classes and for generating the error models based on the extracted classes. We then reused the models for analyzing the reliability of YOLOv3, a full-fledged object recognition network. The flexibility offered by the simulator allowed us to obtain insights into the reliability of YOLOv3, allowing us to calculate the vulnerability factors at the level of detail of single layers and single operators. The latter allowed us to compare the resilience to soft errors of different strategies for convolution using two different fault models. From the results, the GEMM convolution was less sensitive to soft errors than the FFT and Winograd convolutions. The rich configurability simulator also allowed us to simulate the effect of a range restriction hardening method proposed in the literature, concluding that this kind of methodology effectively improves the reliability of the network, by protecting the feature maps from a single erroneous location with a spiking out-of-range value that alone is enough to critically corrupt the result of the inference.

6.1. Future Works

There are several open points that can be investigated in the next future:

- **Extension to other computing devices**: CLASSES was born as a framework for analyzing CNNs executed in the GPU. We think, however, that this framework can be easily extended to other types of devices such as FPGAs and dedicated hardware accelerators, by using specialized injectors or hardware simulators for performing fault injection campaigns.

- **In-Depth Domain Analysis**: While in this work we improved the characterization of the domains in the error models, we think that it is possible to improve it even more, especially because in this work we noticed that the domains of the corrupted values can have a great impact on the reliability of the network. For example, instead of having a binary classification (In-Range and Out-Of-Range) of the "valid" erroneous floating points, we can study how the value of the deviation from the golden values affects the reliability.
- **Fully Automation of Error Modeling**: In this work, we made a step towards the full automation of the error modeling phase, however, we did not manage to automate completely the step. Various methods can be studied to automate this step, like, for example, non-supervised automatic clustering of the faulty tensors based on their spatial and/or domain distributions.
- **Quantization**: Normally, the weights and the feature maps values are 32-bit floating point. Different approaches [23], quantize the network having weights and/or hidden feature maps representing their values with integers smaller than 32 bits, to reduce the memory footprint and speed up the inference. The proposed methodology is mostly compatible with quantization, with the notable exclusion of the value classes that should be revisited, since In-Range and Out-Of-Range classes may become ill-defined in this case. Also, the network under test needs to be reimplemented and re-trained for supporting quantization.
- **Expand the variety of networks analyzed**: In this work, due to time limitations, we performed the application-level reliability analysis only on a single CNN, YOLOv3. This analysis could be extended to other networks, even ones that execute different tasks other than object detection, such as classification, segmentation and regression. Other networks can be analyzed easily, even without any additional fault injection campaigns (if we use the same experiment environment). It could be interesting also to analyze other types of networks, such as Recurrent Neural Networks (RNN) [48] that maintain an internal state between inferences, with the possibility, in the event of a fault, of spreading errors in multiple outputs.

Bibliography

[1] Nvidia cudnn developer guide. `http://web.archive.org/web/20230630023042/https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html`. Accessed: 2023-07-28.

[2] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019. doi: 10.1109/IEEESTD.2019.8766229.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

[4] T. Athil, R. Christian, and Y. B. Reddy. Cuda memory techniques for matrix multiplication on quadro 4000. In *2014 11th International Conference on Information Technology: New Generations*, pages 419–425. IEEE, 2014.

[5] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316, 2005.

[6] M. Biasielli, C. Bolchini, L. Cassano, E. Koyuncu, and A. Miele. A neural network based fault management scheme for reliable image processing. *IEEE Transactions on Computers*, 69(5):764–776, 2020. doi: 10.1109/TC.2020.2965518.

[7] C. Bolchini, L. Cassano, A. Miele, and A. Nazzari. Selective hardening of cnns based on layer vulnerability estimation. In *2022 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2022. doi: 10.1109/DFT56152.2022.9962339.

[8] C. Bolchini, L. Cassano, A. Miele, and A. Toschi. Fast and accurate error simulation for cnns against soft errors. *IEEE Transactions on Computers*, 72(4):984–997, 2022.

[9] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben. Tensorfi: A flexible fault injection framework for tensorflow applications. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 426–435. IEEE, 2020.

[10] Z. Chen, G. Li, and K. Pattabiraman. A low-cost fault corrector for deep neural networks through range restriction, 2021.

[11] F. Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[12] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky. NVIDIA a100 tensor core GPU: Performance and innovation. *IEEE Micro*, 41(2):29–35, Mar. 2021. doi: 10.1109/mm.2021.3061394. URL `https://doi.org/10.1109/mm.2021.3061394`.

[13] J. E. R. Condia, B. Du, M. S. Reorda, and L. Sterpone. FlexGripPlus: An improved GPGPU model to support reliability analysis. *Microelectronics Reliability*, 109:113660, June 2020. doi: 10.1016/j.microrel.2020.113660. URL `https://doi.org/10.1016/j.microrel.2020.113660`.

[14] J. E. R. Condia, F. F. dos Santos, M. S. Reorda, and P. Rech. Combining architectural simulation and software fault injection for a fast and accurate cnns reliability evaluation on gpus. In *2021 IEEE 39th VLSI Test Symposium (VTS)*, pages 1–7. IEEE, 2021.

[15] J. E. R. Condia, J.-D. Guerrero-Balaguera, F. F. Dos Santos, M. S. Reorda, and P. Rech. A multi-level approach to evaluate the impact of gpu permanent faults on cnn's reliability. In *2022 IEEE International Test Conference (ITC)*, pages 278–287, 2022. doi: 10.1109/ITC50671.2022.00036.

[16] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23 (4):14–19, July 2003. doi: 10.1109/mm.2003.1225959. URL `https://doi.org/10.1109/mm.2003.1225959`.

[17] F. F. dos Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech. Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 169–176. IEEE, 2017.

[18] F. F. dos Santos, A. Kritikakou, J. E. R. Condia, J. D. G. Balaguera, M. S. Reorda, O. Sentieys, and P. Rech. Characterizing a neutron-induced fault model for deep neural networks, 2022.

[19] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. Gpu-qin: A method-

ology for evaluating the error resilience of gpgpu applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230, 2014. doi: 10.1109/ISPASS.2014.6844486.

[20] P. N. Glaskowsky. Nvidia's fermi: the first complete gpu computing architecture. *White paper*, 18, 2009.

[21] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[22] L. A. Goodman. On simultaneous confidence intervals for multinomial proportions. *Technometrics*, 7(2):247–254, 1965.

[23] Y. Guo. A survey on methods and theories of quantized neural networks. *arXiv preprint arXiv:1808.04752*, 2018.

[24] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. 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)*, pages 249–258. IEEE, 2017.

[25] P. Henderson and V. Ferrari. End-to-end training of object class detectors for mean average precision, 2016.

[26] International Organization for Standardization. Road vehicles – functional safety. ISO Standard 26262, 2018. URL `https://www.iso.org/standard/68383.html`.

[27] L. Jia, Y. Liang, X. Li, L. Lu, and S. Yan. Enabling efficient fast convolution algorithms on gpus via megakernels. *IEEE Transactions on Computers*, 69(7):986–997, 2020. doi: 10.1109/TC.2020.2973144.

[28] T. Karnik and P. Hazucha. Characterization of soft errors caused by single event upsets in cmos processes. *IEEE Transactions on Dependable and secure Computing*, 1(2):128–143, 2004.

[29] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4013–4021, 2016.

[30] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791. URL `https://doi.org/10.1109/5.726791`.

[31] Y. LeCun, C. Cortes, and C. Burges. The mnist dataset of handwritten digits (images). *NYU: New York, NY, USA*, 2, 1999.

[32] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 502–506. IEEE, 2009.

[33] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár. Microsoft coco: Common objects in context, 2014.

[34] E. Linder-Norén. Pytorch-yolov3, a minimal pytorch implementation. `https://github.com/eriklindernoren/PyTorch-YOLOv3`, 2023. Accessed: 28-08-2023.

[35] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari. Pytorchfi: A runtime perturbation tool for dnns. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 25–31, 2020.

[36] C. McClanahan. History and evolution of gpu architecture. *A Survey Paper*, 9:1–7, 2010.

[37] M. A. Mercioni and S. Holban. The most used activation functions: Classic versus current. In *2020 International Conference on Development and Application Systems (DAS)*, pages 141–145, 2020. doi: 10.1109/DAS49615.2020.9108942.

[38] Nicolas Vasilache, Jeff Johnson, Michael Mathieu, Soumith Chintala, Serkan Piantino, Yann LeCun. Fast Convolutional Nets With fbfft: A GPU Performance Evaluation. In *Proc. Int. Conf. on Learning Representations*, pages 1–17, 2015.

[39] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, 1996. doi: 10.1109/23.556861.

[40] S. of Automobile Engineers (SAE International). Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. `https://www.sae.org/standards/content/j3016_202104`, 2021.

[41] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019. URL `https://arxiv.org/abs/1912.01703`.

[42] J. Redmon and A. Farhadi. Yolov3: An incremental improvement, 2018. URL https://arxiv.org/abs/1804.02767.

[43] J. C. Redmon. Yolov3 configuration. https://pjreddie.com/darknet/yolo/, 2023. Accessed: 26-08-2023.

[44] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks, 2015. URL https://arxiv.org/abs/1506.01497.

[45] A. Ruospo, L. M. Luza, A. Bosio, M. Traiola, L. Dilillo, and E. Sanchez. Pros and cons of fault injection approaches for the reliability assessment of deep neural networks. In *2021 IEEE 22nd Latin American Test Symposium (LATS)*, pages 1–5. IEEE, 2021.

[46] A. Ruospo, G. Gavarini, C. de Sio, J. Guerrero, L. Sterpone, M. S. Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale. Assessing convolutional neural networks reliability through statistical fault injections. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023. doi: 10.23919/DATE56975.2023.10136998.

[47] A. Ruospo, E. Sanchez, L. M. Luza, L. Dilillo, M. Traiola, and A. Bosio. A survey on deep learning resilience assessment methodologies. *Computer*, 56(2):57–66, 2023.

[48] H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee. Recent advances in recurrent neural networks. *arXiv preprint arXiv:1801.01078*, 2017.

[49] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. 2015.

[50] V. Sridharan and D. R. Kaeli. The effect of input data on program vulnerability. In *Workshop on System Effects of Logic Soft Errors (SELSE-5)*, 2009.

[51] A. A. Taha and A. Hanbury. Metrics for evaluating 3d medical image segmentation: analysis, selection, and tool. *BMC Medical Imaging*, 15(1), Aug. 2015. doi: 10.1186/s12880-015-0068-x. URL https://doi.org/10.1186/s12880-015-0068-x.

[52] S. K. Thompson. Sample size for estimating multinomial proportions. *The American Statistician*, 41(1):42–46, 1987.

[53] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler. Nvbitfi: Dynamic fault injection for gpus. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 284–291. IEEE, 2021.

[54] B. Xu, N. Wang, T. Chen, and M. Li. Empirical evaluation of rectified activations in convolutional network, 2015. URL https://arxiv.org/abs/1505.00853.

[55] A. Ćorović, V. Ilić, S. Đurić, M. Marijan, and B. Pavković. The real-time detection of traffic participants using yolo algorithm. In *2018 26th Telecommunications Forum (TELFOR)*, pages 1–4, 2018. doi: 10.1109/TELFOR.2018.8611986.

List of Figures

List of Tables