POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A methodology for the reliability analysis and the efficient hardening of Convolutional Neural Networks

Tesi di Laurea Magistrale in
Computer Science and Engineering -
Ingegneria Informatica

Author: **Alessandro Nazzari**

Student ID: 945370
Advisor: Prof. Antonio Rosario Miele
Co-advisors: Luca Cassano
Academic Year: 2021-22

Abstract

Convolutional Neural Networks (CNNs) usage has been steadily increasing in the last decade, especially for perception functionalities in both safety-critical systems and not.

An example of this phenomenon are the Autonomous Driving Systems (ADS), a set of high-level functionalities to substitute the human driver. Multiple ADS components, such as image recognition for the detection of street signs or steering angle detection, integrate CNNs due to their ability to deal with images and video inputs.

Due to the high relevance of these components, the effects of faults striking them would be disastrous for the users' lives. For this reason, design principles for digital systems in safety-critical applications are strictly regulated by standards such as ISO 26262 [1]. Similarly, the Society of Automotive Engineers (SAE) regulates ADS functionalities. Both standards require a high level of reliability and fault detection mechanisms.

It is thus necessary to be able to analyze the reliability and robustness of a given CNN to identify the most promising ways to harden the processing system. Moreover, it is crucial to perform such an analysis at the early stages of the development flow to give fast feedback for design refinements. From a literature review emerges that the currently available tools of analysis present various limitations, either technological or methodological ones.

In this thesis, we selected an existing error simulator for CNNs called CLASSES with the goal of improving and adopting it to design a reliability analysis methodology. We then designed and executed an error simulation campaign using CLASSES targeting a set of CNNs used in the ADS field. The objectives of this campaign were twofold: validate the framework's effectiveness and assess the robustness against faults of multiple CNNs. We then defined a hardening strategy for CNNs capable of exploiting in an automated and effective way information produced by CLASSES.

Keywords: Reliability, Convolutional Neural Networks, Graphic Processing Units, Deep Learning, Artificial Intelligence, Fault Tolerance

Abstract in lingua italiana

L'uso di Reti Neurali Convolutive (RNC) è cresciuto costantemente nell'ultima decade, in particolare per quanto riguarda le funzionalità di percezione in sistemi safety-critical e non. Un esempio di questo fenomeno è rappresentato dal settore dei sistemi di Guida Autonoma. Un set di funzionalità di alto livello con lo scopo di sostituire, parzialmente o totalmente, il guidatore umano. Molti componenti di questi sistemi integrano RNC per via delle loro capacità di elaborare immagini o input audiovisivi. Esempi di queste funzionalità comprendono il riconoscimento di immagini per il riconoscimento di cartelli stradali o la predizione dell'angolo di rotazione del volante a partire da telecamere a inquadratura frontale.

Per via dell'importanza di questi componenti l'effetto di guasti che li colpiscano sarebbe disastroso per la vita degli utenti del sistema. Per questo motivo i principi di progettazione di sistemi digitali usati in applicazioni safety-critical sono strettamente regolati da standard come ISO 26262 e analogamente la Society of Automotive Engineers (SAE) regola le funzionalità dei sistemi di guida autonoma. Entrambi gli standard richiedono un alto livello di affidabilità e meccanismi di rilevamento dei guasti. È quindi necessario essere in grado di analizzare l'affidabilità e la robustezza di qualsiasi RCN per identificare le modalità più efficienti di irrobustimento di questi sistemi. Inoltre, è cruciale poter effettuare questo tipo di analisi negli stadi iniziali dello sviluppo per fornire feedback rapidi al processo di design. Da una review della letteratura emerge che gli stumenti di analisi attualmente disponibili presentano varie limitazioni, sia di tipo tecnologico che di tipo metodologico.

In questa tesi abbiamo selezionato un simulatore di errori per RNC esistente chiamato CLASSES con il goal di migliorarlo e adottarlo nello sviluppo di una metodologia di analisi dell'affidabilità. Abbiamo quindi ideato ed eseguito una campagna di simulazione di errori per dimostrarne la fattibilità e abbiamo usato i risultati per definire una strategia di irrobustimento per RNC efficiente.

Parole chiave: Affidabilità, Reti Neurali Convolutive, Processori Grafici, Deep Learning, Intelligenza Artificiale, Tolleranza ai Guasti

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

1 Introduction **1**

1.1 Goal . . . 3

1.2 Thesis Outline . . . 4

2 Background and related works **7**

2.1 Convolutional Neural Networks . . . 7

2.1.1 Neural Networks . . . 7

2.1.2 Convolutional Neural Networks . . . 8

2.1.3 Tensor . . . 9

2.1.4 Convolution . . . 9

2.1.5 Batch Normalization . . . 10

2.1.6 Activation functions . . . 11

2.1.7 Element-wise Operators . . . 14

2.2 Machine Learning Frameworks . . . 15

2.2.1 TensorFlow . . . 15

2.2.2 Caffe . . . 16

2.2.3 cuDNN . . . 17

2.3 Graphic Process Units - GPUs . . . 17

2.3.1 GPUs Hardware Architecture . . . 18

2.3.2 Parallel programming . . . 20

2.4 Faults in Digital Systems . . . 21

2.4.1 Soft errors . . . 21

2.5 Related Work . . . 22

2.5.1 GPU Fault Injectors . . . 23
2.5.2 SASSIFI . . . 27
2.5.3 NVBitFI . . . 28
2.5.4 Methodologies for Reliability Analysis . . . 30

3 CLASSES **37**
3.1 The Methodology . . . 37
3.1.1 Operators Extractor . . . 39
3.1.2 Operators Selection Phase . . . 39
3.1.3 Operators translation . . . 40
3.1.4 Architectural Fault Injection . . . 40
3.2 Error Model Definition . . . 42
3.2.1 Spatial Patterns . . . 44
3.2.2 Obtained Error Models . . . 47
3.2.3 Spatial patterns . . . 54
3.2.4 Error Simulation . . . 55
3.3 Framework Implementation . . . 56
3.4 Improving the Framework . . . 57

4 Case studies in reliability analysis **61**
4.1 Steering Angle Detection . . . 61
4.1.1 PilotNet . . . 62
4.1.2 Comma's AI Steering Wheel Model . . . 67
4.2 Image Classification . . . 71
4.2.1 CIFAR-10 Model . . . 71
4.2.2 Vgg11 . . . 79
4.3 Timing Overhead . . . 84

5 Hardening CNNs **85**
5.1 CNN Robustness Metric . . . 85
5.2 Usability aware hardening strategy . . . 87
5.2.1 Hardening the CNNs under analysis . . . 88
5.3 Results . . . 89
5.3.1 PilotNet . . . 89
5.3.2 Comma's AI Steering Wheel Model . . . 91
5.3.3 CIFAR-10 Recognition Model . . . 92
5.3.4 VGG 11 . . . 93
5.4 Conclusions . . . 94

6 Conclusions and future developments 97
6.1 Future developements 98
6.1.1 Error Simulation Framework 98
6.1.2 Hardening CNNs 99

Bibliography 101

List of Figures 109

List of Tables 113

List of Symbols 115

1 | Introduction

In the last decade the diffusion of digital systems increased dramatically in all aspects of our day to day life. Depending on their role and working scenario these products may assume a critical level for the success of the application in which they are employed. A crucial set of digital systems is the one of *safety-critical systems*, composed by those products that, in case of failure or malfunctioning, would cause injuries to the peoples that are around or working with them.

Autonomous Driving Systems (ADS) are one of the most challenging safety critical systems that saw an increase in usage in recent years [2]. An ADS can be defined as a set of high-level functionalities aimed at substituting partially or entirely the human driver. There are multiple functionalities that can be replaced such as recognizing the lanes of the current track or identifying pedestrians, bikes and other obstacles. An ADS usually exploits Machine Learning algorithms and Computer Vision to properly carry out its tasks. The Society of Automotive Engineers (SAE International) has proposed a six-level identification system to classify ADS functionalities [3]. It ranges from level 0, where the automated system is limited only to issuing warnings, to level 5, called "steering wheel optional" where no human intervention is required at all. It is clear that an ADS of level 5 is considered a safety-critical system and thus requires an high level of reliability. As with all kinds of digital systems, one of the most relevant problems is the possibility of becoming subject to physical faults. The occurrence of which could lead to failures and malfunctioning in the system itself.

Faults can have two different types of causes. The first ones are the internal causes, such as breakage of components or aging effects on parts of the system. These types of causes are usually an effect of the day-to-day usage of the system that slowly deteriorates the robustness of the components. Differently external causes are the effects that the environment has on the system. We identify in this group thermal stress, radiation damage, humidity, etc. Protecting against the former group of causes requires the designer to implement systems with periodical checks against wear and tear of core components. Instead, protecting against external causes requires a deep analysis of the conditions in which the system has to operate. Overall it is a complex and costly process.

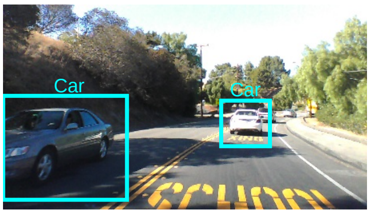

Figure 1.1: Correct labelling

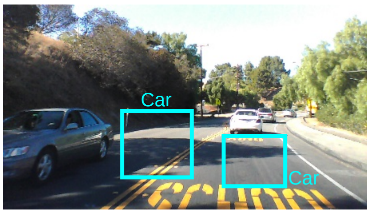

Figure 1.2: Wrong placement

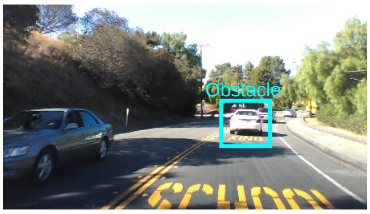

Figure 1.3: Wrong labelling

Digital systems have, therefore, a high probability of experiencing either permanent faults, such as destructive break-downs, or transient faults such as memory loss or corruption; the latter appearing with an higher frequency.

To understand the impact that faults might have on Autonomous Driving System we have to highlight the results obtained by the Boeing Defence and Space Group in a paper published in 1996 [4]. The authors observed that despite the lower frequency of appearance at ground level than in the aerospace domain, the number of faults that occurred is around two every thousand billion hour. While it might seem a very low number if we consider that in 2021 the number of cars travelling in Europe was almost 400 millions, it means a fault can be observed on any car every 3.5 hours on average. Joining this number with the knowledge of ADS being a safety-critical system, the request for a very high level of reliability takes a critical priority.

The set of sensors and functionalities that make up an ADS is ample, and as a consequence, not all faults have the same effects on the whole system. It is reasonable to presume that a fault striking the module responsible for detecting obstacles and surroundings has a stronger impact on the system's reliability. Being the entry point of the ADS means that an error on the perception module spreads among all other components resulting in an higher probability of critical outcomes. Let's discuss an example that better explains the criticality.

Figure 1.1 shows the situation in which an ADS correctly identifies two cars on the street with different light conditions, this case is optimal, and the system maneuvers the vehicle without mistakes. Figure 1.2 shows a possible effect of a soft error striking the perception module, the system is able to correctly identify two cars, but the placement is wrong. Finally figure 1.3 shows a different effect of a soft error where the system is incapable of recognizing the car in the shade and assigns the wrong label to the other car.

In both cases the results might be life-threatening. In case of misplaced labels the ADS could trigger an emergency brake procedure that could lead to a crash with vehicles coming from behind. Instead, in case of a missing label an overtake procedure might not consider the hidden vehicle. It is therefore clear that is necessary to analyze the reliability issues of safety-critical systems, with particular attention to the perception functionalities. Such analysis is performed through an error injection campaign that aims at evaluating the effects of faults on the system's behavior.

With the evolution of computer vision techniques for image detection, the state-of-the-art used in CV shifted from traditional processing pipelines to the usage of Convolutional Neural Networks (CNNs). CNNs are data-intensive computational models accelerated on Graphical Processing Units that perform image recognition tasks with a high level of precision. Combining the high performances of these models with the acceleration provided by GPUs is a winning partnership that, however, complicates the reliability assessment. It is necessary to perform this analysis on a specific system without the possibility of generalization.

Considering the high level of complexity of both the most recent ML models and the hardware accelerators used to speed their execution, it is common to execute the reliability analysis as the last step in the development phase. Waiting until the end of the development cycle ensures that the results obtained will be the most accurate. On the other hand, any possible upgrade identified during this analysis will require a substantial effort to introduce. In fact, we have to consider the time to develop the proposed update, integrate it into the existing model and test the whole system again.

It is, therefore, necessary to understand the reasons that push this crucial analysis so far ahead in the development phase. Apart from timing and cost constraints that are different for each developer, the main reason we identified resides in the available tools. The state-of-the-art reliability analysis tools, in fact, are either complex to introduce in the development workflow or time demanding. Both these factors discourage possible users from incorporating them into the design workflow.

For this reason, we deemed it necessary to introduce a methodology for the reliability analysis of ML models that is both fast and accurate and that can be incorporated as early as possible into the design flow of safety-critical systems.

1.1. Goal

The goals of this thesis are threefold:

1. **Improve CLASSES:** The first one is to improve the prototype implementation of the error simulator framework. To achieve it, we identified the framework's most critical components and designed solutions to fix them. The improvements were of two kinds, either technological advancements to adopt the state-of-the-art tools or

usability enhancements. Regarding the latter, we designed a more flexible interface and a comprehensive guide on integrating the framework into a workflow.

2. **Extensive error simulation campaign:** The second goal was to design and execute an extensive error simulation campaign. The objectives of this campaign were, in turn, twofold. Firstly to validate the framework's effectiveness and demonstrate that it is possible to perform an error simulation campaign of substantial size. Secondly, to assess the robustness against faults of multiple CNNs to verify that the results obtained with CLASSES are meaningful.

3. **Hardening of CNNs:** The third and final goal of the thesis was to define a hardening strategy for CNNs based on the information produced by CLASSES. We decided to base our methodology on the concept of Layer Vulnerability Factor and selective Duplication With Comparison. We demonstrated the feasibility of our approach against the same CNNs we targeted for goal 2.

1.2. Thesis Outline

The thesis is organized as follows:

- Chapter 2 presents the background knowledge required for the understanding of this work. It follows a review of the literature where we perform an analysis on the evolution of the GPU fault injectors and on the techniques that emerged in the last years.

- Chapter 3 presents the error simulation framework used. Here we deeply describe how it works, the technologies used to develop it, and how it was modified during this thesis. This chapter targets Goal 1, **Improve CLASSES**.

- Chapter 4 describes the results obtained during the error simulation campaign. Here we define the CNNs under analysis, presenting the motivations under which we selected them. Then a complete analysis of the produced results is performed. This chapter targets Goal 2, **Extensive error simulation campaign**.

- Chapter 5 describes the innovative metric proposed in this thesis. We provide an explanation on how it is build and then discuss the values obtained from the results produced in the previous chapter. This chapter targets Goal 3, **Hardening of CNNs**.

- Chapter 6 contains the conclusions obtained. We also present possible future works that can arise from the results produced and the problems encountered during this thesis.

2 | Background and related works

This chapter introduces the main concepts necessary to understand our work, we will first discuss about Convolutional Neural Networks, their components and the main algorithms involved. Thank we will present the concepts regarding the acceleration of ML models on GPUs and finally a discussion on related works will follow.

2.1. Convolutional Neural Networks

2.1.1. Neural Networks

Neural Networks, also know as Artificial Neural Networks, are a subset of machine learning, the field of artificial intelligence that aims at producing algorithms that can imitate the way humans learn, gradually improving accuracy over time through processes of trial and error. ANNs are formed by nodes that are connected together, like neurons, forming a graph that presents an input layer, one or more hidden layers, and an output layer. Each node can be expressed as a linear regression model where a set of inputs are weighted and combined with biases to produce an output, a formula that would look like this:

$$\sum_{i=1}^{m} w_i x_i + bias = output$$

Neural Networks can be used for many different tasks, from speech recognition to clustering of unknown data, and the types of models can be divided into two groups, those that require a training phase to perform their functions and those that do not require it. The former class of algorithms, usually composed by those models that, given an input, are able to derive some kind of information about it, relies on what is called *supervised learning*, a method in which the developer feeds the model with a set of inputs and specifies the correct information related to each input, the algorithm then tries to predict the correct answer and computes a *loss function*, a value that determines the correctness of the output, this is then used to update the weights in order to produce better results in the future.

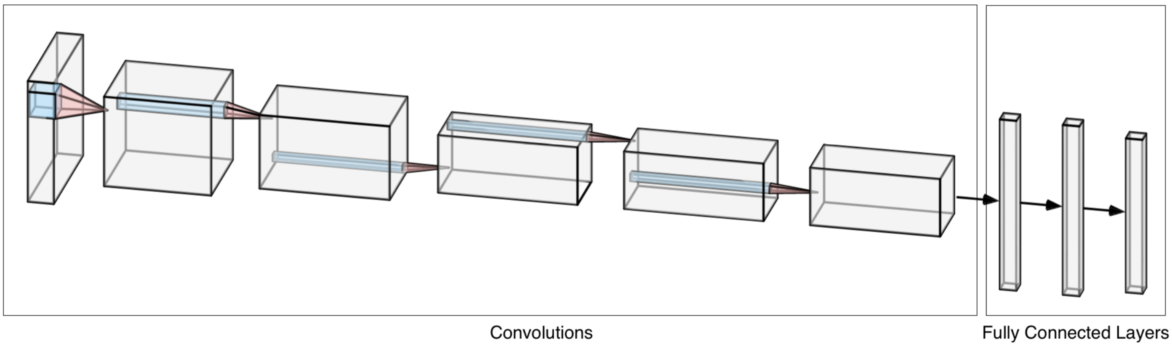

Figure 2.1: CNN example

2.1.2. Convolutional Neural Networks

Convolutional Neural Networks are a particular class of Neural Networks that presents superior performances with image, speech, or audio signals. They manage multidimensional data, known as **tensors**, and aim at deriving semantic representation from the input to accomplish a high-end task. They have three main types of layers:

- Convolutional layer
- Pooling layer
- Fully-connected layer

Each model is composed of various blocks of layers. Usually, there are multiple blocks composed of a convolutional layer, a pooling layer, and an activation function. These gradually reduce the dimensions of the data while extracting features, and then the model ends with a fully connected layer whose depth and final shape depend on the algorithm's task. Figure 2.1 shows the standard topology of a CNN, each layer has a different purpose, and convolutional ones are used to learn and extract features from the input by updating their weights accordingly to a specified loss function as explained in Section 2.1.1. Pooling layers help reduce the data size to increase the degree of generalization. **Activation functions** are mathematical functions applied element-wise to mimic the biological activation of neurons, fixing the output of each layer into a specific range of values. Moreover, for the purpose of shape reduction and normalization, networks might include **BatchNormalization** layers interleaved with convolutional layers.

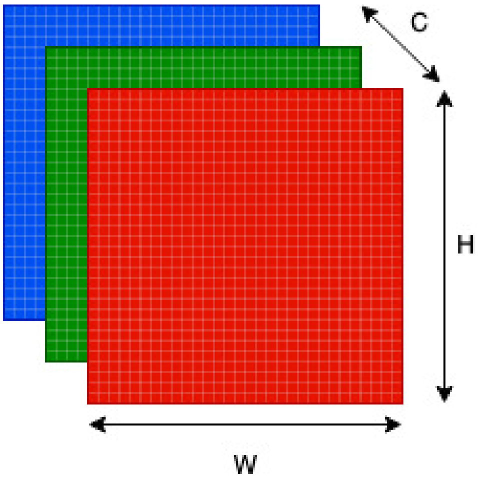

Figure 2.2: Representation of a tensor storing a RGB image

2.1.3. Tensor

A tensor is an array of numbers arranged on a regular grid with a variable number of axes. It is the basic unit of a CNN, forming the input and output of any operator or layer. One of the most common representation of a tensor is a RGB image, here we have a grid of $H \times W$ pixels, each pixel presents three different values, one for each color class. We can therefore describe one image as a stack of grids, this can be expressed as a tensor of shape $(C \times H \times W)$ where C is the number of channels, three in the case of a RGB image, H and W are the height and width of each grid, as seen in Figure 2.2. This representation is known as *channel-first* due to the ordering of the shape, the other possible representation is called *channel-last* and, as the name suggests, places the channels after the grid size. Both the representations are equivalent except for the low-level memory allocation layout but they can be used to interpret data depending on the scope.

2.1.4. Convolution

Convolution is a mathematical operation that correlates each input with its immediate neighbours and a set of weights in a linear combination, through the convolutional operation we are able to highlight some features of the input. Let's describe an example of this behavior, suppose we are tracking a moving car with a laser sensor, our laser provides a single output $x(t)$ which is the position of the car at time t. Unfortunately this value is noisy due to external conditions and we would like to average several measurements to obtain a cleaner value, moreover we value the recent measurements more so we will apply a weighted average that gives them more weight. Applying this weighted average operation at every moments yield us a new function $k(t)$ providing a smoothed estimate of the position of the car, this operation is the convolution.

$$k(t) = \int x(a)w(t-a)da \tag{2.1}$$

In ML literature weights are also called kernel or mask and the output is called feature map. What we defined in Equation 2.1 is the convolution on one dimension, time in our example, but as we explained in Section 2.1.3 CNN deal with multidimensional tensors, usually stack of two-dimensional grids, for this reason we should introduce a two-dimensional convolution:

$$K(i,j) = (I * W)(i,j) = \sum_m \sum_n I(m,n)W(i-m, j-n) \tag{2.2}$$

We should note that the convolution is commutative as seen in Equation 2.3 because we have flipped the weights relative to the input, in the sense that as m increases, the index into the input increases while the index into the weights decreases.

$$K(i,j) = (W * I)(i,j) = \sum_m \sum_n I(i-m, j-n)W(m,n) \tag{2.3}$$

Many ML frameworks implement a related function called **cross-correlation** that behaves exactly as a convolution without flipping the kernel, as in Equation 2.4

$$K(i,j) = (W * I)(i,j) = \sum_m \sum_n I(i+m, j+n)W(m,n) \tag{2.4}$$

Many implementations are available for the convolution, usually they are based on matrix multiplication with a rearrangement of the input into a linear vector thus projecting what originally was a two-dimensional space into a linearized one.

2.1.5. Batch Normalization

Training deep neural network with huge numbers of hidden layers can be challenging for multiple reasons, one being that the model is updated layer-by-layer from the output to the input under the assumption that the weights in the layers prior to the current one are fixed, but due to how layers are updated the distribution of each layer's input changes during training. The authors of the paper [5] that first introduced Batch Normalization refer to this change as *internal covariate shift* and they propose a technique capable of

standardizing the inputs to a layer for each mini-batch resulting in a stabilization of the learning process and a significant reduction in the number of epochs required to train a deep network.
The normalization consists of scaling the input data to have zero mean and unitary variance, the scaling is not performed globally over the whole input but it is applied grid-wise.
Given a mini-batch of input tensors $\mathcal{B} = \{\boldsymbol{x}_1 ... \boldsymbol{x}_m\}$ each with shape $(H \times W)$ we refer to the *i-th* channel of the batch as $\boldsymbol{x}_{(i)}$, then:

$$\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{2.5}$$

$$\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \tag{2.6}$$

We first calculate the mini-batch mean as seen in Equation 2.5, then the variance in Equation 2.6

$$\widehat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{2.7}$$

$$y_i = \gamma \widehat{x}_i + \beta \tag{2.8}$$

Lastly we normalize the input $\boldsymbol{x_i}$ by applying Equation 2.7 where ϵ represents a constant added for numerical stability. Simply normalizing each input of a layer may change what the layer can represents, for instance, normalizing the inputs of a sigmoid would constrain them to the linear regime of the non linearity. For this reason we apply Equation 2.8 to ensure that the transformation inserted in the network can represent the identity transformation, this is done through the insertion of two learnable parameters: γ and β.

2.1.6. Activation functions

An activation function is a mathematical function that is applied element-wise to a tensor and has the role of emulating biological activation of neurons [6]. There are multiple activation functions that can be used, mainly divisible into three families:

- **Ridge activation functions:** characterized by a linear combination of the input variables, when used in biologically inspired neural networks they represents the rate of action potential firing in the cell.
- **Radial activation functions:** functions whose value depends only on the distance between the input and some fixed point, they are used in RBF networks that are

extremely efficient universal function approximators.

- **Folding activation functions:** they perform aggregation over the inputs and for this reason are extensively used in pooling layers.

We will now describe the Rectified Unit and the Sigmoid function among the available ones, both belonging to the ridge family.

Rectified Units

Rectified units [7] [8] is a class of activation functions characterized by a common behavior, they are zero or almost zero for any negative input while being linear for any positive ones. The main advantages [9] of this class of activation functions are:

- Ease of computation resulting in a speedup of the execution.
- Representational sparsity meaning they can output a true zero value unlike tanh and sigmoid that learn to approximate such value. A behavior particularly useful in representational learning (e.g. with autoencoders).
- Linear behavior that makes the networks easier to optimize since it almost completely avoid the problem of vanishing gradients.

Multiple different implementations of the Rectified Units exists, Table 2.1 shows three of the most popular ones. ReLU is the classical implementation while Leaky ReLU and Parametric ReLU are relaxations of the function where negative values are not cropped to zero but instead a slow slope is allowed preventing the neuron from dying. The difference between Leaky ReLU and Parametric ReLU is that the slope is fixed for the former while the latter presents a trainable parameter that defines it.

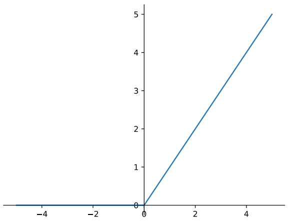

Figure 2.3: ReLU

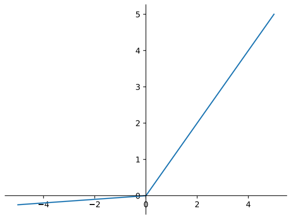

Figure 2.4: Leaky ReLU

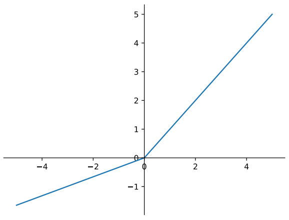

Figure 2.5: PReLU

Exponential Linear Unit

The exponential Linear Unit (ELU) is a an activation function that, in contrast to the standard ReLU, have negative values. While LReLUs and PReLUs have negative values

Name	**Formulation**	**Figure**
Rectified Linear Unit	$f(x) = max(0, x)$	Figure 2.3
Leaky ReLU	$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0 \end{cases}$	Figure 2.4
Parametric ReLU	$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{if } x < 0, \ \alpha \in \mathcal{R}^+ \end{cases}$	Figure 2.5

Table 2.1: Different ReLU implementations

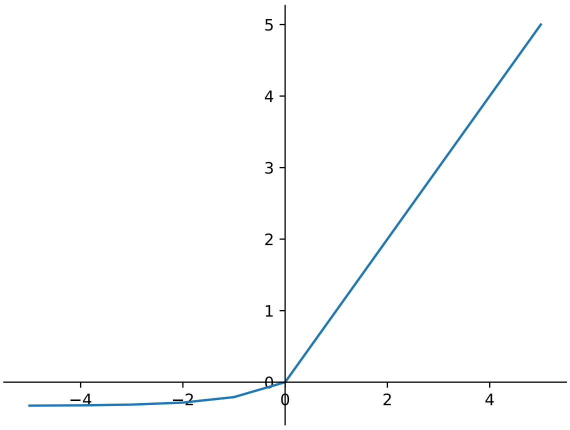

Figure 2.6: ELU

the ELU function ensures a noise-robust deactivation state, moreover it saturates to a negative value with smaller inputs and thus decrease the forward propagate variation and information.

The equation of the ELU is the following:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases} \tag{2.9}$$

Figure 2.6 shows a graphical representation of the ELU.

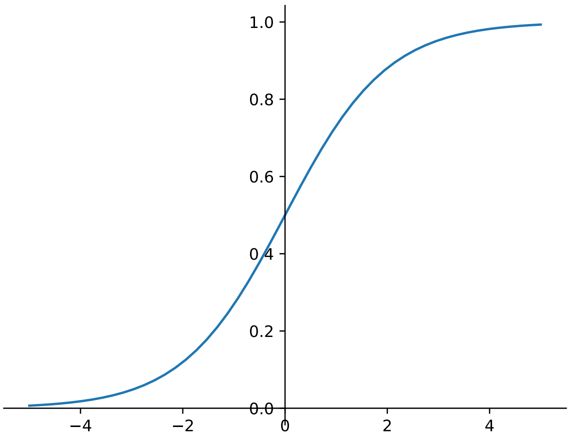

Figure 2.7: Sigmoid

Sigmoid

The logistic function, know as sigmoid activation function [10] in the field of neural network, is a monotonic real function defined as:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.10}$$

Its popularity is due to its smooth exponential transition between its boundaries which are usually 0 and 1, for this reason it is commonly used to represent probabilities. Figure 2.7 shows a graphical representation of the sigmoid.

2.1.7. Element-wise Operators

It is necessary to discuss a set of element-wise operators that are commonly found in CNNs as nodes or internally in the previously mentioned complex operators. They are mathematical transformations of tensors that do not require any particular implementation.

- Add: sums two tensors, element wise.
- BiasAdd: sums a set of scalar biases to each feature map of a tensor. Such biases are trained during the training phase.
- Mul: scales each feature map of a tensor to a set of scalar values.
- Div: divides a tensor by a constant dividend.

- Exp: applies element-wise to a tensor the exponential function.

2.2. Machine Learning Frameworks

Developing a Machine Learning software from scratch is a challenging activity due to the complexity of the algorithms involved and the need to develop code that can benefit from specific hardware accelerators such as GPUs, a task that requires a deep understanding of the underlying architecture. Due to this complexity many frameworks have been created to ease the developer during the design phase of a ML model. These frameworks provide a standard implementation of the most common operators needed, as seen in Section 2.1, while hiding the technical details, moreover they are usually developed to exploit different types of hardware accelerators, these details ensure that the developer can fully focus on the implementation details of the network.
Multiple frameworks are available to the public for free, among all of those we will now briefly present TensorFlow and, later in Section 2.2.2, since they have been employed in different phases of this work.

2.2.1. TensorFlow

TensorFlow [11] is a Machine Learning framework developed by Google, it allows developers to write and deploy ML models on a wide variety of systems, from CPUs to mobile devices to GPUs. TensorFlow, like many other ML frameworks, describes each model as a data-flow graph where nodes are CNN operators, the edges that connect each node are tensors containing the data that is currently used by the model.
In the first version of TensorFlow, which has been used in this work, there is a behavior not usually seen in other ML frameworks, in fact it decouples the definition of a model from its instantiation and execution. During the creation phase of a network the developer adds operators to the data-flow graph, at this time they are abstract computation called *operation*. Later, when the model is instantiated and executed, inside a session, TensorFlow automatically assigns the best available implementation of each operation to the nodes, these are called *kernel*, this method ensures that the underlying hardware is used as best as possible. This behavior has been hidden in TensorFlow2 where the concept of session, the allocated space for a model to be executed, has been replaced by eager execution by default, we should note nonetheless that this change does not invalidate the our methodology since its still possible to access the data-flow graph in TensorFlow2 thanks to a set of dedicated APIs. Two more components of Tensorflow should be discussed since they are fundamental in the following discussion about the error simulator:

- Control dependency: it is a special edge that can be introduced in the control-flow graph between two operations. It enforces the order in which they are executed, so if operation X is connected to operation Y with a control dependency edge then X will be always executed before Y.
- Variable: a special *operation* that references a mutable tensor that persists among different executions of the data flow graph, a behavior that other tensors do not possess. Variables are usually used to store weights and biases of operations that need to persists during all the executions.

The popularity of TensorFlow is also due to the availability of high-level APIs such as Keras [12] which presents an high degree of abstraction allowing the developer to create complex CNN models with a relatively small amount of code.

2.2.2. Caffe

Caffe [13] is a deep learning framework developed by Berkley AI Research that provides a complete toolkit for training, testing and deploying ML models. The framework has multiple strengths among which:

- **Modularity:** Multiple layers and loss functions are already implemented but the framework itself is designed to be as modular as possible allowing each developer to introduce new layers or extensions to new data formats.
- **Separation of representation and implementation:** Model definitions are written as config files using the Protocol Buffer language, upon instantiation Caffe reserves the least amount of memory needed.
- **Python and MATLAB bindings** Caffe provides bindings to both Python and MATLAB, two of the most common languages for building and deploying ML models.

Caffe models are stored as directed acyclic graphs composed of two types of structures:

- **Blobs** Blobs are 4 dimensional arrays used by the framework to store batches of images, parameters or parameters update. They hide the computational overhead of mixed CPU/GPU operations by synchronizing as needed. In practice the developer only has to upload data into a blob in CPU code, then a CUDA kernel is called to do the calculation and the result can be fetched directly from the blob itself ignoring low-level details.
- **Layers** Layers are the operations that constitute a network, they take one or more

blobs as input and produce one or more blobs as outputs. As any other ML frameworks layers present two key operations, one *forward pass* that produces the outputs given a set of inputs and one *backward pass* that takes the gradient with respect to the output and computes the gradients with respect to the parameters and to the inputs, which are then back-propagated to earlier layers.

2.2.3. cuDNN

cuDNN is the GPU-accelerated library of primitives for deep neural network developed by NVIDIA, it provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization and activation layers.

Multiple widely used deep learning frameworks rely on cuDNN for GPU accelerations, examples include Caffe2, Keras, MATLAB, PyTorch and Tensorflow. With respect to the stack of softwares involved in the ML deployment cuDNN is the layer that connects high-level frameworks to bare metal operations directly on GPU. Particular interests has to be placed on the implementation of the convolution by cuDNN, as described in [14] the goal of the framework is to provide performance as close as possible to matrix multiplication while using no auxiliary memory, multiple ways of doing so are possible, from Fast Fourier Transformation as described in [15] to lowering the operation into a matrix multiplication as described in [16].

cuDNN provides the developer eight different convolution algorithms, moreover the developer has also access to a routine that, given the parameters of the inputs, returns the supposedly fastest algorithm or the one that uses the least amount of memory.

2.3. Graphic Process Units - GPUs

Graphic Process Units are specially designed electronic circuits with the ability to rapidly manipulate and alter memory to accelerate the rendering of 3D scenes and images.

This acceleration is possible due to the extreme regularity in all the processing steps within the graphics pipeline, each operation needed is applied mostly independently to each pixel or element without needing a complex control logic, these operations can be therefore executed in parallel threads that are free of race conditions.

All these conditions initially led to the development of GPUs, circuits that, unlike general purpose CPUs, where designed for a set of very specific tasks, a behavior that has been slowly outdated in recent years where GPUs have seen a rise in usage in High Performance Computing tasks that exhibit a high level of data parallelism. These tasks include General Matrix Multiplications (GEMM) [17] which represents the fundamental block of many

ML operators, i.e. convolution as we have seen in Section 2.1.4, and Basic Linear Algebra (BLAS) operations that can be applied to n-dimensional matrices.

2.3.1. GPUs Hardware Architecture

Each GPU is composed by multiple Processor Clusters (PC), each PC contains an array of Streaming Multiprocessors (SM) which are Single Instruction Multiple Data (SIMD) multi-core processors that supports parallelism by simultaneously executing the same task over multiple memory locations.

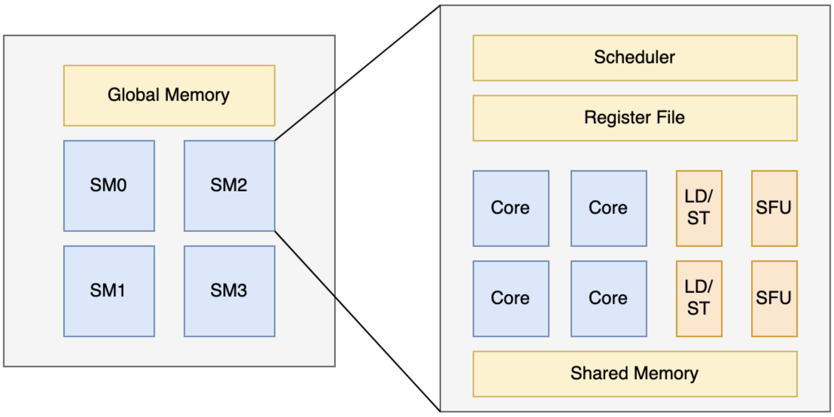

Figure 2.8: SM architecture

The internal structure of each SM is shown in Figure 2.8 and it consists of:

- Scheduler: the unit responsible of fetching and decoding each operation.
- Register File: the unit that contains the registers used by each thread. The maximum amount of registers per thread is 255, instead the maximum amount of 32-bit registers per SM is 32K or 64K depending on the architecture.
- Execution Units: each streaming multiprocessor contains hundreds of execution units that are responsible for the execution of the loaded instruction. There are three main types of execution units:
  - Load/Store: execution units responsible for the interactions with the memory.
  - Alu: arithmetic logic units, responsible for the management of 32-bit floating point or 64-bit integers values.
  - SFU: special function units that execute functions such as cosine, sine or square root.

Each SM has the same workflow, it begins with the fetch and decode of each instruction by the scheduler unit. Then N execution units perform the required computation in parallel on N different chunks of data. This execution in parallel is an enhanced version of the previously discussed SIMD paradigm called Single Instruction Multiple Thread (SIMT). However the scheduler cannot manage an arbitrary number of threads, for this reason threads are partitioned into groups called *warps* of fixed size, in particular NVIDIA architecture has a warp size of 32 threads. Since the scheduler unit is able to deliver multiple instruction streams in time-multiplexing the SM is able to support task-parallelism and also it is possible to have interleaved execution of multiple warps thus maximizing the throughput while hiding the latencies of slow operations such as those interacting with the memory.

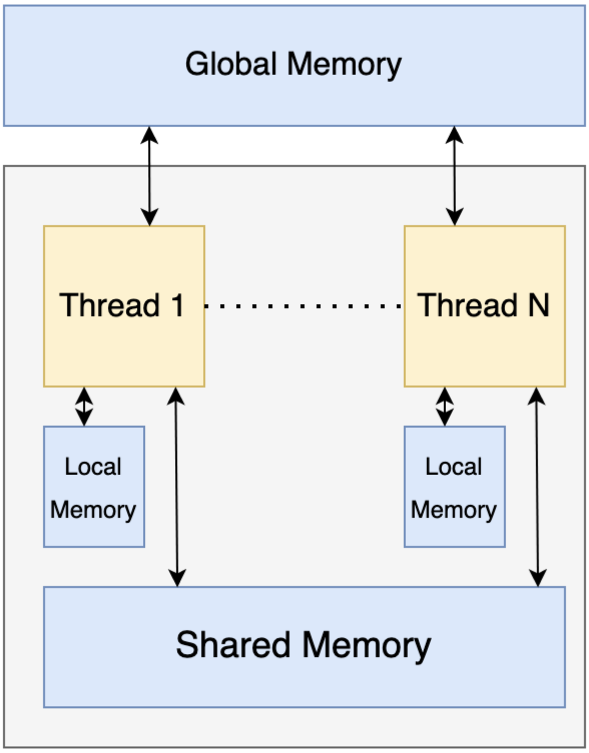

Figure 2.9: Thread Memory

Any developer that works with a GPU should explicitly keep track of the complex memory hierarchy exposed by these circuits, as seen in Figure 2.9. There are four different levels of memory:

- Register File: this is the fastest memory available, located directly onto the SM. Usually holds local variables for each thread.

- Local Memory: another private memory for each thread, it too is located onto the SM and holds local variables that do not fit into the register files.
- Shared Memory: this is a user defined cache intended to store data that has to be shared among threads in the same SM to obtain better performances.
- Global Memory: this is the largest and slowest memory available, it is located off chip and holds all the data that is not immediately required by any thread.

2.3.2. Parallel programming

Writing a program that can be accelerated on a GPU requires different precautions than a classical sequential programming workflow, the NVIDIA CUDA Framework for parallel programming defines *kernel* the basic function that can be accelerated. Each kernel is a standard function that is automatically parallelized by the GPU assigning it a large set of threads, each identified by a unique index and organized into a matrix directly mapping the input/output data that has to be processed. This matrix could be partitioned in sub-grids called blocks that are all executed on the same SM, for this reason block's threads use the same shared memory to exchange data.

```
void cpu_sum_vectors(int* vector1, int* vector2) {
    for (int idx = 0; idx < vector1.size(); idx++) {
        vector1[idx] += vector2[idx];
    }
}
```

Listing 2.1: Sequential sum of two vectors

```
__kernel__ void gpu_sum_vectors(int* vector1, int* vector2) {
    int idx = get_thread_index ();
    vector1[idx] += vector2[idx];

}
```

Listing 2.2: Parallel sum of two vectors

Code 2.1 shows the classical sequential approach at summing two vectors, here an index is gradually incremented allowing the developer to slide across the entire vector and retrieve the correct data. In a parallel framework, as seen in Code 2.2 we exploit the fact that the number of thread generated is specified at the time of the kernel invocation, for the reason we can assume that each thread will be responsible of one single value thus transforming a loop sequence into a perfectly parallelized operation.

2.4. Faults in Digital Systems

Resilience and reliability are two fundamental properties required by any safety-critical or mission-critical systems like Autonomous Driving Systems or Image recognition in satellite missions. For this reason it is critical to understand what is considered a fault in these systems and study their occurrence and the consequences of their presence.

A fault is defined as an irregularity in a circuit that causes a deviation of the system from its nominal behavior. In literature [18] faults are classified based on their duration and persistence in the following classes:

- Permanent faults: these are irreversible physical changes in a circuit.
- Intermittent faults: these are faults that are caused by hardware instability arising from variation in working conditions such as humidity, chip temperature or supply voltage. Usually they do not compromise permanently the functionalities of the systems but signal that a permanent fault is likely to happen.
- Transient faults: these are temporary and reversible faults caused by environmental conditions such as electromagnetic fields, power supply and radiations.

Both permanent and intermittent faults are usually caused by problems in chip design that are initially ignored by designers and then fixed after particular working conditions highlight their presence. Unlike them transient faults are not easily reduced by manufacturers since their triggers are environmental conditions that are unpredictable during the design phase.

Transient faults lead to glitches in the circuits called soft errors, when these occurs the main effects observed are bit-flips in memory cells, i.e. values that change from 0 to 1 or vice-versa, thus corrupting the processed data. Any system under these type of error can be restored by resetting it or rewriting the memory corrupted, a process that might be time or resource consuming.

2.4.1. Soft errors

Soft errors have been observed in digital systems since the 1970s, particularly in spaceborn electronics that were subject to cosmic radiations as described in [19] but also at ground level [20], both these papers describe a situation where some bits had randomly changed due to cosmic rays or alpha particles without damaging the memory.

With the increased complexity and diffusion of circuits transient faults became one the most relevant failure phenomena in modern digital systems, as explored in [21] and [22], to better understand this increased relevance we should observe the difference between two top-classes GPUs produced by NVIDIA in the span of 5 years, the NVIDIA GTX Titan [23], produced in 2013, has 7.08 billion of transistors and a process size of 28 nm. Instead, 8 years later, the newest NVIDIA GeForce RTX 3080 [24] possesses 28.3 billion transistors with a process size of only 8nm, so despite a 300% increase in the number of transistors the process size has been decreased more than 3x. These numbers clearly show that GPU-based systems must accurately take into account soft errors and developers should devote enough time at testing against presence of transient faults, a process to which NVIDIA already devotes a large effort to allow the employment of their GPU in safety critical digital systems [25] [26]. The effects of a soft error on a system can be classified based on the difference between the expected output and the one produced, the four main classes are:

- Masked: the output of the application is equal to the golden reference, this is due to the placement of the soft error in a portion of the data that is either not used or absorbed by the system (i.e. a maxpool operation with a greater value in the same cell).
- Silent Data Corruption (SDC): the application correctly ends its execution but the output differs from the expected one.
- Application timeout: the application is stuck in an unfinished state without any possible recovery.
- Application Crash: the application suddenly terminates.

From a reliability point of view the last two types of effects are trivial to detect and do not presents a major hazard, moreover creating a solution is straightforward and does not require any type of knowledge of the type of data used by the application. This is not true regarding Silent Data Corruptions, since the application correctly completes its execution the developer does not have the possibility to identify the presence of a fault apart from building an ad-hoc solution for the system under analysis. For this reason SDCs represent a critical issue in digital systems and this work will focus on analyzing their effects on GPUs.

2.5. Related Work

The following part of the chapter will review the literature related to the reliability analysis of GPUs applications, first we will discuss the existing fault injection frameworks then we will presents the most interesting works that use them to analyze the effects of faults in CNNs.

2.5.1. GPU Fault Injectors

Historically, one of the core techniques for performing reliability analysis in digital systems has been fault injection. This technique consists of inserting one or more erroneous values in the system's data flow and observing the effects. As with other techniques on digital systems, fault injection can be performed either on a real system or on a software modeling of it. Each approach has pros and cons that we will highlight in the following discussion. Fault injectors, while different from each other, present the same execution flow that begins by identifying the injection sites that are the places where the faults will be injected. This step is called the profiling phase and it is followed by the generation of a list of faults to inject that are usually tuples composed of an injection site and a value.
The last two steps consist of selecting one or more faults, randomly or not, executing the system after the fault has been injected, and then analyzing the context-dependent output.

CUDA-GDB

CUDA-GDB [27] is the NVIDIA extension to GDB, the GNU Project Debugger, developed for debugging CUDA applications running on Linux. Although it has not been designed for injection purposes it possesses features that enable this mechanism such as the ability to freeze the execution of a program at any given point, change the value of any variable and resume the execution.
Before executing a fault injection campaign the profiling phase requires that an ad-hoc routine extracts the list of kernels and all their local variables. The three aforementioned features allow a developer to to easily create a fault injection tool using CUDA-GDB, unfortunately using a debugger requires the code to be compiled with debug symbols and the overall execution time of the process is drastically increased.

GPU-Qin

GPU-Qin [28] is a fault injector build using CUDA-GDB from which inherited pros and cons. The main difference resides in the grouping phase, the idea is that threads within a kernel might not execute the same amount of instructions causing a divergence in the timing. Under this assumption GPU-Qin groups threads accordingly to the divergence factor, then a thread for each group is selected and executed in a simulator, GPU-Sim, that is able to extract a complete trace of execution. Then, the profiler analyzes all the traces and tries to map each instruction to their corresponding source-code line. Then the injection is performed, GPU-Qin is only able to target the outputs of instructions, the Register File and the source operands of Load/Store instructions. The main drawback of this tool is the grouping phase, while it allows for a more precise tracking of instructions it requires step-by-step analysis by the debugger and thus it is highly time consuming.

LLFI-GPU

LLFI-GPU [29] is a GPU fault injector build upon the open-source CPU LLFI fault injector, it uses the LLVM compiler to complete its operations. The instrumentation phase is performed by intercepting any call performed by the CUDA compiler to LLVM, this allows the tool to extract the number of kernel calls, threads per kernel and the instructions executed. After this phase the code necessary for the injection phase is inserted and the project under analysis is compiled, this returns the Intermediate Representation (IR) which is then passed to the CUDA compiler to be converted into SASS assembly. The injection phase follows the standard workflow already described.
While this tool is faster than other frameworks that rely on debugging tools it sill suffers from the need to recompile the whole code-base and the overall options available for the injection phase are less than what SASSIFI offers.

CAROL-FI

CAROL-FI [30] is a fault injector tool built upon GDB with Python support. It has been tested on programs running on an Intel Xeon Phi and despite targeting a different architecture than our research this project provides useful insights on how compiler-based fault injectors are built. The workflow of CAROL-FI consists of 5 steps:

- The code under analysis is executed under GDB.
- At random time an interrupt is sent to the program, GDB triggers a script that examines the memory content.
- The script randomly chooses a variable from the stack of the program.
- Accordingly to the chosen fault model the value of the variable is changed.
- GDB resumes the execution of the program.

While the tool is capable of detecting the high level sources of harmful effects it suffers from the need of using debug symbols and thus the time overhead produced is not negligible.

TensorFI

TensorFI [31], regardless of the name, is a high level error simulator for TensorFlow. This tool differs from the previously discussed fault injectors since it targets ML models at an higher level without requiring any information on the underlying architecture. This abstraction ensures that the heterogeneity caused by the huge amount of low level algorithms used and the different architectures over which a ML model could be executed does not represent an insuperable limit in the first phases of the deployment.
Otherwise during the first phase of development it would be impossible to evaluate the resilience of a ML application due to implementation details or complexity.
TensorFI has been developed following three main goals [32]:

- Ease of Use and Compatibility: the injector should be as transparent as possible for both the developer and TensorFlow.
- Portability: TensorFI should be attachable to any TensorFlow model without requiring to modify the model itself, TensorFlow nor the framework.
- Speed of Execution: using TensorFI should not interfere drastically with the normal execution of the model, either by not interfering at all when faults are not injected or by keeping any delay reasonably small. Further it should not render the main graph incapable of being executed on GPUs or make it nonparallelizable due to its modifications.

The granularity of the injector is the operator of the TensorFlow's graph and the execution flow of the tool is divided into two parts. The first one is the *instrumentation phase*, here the original graph is replicated operator-by-operator while inserting the necessary code to inject faults. This process is required for two reasons, first it ensures that the original graph is not modified secondly due to how TensorFlow 1.x works it is not possible to modify a graph that has been already built, making it necessary to duplicate it.
The second phase is the *execution phase*, the first step performed extracts a set of information from a setting file that has to be modified by the user, then the faulty graph is executed and the selected errors are injected, the final output of this phase is the model's output. The setting file is composed by the following fields:

- Fault type: the type of faults that will be inserted, it must be specified both for scalars and for tensors. The possible fault types are:
  - None.
  - Random Value.

    - Zero Value.
    - Random Element.
    - Bit_Flip.

- Operations and Probability: a list of operations where faults can be injected, each operation has a probability that represents the probability that the fault will be injected in that particular operation.
- Random Seed.
- Skip count: the number of operations to skip at the beginning of the program before beginning the fault injection.
- Instances: the number of instances of each operation in the model.
- Inject Modes: the three available inject modes are:
    - Error Rate: injects based on the probability specified in the config file.
    - Dynamic Instance: injects each operation type in the program once.
    - One Fault per Run: performs one random injection per run.

Despite being highly configurable the tool shows a critical design flaw regarding the fault models available, while being valid at the architectural level there is no validation presented by the developers that supports their legitimacy at the functional level. Such validation is fundamental to justify the usage of a tool in a real-world environment.

Moreover TensorFI presents several development issues, certain operators are replicated with their NumPy's implementation, an approach that results in two different implementation, TensorFlow's and NumPy's one, of the same algorithms that could lead to differences in the outputs especially when working with floating point numbers. Moreover when the selected injection mode is either Dynamic Instance or One Fault per Run the developer is required to insert the number of operators in the model, this is a process both time consuming and prone to errors since many ML models contains hundreds if not thousands operators. This requirement defies the set of design principles defined by the developers themselves. Finally the replication of the graph is performed opening a new TensorFlow's session each time, this is a time consuming operation that is even more emphasized when the operator is accelerated on a GPU since it requires all the memory to be transferred from CPU to GPU. Nonetheless the performances of the tool are superior than many fault injectors, TensorFI is consistently faster than any architectural fault injector, even SASSIFI, since it belongs to the application domain.

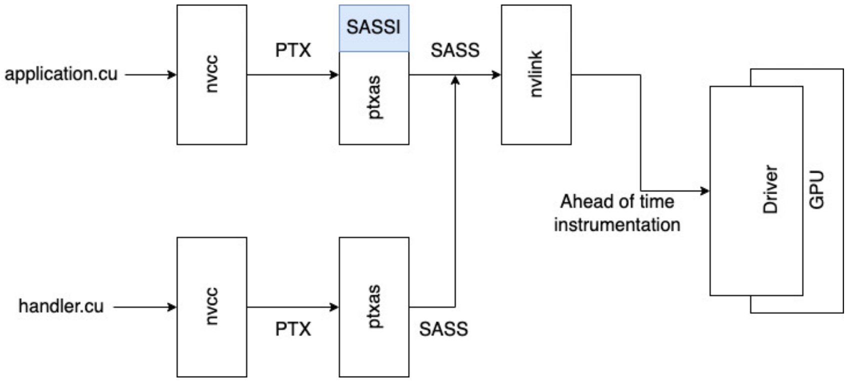

Figure 2.10: SASSI instrumentation flow

2.5.2. SASSIFI

SASSIFI (SASSI-based Fault Injector) [33] as the name suggests is a framework built with SASSI that provides an automated flow to perform an error injection campaign. SASSI [34] is an assembly-language instrumentation tool for GPUs that works at the SASS level, developed by NVIDIA. Figure 2.10 shows the compiling flow that includes SASSI instrumentation process.

The workflow of SASSIFI can be divided into three parts:

1. Profiling and identifying the error injection space.
2. Statistically selecting error injection sites.
3. Injecting errors into executing applications and monitoring error behavior.

SASSIFI proposes the different injection modes:

1. RF Mode: selects a random instruction from a random thread and inserts a fault into one of the allocated registers for that thread.
2. Instruction Output Value Mode (IOV): selects a random instruction from those that write to the RF within a thread and inserts a fault in the destination register **after** its execution.
3. Instruction Output Address Mode (IOA): selects a random instruction from those that write to the RF or memory within a thread and inserts a fault in the destination register or memory address **before** its execution.

and four different error models as explained in Section 3.1.4.

The developer has to specify which class of injection sites to produce, these can be done by selecting one injection mode, one error model and one class of instructions to target.

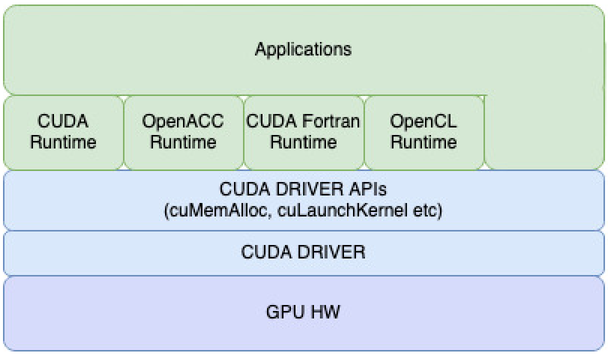

Figure 2.11: CUDA Software Stack

After this decision is taken the tool produces a list of injection sites that is a list of tuples specifying the kernel, the instruction opcode and the value to be inserted.
At the time of the development of the first version of this framework SASSIFI was the state of the art fault injector, fast and with the broader scope, unfortunately it still suffers from several limitations.
As explained in Section **??** each tool built with SASSI has to be compiled into the target program, this means that changing the injection mode requires compiling the code from scratch. This process is time-consuming for large projects and entirely impossible for closed-source libraries like cuBLAS making them not injectable. Moreover SASSIFI works only with CUDA 7 which is now obsolete.

2.5.3. NVBitFI

NVBitFI [35] is the state of the art fault injector that targets programs running on NVIDIA GPUs built using NVBit [36]. Figure 2.11 shows the software stack of a CUDA environment, programs running on NVIDIA GPUs interact with GPU drivers through a series of well defined CUDA driver APIs, these can be accessed by run-times such as OpenCL, CUDA-Fortran, OpenACC or CUDA or directly by the application.
NVBit interfaces directly with the CUDA Driver thus seamlessly interposing between the software stacks that exist above it, the compiling and usage of an NVBit tool is shown in Figure 2.12 The main advantages that NVBitFI offers are:

- Binary instrumentation: working at SASS level and using NVBit's tools NVBitFI is capable to perform instrumentation without the need for source code, a feat that was not available with SASSIFI.
- Dynamic instrumentation: NVBitFI intercepts dynamic GPU kernel calls, this property allows it to target dynamically loaded libraries, even those that are not known

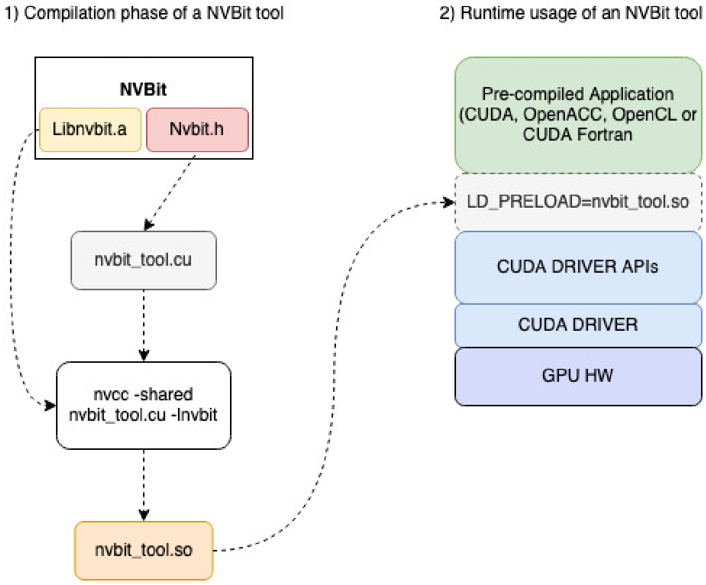
1) Compilation phase of a NVBit tool
2) Runtime usage of an NVBit tool
NVBit
Libnvbit.a
Nvbit.h
nvbit_tool.cu
nvcc -shared nvbit_tool.cu -lnvbit
nvbit_tool.so
Pre-compiled Application (CUDA, OpenACC, OpenCL or CUDA Fortran
LD_PRELOAD=nvbit_tool.so
CUDA DRIVER APIs
CUDA DRIVER
GPU HW

Figure 2.12: NVBit tool compilation and usage

at build time. Moreover the instrumentation is limited to that needed for the fault injection process keeping the performance overhead limited.

- Architectural abstraction: NVBitFI presents a single interface that works on all recent NVIDIA architecture families.

As of the date of writing this thesis the tool is composed of a profiler and an injector, both are implemented as dynamic libraries that are attached to a target program at run-time. The profiler is the first tool invoked in the injection process, it creates a profile containing one line for every dynamic kernel and the total dynamic instruction count for every opcode in every thread in that dynamic kernel.
Starting from this profile a dynamic instruction will be chosen randomly, a tuple of shape $<kernel\_name, kernel\_count, instruction\_count>$ is generated to instruct the transient fault injector to inject an error for that specific dynamic instruction. The injection is carried on by the injector library, when the selected dynamic instruction is reached the destination register is corrupted based on the chosen bit-pattern value.
It should be noted that a complete profiling of the program might be time consuming, for this reason NVBitFI provides two different types of profiling, exact and approximate. The former counts every dynamic instruction while the latter counts only the dynamic instructions in the first instance of every static kernel and assumes that the instruction count for subsequent instances of the same static kernel are the same. For our framework we decided to use exact profiling since the injected programs are relatively small and time is not an issue unlike accuracy. Determining if the execution of a program is successful

depends on user conditions, for this reason NVBitFI requires a SDC checking script to be provided by the developer, this script should reference a saved golden reference, and any other file or output produced during each execution, the possible outcomes of NVBitFI injections are shown in Table 2.2

Outcome	Symptom
SDC	Standard output is different
	Output file 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

Table 2.2: NVBitFI injection's possible errors

2.5.4. Methodologies for Reliability Analysis

The goal of reliability analysis of ML applications is to identify the operators that, if subject to faults are more likely to cause the output of the system to deviate from the nominal one. Usually, the outcome of this analysis is a statistical model that associates each operator or layer, depending on the granularity of the analysis, with its probability of causing a deviation from the expected behaviour. We should note that the term deviation used in this context might refer to a custom error class defined by the developer depending on the task performed by the model under analysis.

The next section will describe the current state-of-the-art reliability analysis of Convolutional Neural Networks accelerated on GPUs.

Error propagation techniques

Error propagation techniques are a set of reliability methodologies where the goal is to study the error propagation among kernels and instructions to find how much a fault on a given portion of code will influence the output of the entire program. Here we will describe two works that use this technique, one directly targeting ML applications while the other targets the broader field of GPU applications.

BinFI BinFI [37] is a fault injector based on TensorFI with the goal of finding the safety-critical bits in ML applications. This is possible since many widely-used ML operators are *monotonic* and as such it is possible to approximate the error propagation behavior of a ML application.

The idea of error propagation in a ML application can be explained with an example, suppose we are computing the KNN-Neighbors algorithm as seen in Code 2.3

```
negImage = tf.negative(testImg)
relativeDistance = tf.add(neighbors, negImage)
absDistance = tf.abs(relativeDistance)
distance = tf.reduce_sum(absDistance, axis=1)
nearestNeighbor = tf.arg_min(distance)
```

Listing 2.3: KNN-Neighbors

The input image is a matrix of dimension (28,28) and the output of the add operator is a matrix of dimension (|N|, 784) where |N| is the number of neighbors and one vector of size 784 is associated with each neighbor. If the fault occurs during the computation of the add operator we will have a faulty relativeDistance, if the i_{th} image is the nearest neighbor then we have $distance_i < distance_j, \forall j \in |N|, i \neq j$.

If the fault occurs at $(i, y), y \in [1, 784]$ (i.e. the vector corresponding to the distance relative to the nearest neighbor) then it might lead to a SDC if the modified bit produces an increase in the calculated $distance_i$, otherwise if the bit can't change the distance value into a greater one or if the fault occurs at $(j, y), y \in [1, 784], i \neq j$ then the fault will not be propagated and it will not results in an SDC.

This leads to the the idea of *SDC-boundary bits* which are bits where faults at higher order would cause an SDC while faults at lower orders would not cause any SDC. The workflow of the tool is the following: for each operator in the model the output is converted into a linear vector, then a fault is placed in the middle of the vector and the model is executed, if this causes an SDC then the next iteration will insert a fault in a lower order bit, otherwise the fault will be inserted in a higher order bit. This binary search is executed until possible, at the end of the process the tool returns for each operator the index of the boundary bit.

While this approach shows an high level of accuracy in finding boundary bits it suffers from the same problems as TensorFI, the error models used are not representative of real world errors. Nonetheless it provides the developers with a unique insight in how a model responds to soft errors in the early phases of developments.

GPU-TRIDENT GPU-TRIDENT [38] is a scalable and accurate technique for modelling error propagation in GPU programs, it extends the already existing TRIDENT [39] to support error propagation among GPU programs.

In this study, the authors define two heuristics aimed at selecting a subset of threads in a kernel to identify the existing memory dependencies. These metrics are necessary since GPU programs have a much higher count of threads executed, making error propagation studies much more time-consuming and error-prone.

The first metric is called H-Intra. It profiles the Intra-Thread Memory dependency. Its goal is to build the memory dependency tree of a given kernel without profiling all memory accesses. It performs this task by grouping threads with identical execution paths. Those threads will then have the same static dependencies between loads and stores. To further reduce the number of threads to analyze, a second selection is applied where only those with unique loops are selected.

To capture Inter-Thread memory dependencies, the authors designed a second heuristic. It consists of selecting the thread blocks to which each intra-dep threads belong and then profiling the shared memory access in each thread block. After those two selections are executed, the number of threads to profile is strongly reduced, and the tool is capable of creating the memory dependency graph and modeling the error propagation inside the program.

The framework is build as a set of LLVM compiler passes and requires the source code to work. This technique, despite being deemed accurate by the authors, is not based on real-world observations, and thus we consider it not robust enough for using it in safety-critical systems.

Revealing GPUs Vulnerabilities by Combining Register-Transfer and Software-Level Fault Injection

This paper [40] starts from the assumption that a low level fault injection campaign on a GPU model requires, due to the high complexity of the hardware, a huge amount of time while a purely software fault injection, despite being quicker, cannot access critical resources for GPUs and usually uses error models that are simplifications of real world errors.

For this reasons the authors propose a technique that combines the accuracy of Register Transfer Level fault injection with the speed of software fault injection, this is performed in two distinct steps.

The first step consists in performing a RTL fault injection campaign, they use an open-source GPU VHDL-based model [41] which implements the NVIDIA G80 architecture, then through a custom RT-level framework [42] they inject one fault into the model. From this injection campaign they generate a report that contains information about the result of the injection, the location of the injected fault, the number of affected threads, the

spatial distribution of erroneous values in the warp output and more.
In the second step of the technique the authors use a modified version of NVBitFI, capable of inserting errors accordingly to the reports produced in the first step, to inject faults into the warps scheduler, the pipeline registers, functional units and floating point units. This approach results in a considerable reduction of the time required for an accurate injection campaign while preserving an high level of accuracy.

Kernel and layer vulnerability factor to evaluate object detection reliability in GPUs

The authors of this paper [43] propose the introduction of two new factors to evaluate the reliability of object detection applications on GPUs. Kernel Vulnerability Factor (KVF) and Layer Vulnerability Factor (LVF) that identify the probability of a kernel or layer to affect the output of the application if a fault occurs. Using these two factors the authors propose an hardening technique that requires to replicate only the layers or kernels with the highest values of vulnerability indices.
KVF is evaluated using fault injections at both the architectural level and the application level using SASSIFI and CUDA-GDB, the metric is assessed for the Histogram of Oriented Gradients (HOG) algorithm. LVF instead is evaluated on YOLO using SASSIFI to perform an architectural fault injection campaign.
On CUDA-GDB the only error mode used is the random value one, while during the architectural injection campaign the authors used bitflip model with two modes: Register File and Instruction Output Value. Based on the analysis of the two metrics produced they propose a Double Module Redundancy strategy that is able to detect up to 80% of SDCs.
This approach is similar to the one we propose. Nonetheless, we believe it to be too vague and limited by the usage of CUDA-GDB instead of a better tool like NVBitFI.

Understanding Error Propagation in Deep Learning Neural Network Accelerators and Applications

The goal of the work in [44] is to experimentally evaluate the resilience characteristics of DNN systems and propose solutions to mitigate the effects of faults. Four popular convolutional neural networks are targeted by the authors: ConvNet, AlexNet, CaffeNet and NiN, all these models are executed on a DNN simulator, Tiny-CNN [45], customized in order to perform fault injection.
The evaluation of the effects of faults classifies them accordingly to the network topology,

data types, layer positions and types. Through this classification the authors are able to highlight that a corruption of the exponent bits of a floating-point value are more likely to introduce an SDC on the system rather than a corruption on the bits belonging to the mantissa and sign. Moreover bitflips are not symmetrical in their effects, a change of value from 0 to 1 has an higher probability to cause an SDC rather than the opposite transition.

The work identifies the Local Response Layer, introduced by AlexNet's authors, as a valid resource in normalizing values, a process that can mitigate the effects of large deviations in the values' domain.

Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs

This work [46] is particularly interesting since, in the first part of the research, it adopts a real radiation test through neutron beams which is a technique not usually seen in other projects. It targets three CNNs: YOLO, Faster R-CNN and ResNet run on three different GPUs architectures developed by NVIDIA: Kepler, Maxwell and Pascal.

This type of approach leverages the differences in the electronic implementations of GPUs, the results show that Pascal GPUs, built using fin field-effect transistor (FinFET), have an error rate that is a magnitude lower than both Kepler and Maxwell GPUs that are built using CMOS devices instead. Moreover the radiation test shows opposite results with respect to software fault injection with an higher percentage of crashes than SDCs. On the contrary the same results observed in software base fault injection are found in this work confirming the tendency of faults to spread among several threads making deeper networks like ResNet and Faster R-CNN more likely to observe SDCs rather than a narrower network like YOLO.

Finally the authors highlight that more than 50% of GPU processing is spent on General Matrix Multiplication operations, for this reason they suggest adopting an algorithm based fault tolerance (ABFT) strategy that focuses on correcting errors in these types of operations resulting in a correction rate of approximate 87%.

The second part of the work uses SASSIFI to perform software based injection on YOLO, from this campaign the authors propose a redesign of the max-pool layer that is able to detect up to 89% of critical SDCs.

Optimizing Selective Protection for CNN Resilience

In this work [47] the authors propose two domain-specific selective protection techniques for CNNs called Feature Map Level Resilience (FLR) and Inference Level Resilience Technique (ILR), by combining these two techniques they claim to obtain a error coverage of approximately 99%.

FLR quantifies the vulnerability of each feature map in a CNN by computing the likelihood that an error appears and propagates to the output of the system. To produce this value the authors of the paper use a custom metric called $\Delta Loss$ which is calculated for each fmap and consists of the average absolute difference between the cross entropy loss values observed in an error free interference and the ones calculated during an error injected inference across all the injections.

The second metric produced in this work is ILR, it is computed from two decision functions which operate on the softmax operator, the first one is called Top1-Conf and assesses the vulnerability of an inference based on the value of the highest confidence value observed, if this value resides above a certain threshold then the inference is deemed robust. The second decision function is called Top2Diff and explore the difference between the top two classes in the softmax, in this case a smaller difference is more likely to suffer an SDC since any fault might have a decisive effect.

The combination of these two techniques is evaluated against 7 CNNs trained on the ImageNet dataset and shows an high error coverage for quantized inferences while keeping a low overhead overall.

In this chapter, we introduced the core concepts necessary to understand our technique. Moreover, we discussed the most interesting related works analyzed during the design phase of the proposed methodology.

We deem the available tools inadequate for the task we set to solve. Either due to technical limitations such as the usage of underperforming error simulator/fault injector tools or due to methodological fallacies such as the absence of real-world assessments of error models.

For this reason, we believe that a methodology combining an efficient and performing architectural fault injection tool with an easy-to-integrate and fast error simulator at the application level is necessary. In particular, it is fundamental to provide an experimental evaluation of the complexity of usage of this tool that must be easy to integrate in order to be efficiently integrated in the design flow of a Convolutional Neural Network.

3 | CLASSES

This chapter presents CLASSES, the error simulation framework selected for our work, and the improvements introduced in this thesis. We first provide an overview of the tool by discussing the key ideas at the basis of it. Then we analyze each improvement highlighting the problems presented by the original version of each component and the reasons behind the changes made.

3.1. The Methodology

As discussed on the previous chapters the reliability analysis of a Convolutional Neural Network running on a GPU relies on two different levels of abstraction, the application and the architectural one. Fault models have been widely validated at the architectural level thus giving realistic methodologies to simulate them, nonetheless it is crucial to simulate errors at the application level in order to compare the behavior of a faulty CNN to the nominal one. The main flaw in the reliability analysis is therefore the disconnection between these two levels, a weakness that this framework aims at solving. The main goal of the framework is to propose robust error models at the application level that are generated through a systematic analysis of the effects of the faults injected at the architectural level, this approach ensures that the proposed error models are reliable and realistic by design thus enabling a more reasonable error simulation directly onto the CNN under analysis.

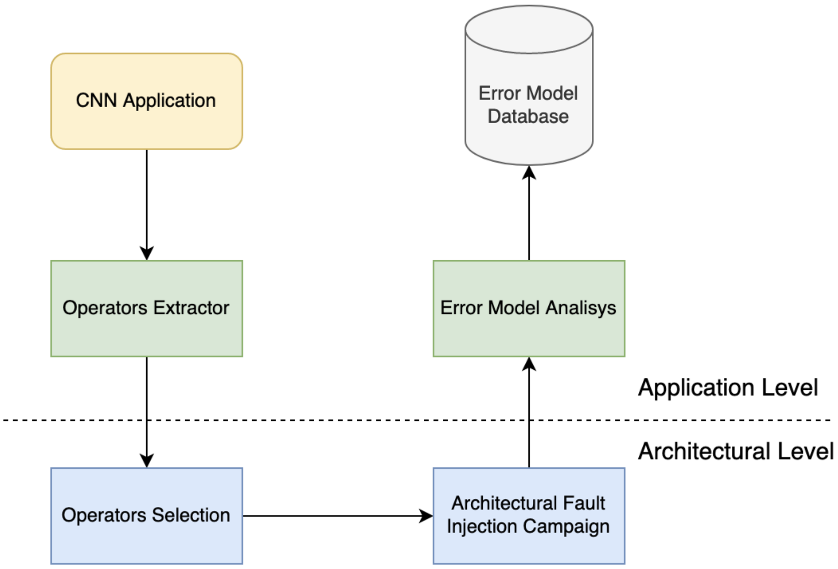

Figure 3.1: Framework Architecture - Initiation Phase

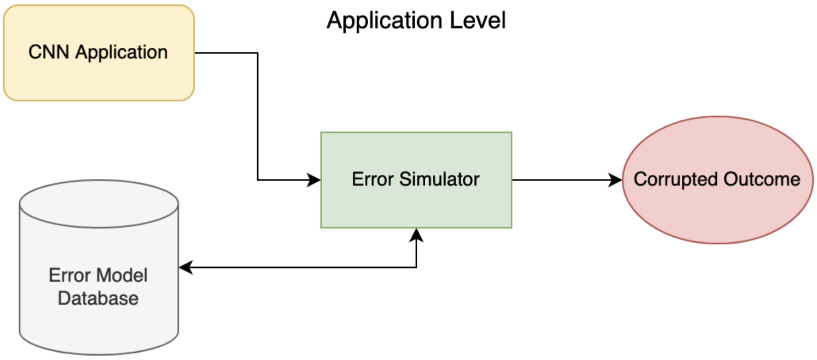

Figure 3.2: Framework Architecture - Usage

To pursue its intended goal the framework is structured into two different sections as seen in Figure 3.1 and Figure 3.2.

One is executed once and is required to build the error models database, the other is executed for each CNN under analysis. The initialization phase begins by writing appropriate scripts using Caffe to execute single ML operators in a controlled environment. These are called **injectable operators** and will be the ones supported by the framework for injection. Then given a set of inputs and golden outputs for these operators what follows is a complete GPU injection campaign carried on for each of them. The goal of this phase is to store any corrupted output. We should note that the inputs and the outputs used have been extracted from existing CNNs to ensure that the values come from real-world applications.

After the campaign is completed the corrupted tensors are analyzed to extract features of interest as explained in Section 3.2. After the analysis is executed a database of error models is created and the initialization phase is concluded.

The standard workflow of a user of the framework begins with the selection of a CNN to analyze. The instrumentation of the error simulator is carried on and the user has to decide which of the supported operators will be corrupted. After this decision is done the error simulator will appropriately extract an error model from the database and the CNN will be executed with the insertion of the extracted error. It is important to notice that the workflow of the user of the framework is entirely executed on TensorFlow.

3.1.1. Operators Extractor

The first step of the analysis consists of the identification of the basic operators that compose a CNN in order to observe and define the error models that characterize their nominal behavior. All the Machine Learning frameworks such as Tensorflow or Pytorch express any ML models as data-flow graphs where the nodes are the units of computation, such as BiasAdd, Convolution or Sum. The edges connecting the nodes represents the set of tensors (multidimensional matrices of data) that are taken as input and produced as output. This identification phase is crucial since the error simulation is carried on at the operator's level. Moreover all the error models are defined based on the differences that are observed between golden and faulty outputs of the operators.

3.1.2. Operators Selection Phase

Executing the architectural fault injection campaign is a time consuming operation, to ensure that it produces meaningful results without without using too many resources we must decide, among all the possible shapes of an operator, which one to use. It is in fact common that a given operator, i.e. Convolution, appears multiple times with different shapes in a single model.

The decision taken was to select the instances with the smallest shape possible. This choice has been made due to lower loading and execution times of the operator's instance and smaller outputs produced. Having a lower execution time drastically reduce the duration of the architectural injection phase that requires thousands of executions.

Instead a smaller output leads to a faster analysis without compromising the correctness of the process since it is always possible to scale the produced error models to fit larger tensors.

3.1.3. Operators translation

After having selected the set of operator's instances to inject they are translated into custom Caffe programs that perform three distinct phases:

1. *Preamble*: Loading of the inputs, nominal output and parameters of the operator's instance
2. *Execution*: Execution of the CNN operator's instance
3. *Epilogue*: Comparison between the obtained output and the nominal one, in case of differences the produced tensor is stored.

Having such small and specific environments ensures that the fault injection campaign can target only the instructions that are related to the execution of the operator under analysis without any interference from the outside. To develop these environments we decided to use Caffe, a C++ framework developed at Berkeley University with the goal of creating a tool for the modelling and deployment of reliable and scalable Machine Learning networks. The decision of using this specific framework instead of relying on Tensorflow C++ API directly is due to an incompatibility on libraries needed to build Tensorflow with respect to SASSIFI which is the Fault Injector described in Chapter 2.5.2. Nonetheless Caffe grants the capability of creating any layer of interest without loss of accuracy making it a valid choice for the deployment of our framework.

3.1.4. Architectural Fault Injection

The Architectural Fault Injection phase aims at producing faulty outputs of CNN operators executed on GPU by injecting a single bit flip during the execution of said operator into the input tensor. The whole phase is composed by different steps that are here explained.

Campaign Sizing

The goal of the first step is to determine the size of the injection campaign which is crucial to obtain meaningful results. An underestimated number of faults injected could lead to a loss of precision and, subsequently, error models that are not relevant in a real life execution of the machine learning model under analysis. On the opposite a campaign size which is overestimated may not provide any additional information and result in a waste of time and resources.

In the context that we consider the injectable sites are the GPU assembly instructions that constitute the GPU kernel of operator's implementation, unfortunately the number of injectable instructions can be enormous, making it de facto infinitely large if we take into account multiple inputs available.

Therefore, we adopted an ad-hoc sizing mechanism that takes into account the architecture of the GPU. It is know by the user that a modern GPU performs data and thread parallelism, each thread does not manage the whole available memory but only a reduced fraction of it and each instruction is executed by multiple threads on different portions of memory to parallelize the execution of the code.

For this reason, under the Single Instruction Multiple Thread (SIMT) paradigm we can reduce the number of instructions to consider to just the set of them executed by one thread. Moreover, depending on the flexibility of the GPU fault injector, the instructions considered for a single campaign could be divided into families of operations with shared behaviors such as FP32 instructions or only the memory instructions (LD/SD). Considering this flexibility and the previously explained considerations we can define the following heuristic evaluation for the campaign size n

$$n = \frac{M}{T} * N \tag{3.1}$$

where the parameters used are

- T: number of threads.
- M: global memory allocated for the kernel.
- N: number of instructions considered

This is clearly an estimate that can be modified according to the user's preference to better suit each particular case considered.

Fault List Definition

The second step of this phase consists in the definition of the fault list applied to the injection campaign. Each GPU fault injector has been developed with a set of available faults to be used, SASSIFI makes usable four different modes that are:

- Single bit flip: a single bit in a register is flipped.
- Double bit flip: two adjacent bits in a register are flipped.
- Zero value: the value of a register is substituted with the value zero.
- Random value: the value of a register is substituted with a random value.

In our methodology we will only consider the single bit flip mode for two reasons:

- It is the most widespread and validated model that represents the effects of a particle strike. Moreover the difference between a single and a double bit flip has been proved [48] to be irrelevant for the considered methodology.
- Random or zero values are of no interests for our framework since the possible values in any input do not span across the entire available range of the floating point domain. Values are filtered either by normalization layers, resulting in zero mean and unitary variance tensors, or the activation functions, which have narrow co-domains.

Campaign Execution

After having decided the size of the campaign, extracted the required tensors and generated the fault list, the campaign for each operators are executed one at a time and all the faulty tensors are stored for the subsequent analysis.

3.2. Error Model Definition

The following section explains the characterization of the errors produced at the architectural level, this classification is intended to provide a scheme that can be replicated at the application level. The basic unit of any Machine Learning model is the tensor, a matrix of data produced and consumed by any layer, for this reason the analysis will be based on differences and characteristics of this type of structure. The formal definition of an error that will hold true for the entire analysis is the following:

$$|x - x'| \geq \varepsilon \qquad \varepsilon > 0 \tag{3.2}$$

Given two floating points values x and x' which represent the golden value and the output value we consider the current value an error if their difference is greater than a small positive number ε. This threshold is context-sensitive, so the correct value depends on different parameters and the accuracy required, always keeping in mind that floating-point arithmetic is not associative, as explained in [49]. Given this definition of an error we consider an erroneous output a tensor that presents at least one error with respect to the golden reference. For the developed framework it has been considered reasonable to chose a value of 10^{-3} for ε, this is due to the fact that all the data managed by the CNN models under analysis are normalized and present a sufficient degree of regularity.

Having defined the concept of error we can now describe the three parameters that were taken into account during the analysis of the faulty outputs to produce the error models, these parameters are the cardinality, the values of the errors and the spatial pattern. They allow an algorithmical description that can be then stored to create a database of error models easily deployable.

Operators Selection

The error modelling has been applied on a subset of the existing operators that was extracted from a TensorFlow implementation of YOLO V3 CNN.
While the framework does not target all the ML operators the ones selected are among the most common and allow for a comprehensive error simulation campaign as we proved during this thesis. Table 3.1 presents the selected instances with their respective sizes that, as explained in Section 3.1.2, are the smallest possible.

Operator	**Input**	**Output**
Convolution 1	512 x 13 x 13	256 x 13 x 13
Convolution 2	128 x 52 x 52	256 x 52 x 52
Convolution 3	256 x 52 x 52	512 x 26 x 26
Add	1024 x 13 x 13	1024 x 13 x 13
Batch Norm	256 x 13 x 13	1024 x 13 x 13
Biasadd	256 x 13 x 13	256 x 13 x 13
Div	1 x 10647	1 x 10647
Exp	1 x 8112 x 2	1 x 8112 x 2
Leaky ReLU	256 x 26 x 26	256 x 26 x 26
Mul	1 x 8112 x 2	1 x 8112 x 2
Sigmoid	1 x 2028 x 80	1 x 2028 x 80

Table 3.1: Set of operators selected for the error modelling

Cardinality

The first parameter under analysis is the cardinality which is an intuitive concept that counts the number of errors that occurred in the output. Recalling the definition of error that we previously discussed the algorithm to produce this parameter is the following:

We first produce the differences tensor which is the absolute value difference bit by bit between the golden and the faulty tensors. Then we compare each difference with the threshold ε and count how many of them are greater, this number is the result of the algorithm. Due to the high degree of symmetry in the operators analyzed different injection campaigns produce a small set of cardinalities, for this reason we create a probability histogram where each cardinality is associated with the probability of its occurrence. This

Algorithm 1: Cardinality extraction

```
Input: Golden Tensor and Faulty Tensor
Output: Cardinality and probabilities
begin
    $differences = ||golden\_tensor - faulty\_tensor||$
    $equality\_tensor = differences > \varepsilon$
    return $equality\_tensor.count()$
end
```

histogram is different for each operator and further analysis will be performed in Section 3.2.2.

Domain of Corrupted Values

The second parameter that is analyzed is the domain of the corrupted values. We want to understand the magnitude of difference between an error and its golden reference. This type of analysis ensures that the error models generated are as faithful as possible.
This analysis was performed by manually inspecting each corrupted value and comparing it with its golden counterpart.

3.2.1. Spatial Patterns

The last parameter under analysis in our framework is the spatial pattern or how each error is placed with respect to the others. The possible patterns are not random but are highly dependant on the computing device used and the implementation of the operators under analysis. Moreover, unlike the two other parameters this time the analysis must take into account the shape of the tensor to produce a valid error model. Therefore the effort was spent to classify these types of patterns according to a model that is reproducible and applicable also to tensors with different shapes from the observed ones.
Lastly, it should be noted that writing a script that is able to identify these patterns must require the developer to have previously manually analyzed a set of faulty tensors. The first step in the analysis of the spatial patterns is to define the index of a fault, we can apply two definitions:

- *Linear:* If we consider a tensor as a uni-dimensional vector then the index is the offset from the initial element.
- *Multi dimensional:* If we consider a tensor as a multi dimensional matrix then the index is a tuple of shape (*feature map*, *height*, *width*)

The choice to use one mode or the other is irrelevant provided that the framework we are dealing with stores each value in the order we expect them, this can be easily checked and, if needed, corrected with a simple script; we will use the first one for an easier discussion. Both the definitions we provided define absolute positions, this means that they are related to the size of the tensor under consideration, in order to remove this limitation we provide two additional vectors that will define the classification of the patterns. Both are created starting from a vector containing all the indexes of erroneous values in the tensor under analysis, called *Errors vector*:

- *Offset vector:* the indexes in the *errors vector* are sorted and the first one is subtracted from all the others, the result is a vector containing the relative offsets from the first index.
- *Stride vector:* contains the strides between consecutive pairs of elements of the *errors vector*

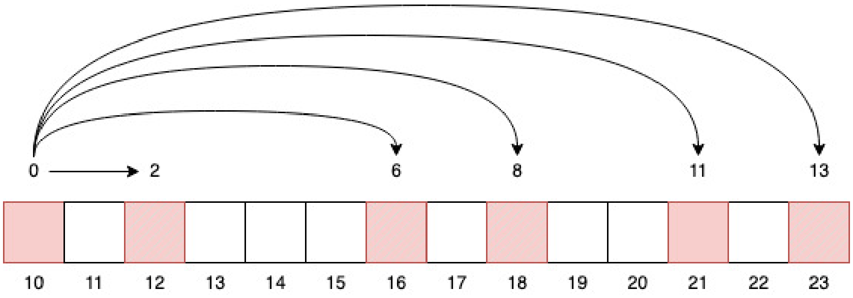

Figure 3.3: Offsets vector

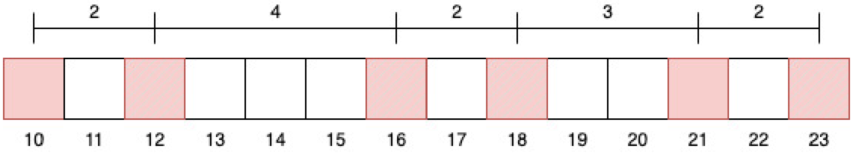

Figure 3.4: Strides vector

Let's analyze an example to better describe these two vectors, suppose we have a faulty tensor with 6 errors, our *errors vector* is the following [10, 12, 16, 18, 21, 23], starting from the first index we subtract the smallest value, 10, to all the elements obtaining the following *offsets vector*: [0, 2, 6, 8, 11, 13] as show in Figure 3.3. Instead the *strides vector* is built by subtracting each pair of consecutive indexes thus obtaining the following [2, 4, 2, 3, 2] as shown in Figure 3.4.

Having produced these two vectors we can now define what kind of pattern we are dealing with, let's suppose we have a tensor whose width is 4 and we would like to understand

if this is a row pattern, i.e. all the errors are on the same row. Dealing only with the cardinality of the errors would make us conclude that this is not a row pattern since the number of errors is greater than the width of the tensors; however we want to generalize our analysis for tensors of any size and under the assumption of analyzing the smallest tensors available we can say that this is indeed a row pattern.
To better understand the reasons behind this conclusion we have to observe the *strides vector*, we can observe that each value is less than the width of the smallest tensor available meaning that it can be declared a row pattern.

Producing meaningful error models

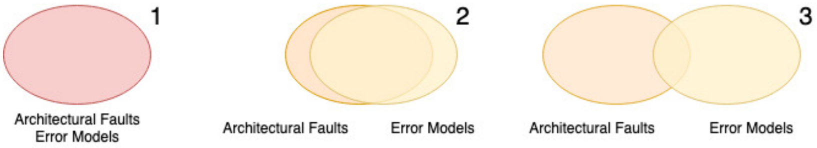

Figure 3.5: Intersection of architectural faults and error models

Translating architectural faults into error models is an error-prone process since it requires the developer to manually analyze the tensors beforehand and then produce the translating script; for this reason we will now discuss three possible situations that represent different overlaps between architectural faults and error models as seen in Figure 3.5.

- Figure 3.4.1 represents the best possible outcome of the process, the generated error models capture the entirety of the architecture faults set.
- Figure 3.4.2 represents the outcome in which the produced error models capture most of the possible architectural faults, this situation in still acceptable but the developer should keep in mind that the effects of some faults will never be tested.
- Figure 3.4.3 instead represents the worst possible scenario, the produced error models cover just a small portion of the architectural faults while introducing a huge number of models that are not representative of real life possibilities. This situation could arise if the error model database used is not tuned to the framework and the GPU used.

The goal of the framework was to produce error models that cover as many architectural faults as possible. Changing the ML framework used for the definition of the networks could worsen the coverage and require a new tuning of the models.

3.2.2. Obtained Error Models

Having described the three parameters that make up an error model we can analyze the results obtained.

Cardinality

For each operator analyzed a probability distribution of the observed cardinalities has been generated, as expected the operators can be split into two main groups. The first one is composed by the "linear kernels" that are Add, Div, Exp, BiasAdd, Leaky ReLU and Mul. These are the kernels that do not involve any kind of shared memory, each thread processes only one element and stores its result into the output. For this reason any kind of error introduced will only affect one bit of the output.

On the contrary Convolution and BatchNorm are composed by various kernels and rely heavily on General Matrix Multiply (GEMM) algorithms that make great use of shared memory and thread cooperation. This approach results in a higher count of corrupted values in the output. It is of interest to note that, despite relying on these kind of shared operations, the probabilities of high cardinalities are still far lower than those of lower cardinalities. This is due to the lower number of instructions that have effects on other threads.

Figures 3.6 to 3.9 in the following page show the probabilities of various operators.

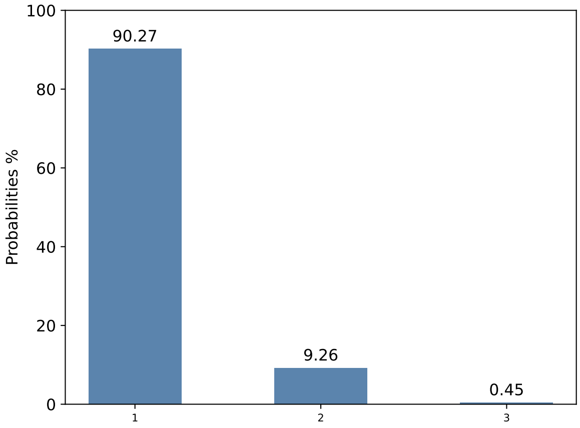

Figure 3.6: Add

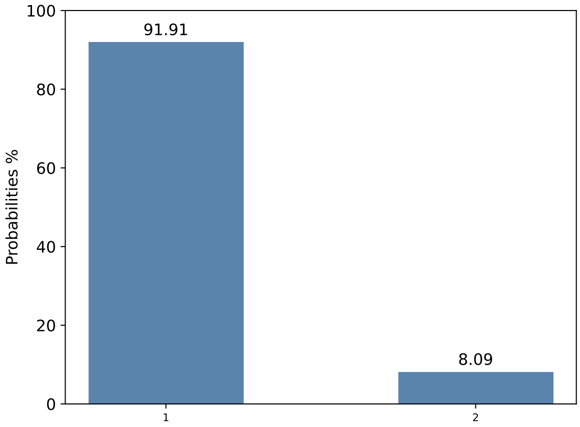

Figure 3.7: Exp

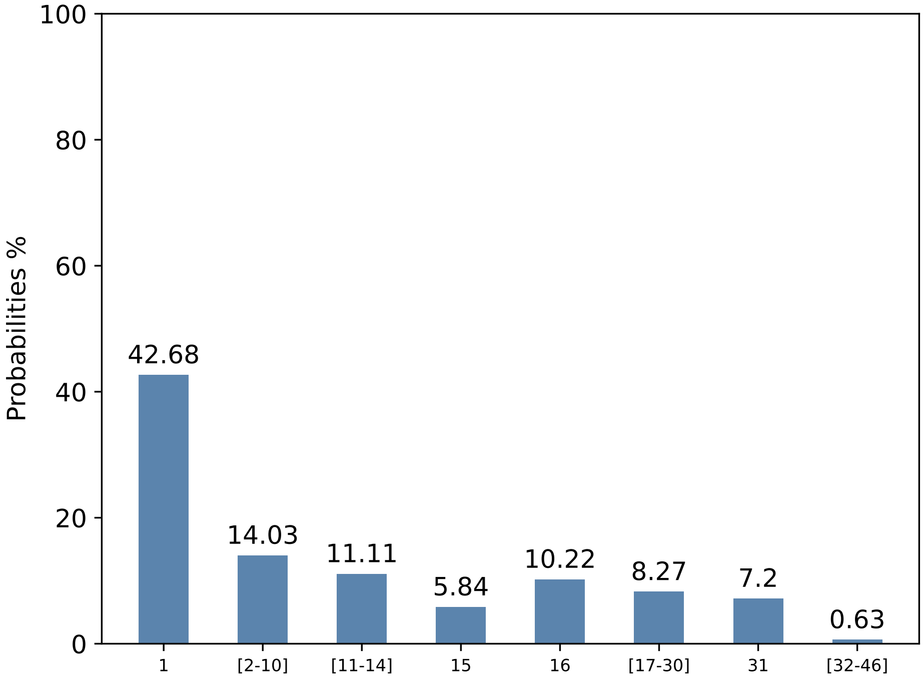

Figure 3.8: Convolution 1

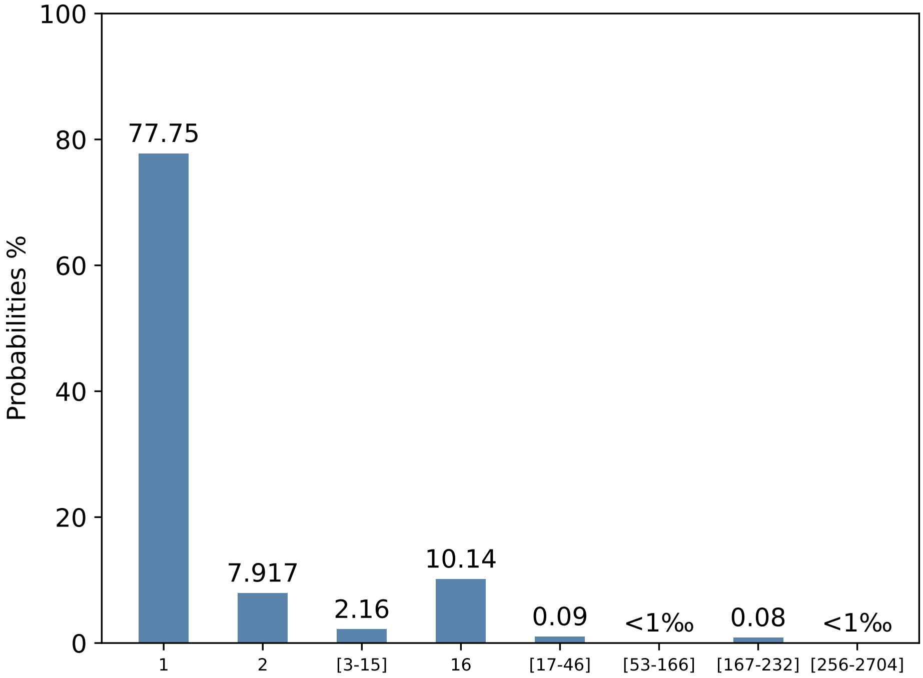

Figure 3.9: BatchNormalization

Domain of Corrupted Values

The analysis on the domain of corrupted values highlighted five different common classes that can be used to split the results. The classes are:

- NaN value: the faulty value becomes NaN while previously it was a valid number, this might be caused by an illegal operation like a load with a wrong address.
- Zero value: The faulty value becomes zero while previously it wasn't. A zero value might happen when a write operation is not executed following a bit flip.
- Bitflip: The faulty value differs from the original one by just one bit. This case arises when the faulty input value is presented as-is in the output tensor, usually happens with operations at the end of the kernel.
- $[-1,1]\setminus\{0\}$: The difference between the faulty value and the golden one is in the closed interval -1,1, this class usually represents error that occurred in the significant bits of a IEEE 754 floating point value.
- Random: This class contains all the error that are not part of the previous classifications.

The following algorithm describes how this classification is performed. It is of interests to note that the order in which the classifications are performed is not random but is structured in a way to ensure that a bitflip error is not wrongfully labelled as a -1,1 error.

Algorithm 2: Classification of domain of corrupted values

```
Input: Faulty tensors
Output: Domain descriptor
begin
    nan_count, zero_count, bitflip_count, between_count, random_count = 0;
    foreach faulty_tensor ∈ faulty_tensors do
        error_locations = find_errors(golden_tensor, faulty_tensor);
        foreach error_location ∈ error_locations do
            f_value = faulty_tensor[error_location];
            g_value = golden_tensor[error_location];
            if is_nan(f_value) and not is_nan(g_value) then
                nan_count++;
            else if is_zero(f_value) and not is_zero(g_value) then
                zero_count++;
            else if is_bitflip(f_value, g_value) then
                bitflip_count++;
            else
                if -1 ≤ g_value - f_value ≤ 1 then
                    between_count++;
                else
                    random_count++;
                end if
            end if
        end foreach
    end foreach
    return all the counters
end
```

It is not possible, unlike the previous parameter, to link the domain of corrupted values to the GPU architecture. The results obtained are applicable to any hardware. Figures 3.10 to 3.13 shows some of the distributions found.

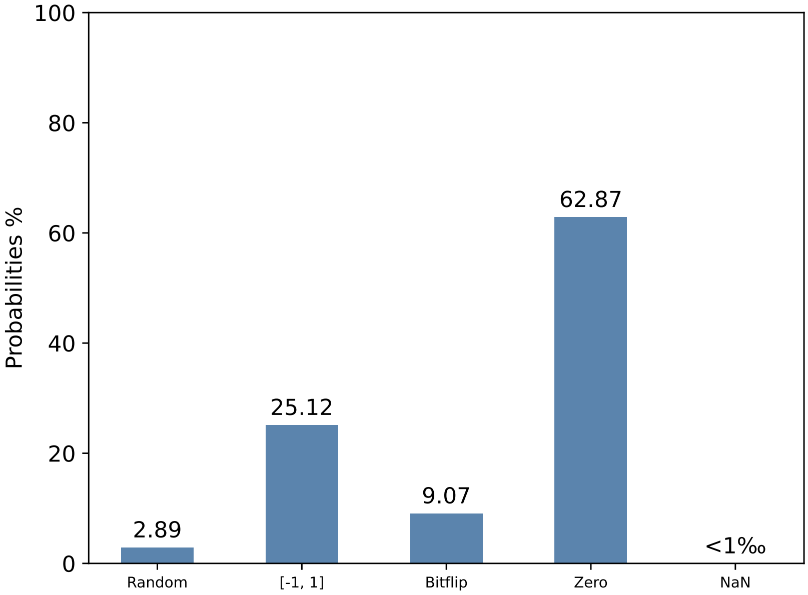

Figure 3.10: Add domain values

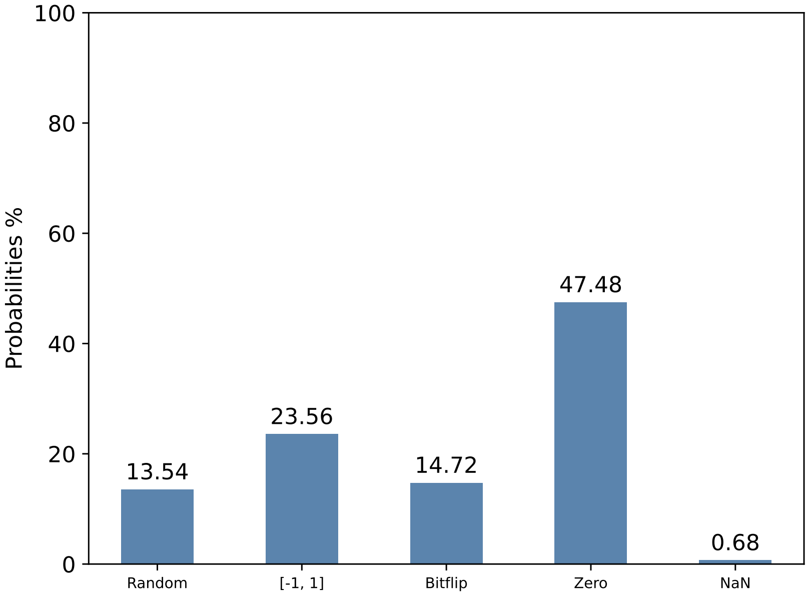

Figure 3.11: Exp domain values

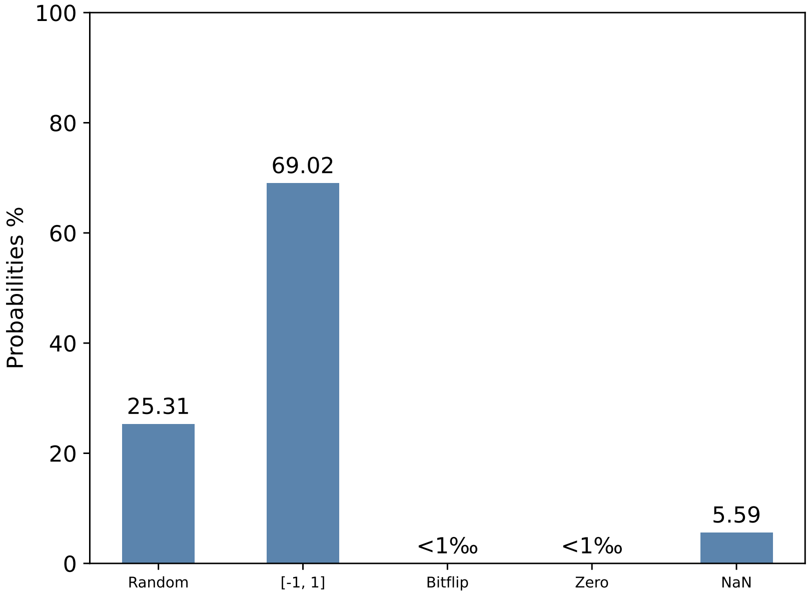

Figure 3.12: BatchNorm domain values

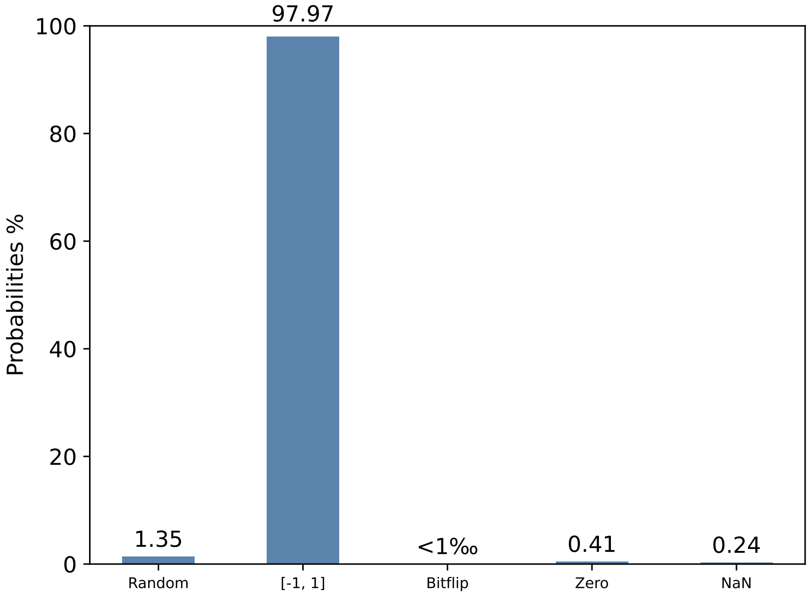

Figure 3.13: Convolution 1 domain values

3.2.3. Spatial patterns

The last parameter that makes up the error model definition is the spacial pattern. It describes the spatial positioning of the errors in the output tensor. Unlike the two previous parameters this time the analysis must take into account the shape of the tensor and, for this reason, we can classify the results into two main classes.

Same Feature Map

The first class of patterns describe those cases in which all the errors are located in the same feature map. For this class four different sub classes were identified:

- **Single Point:** This is the simplest case possible where the output presents only one erroneous bit. The analysis suggests that there is no particular position where probability of an error appearing is higher that the others.
- **Same Row/Column:** This is the case in which multiple errors are either in the same row or in the same column of a feature map. The errors could be all one next to the other or with one or more valid bits in the middle.
- **Same Block:** The locations are spaced in multiple of the GPU block size.
- **Unclassified:** Represents any case that does not fit the previous descriptions.

Multiple Feature Map

The second class of patterns identified contains those cases in which the errors are spread among multiple feature maps. This class is observed only in complex operators and only when the cardinality is high (≥ 16). Five sub classes have been identified:

- **Bullet Wake:** The same location is corrupted in multiple feature maps. This can be seen as a Same Row/Column case along the third dimension. Likewise the feature maps involved can be sequential or interspersed with feature maps without any corrupted value.
- **Same Block:** Similarly than the previously discussed scenario here errors are spaced in multiple of the GPU block size and spread across multiple feature maps.
- **Shatter Glass:** This can be seen as a more complex bullet wake. There is still a common location among multiple feature maps but, in some of them, the error is spread into multiple positions following the Same Row/Column classification previously described.

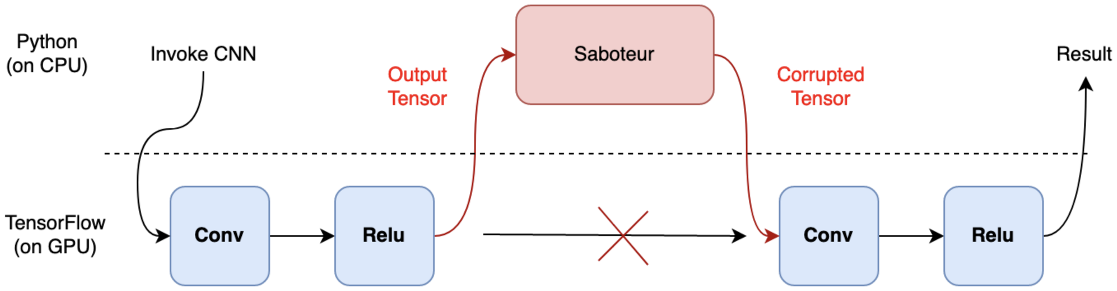

Figure 3.14: Error Simulator

- **Quasi-Shatter Glass:** This is a relaxation of the previous case where the shared bullet wake position is missing in some feature maps but the row or column pattern is still present.
- **Unclassified:** Represents any case that does not fit the previous descriptions.

3.2.4. Error Simulation

The last element of the framework is the error simulator where the reliability of a CNN model is tested by inserting the errors that were previously modelled. This step is entirely executed at the application level as shown in Figure 3.14.

The process of error simulation is based on the concept of the *saboteur* that splits the execution of a CNN in two different halves in order to modify the output of a single operator instance and resume the execution. To ensure that the error simulation can be used as soon as possible in the development phase of a ML model it should comply with the following constraints:

- *Small overhead:* the overhead introduced by the error simulation should be reasonably small, both in memory usage and timing. The timing constraint is required to ensure that our framework can be used even with real-time systems with strict temporal deadlines. Instead requiring small memory overhead is needed to avoid any kind of memory saturation, especially with models manipulating large tensors, and related memory errors.
- *Transparency:* the error simulator should be easy to integrate with the available CNN model without requiring any modification of the existing code. This is required to make our framework convenient for the developer to use.

The error simulation phase is based on the assumption that the error models used are representative of real life architectural faults as explained in Section 3.2.1, the following section provides a description of the technologies used in the development of the first version of the framework for which the provided error models are the most representative.

3.3. Framework Implementation

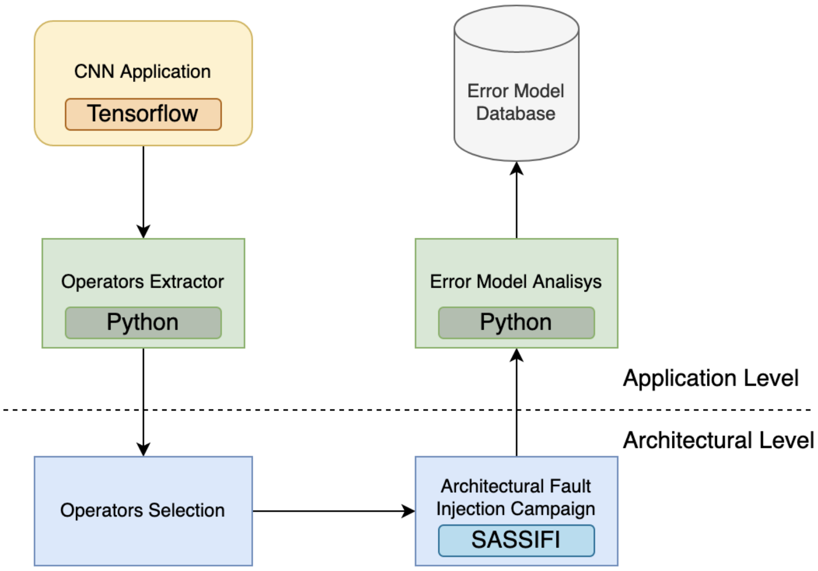

Figure 3.15: Framework Architecture Technologies - Initiation Phase

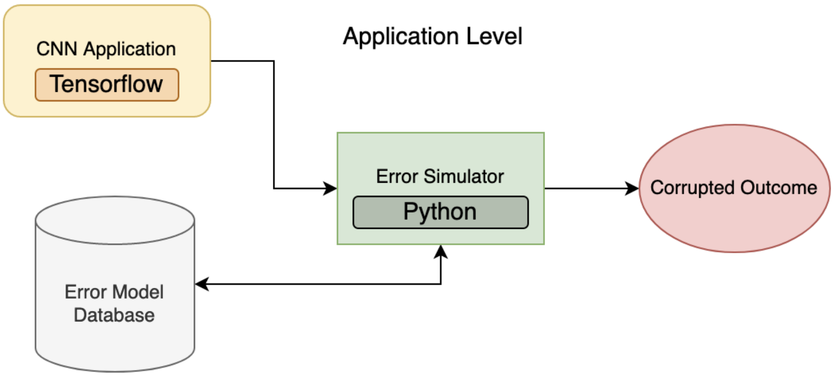

Figure 3.16: Framework Architecture Technologies - Usage

For the implementation of the framework both state-of-the-arts tools and in-house ones were used, as seen in Figure 3.15 and Figure 3.16.

The development of the CNN applications under analysis was performed using Tensorflow 1 which was the latest version available when the framework was developed, moreover Tensorflow is one of the most used Machine Learning frameworks which explains why we decided to use it. Following the decision to use Tensorflow we built an Error Simulator to perform the application level reliability analysis using Python and integrating it with the ML framework. We decided to discard the publicly available state of the art tools, i.e. TensorFI, due to the limitations in the error models available, the effort required to setup a campaign and the execution times needed. It is worth noting that, even if the error simulator has been developed with Tensorflow as the target the methodology is perfectly transportable to any other ML framework such as Caffe or Pytorch. The architectural part of the framework has been developed using Caffe C++ and SASSIFI, we used Caffe instead of Tensorflow due to an incompatibility with the required version of CUDA. Tensorflow requires CUDA 8.0 while SASSIFI requires CUDA 7.0, this forced us to develop the ML Operators using Caffe, however this discrepancy between levels does not tamper the framework itself since there is a one-to-one mapping between CNN's operators in Tensorflow and Caffe. With the same inputs we are able to obtain the same output at any level of abstraction.

Finally we have two custom scripts wrote in Python for the Operators extractor and the Error model analyzer, the former produces a list of injectable operators from a given CNN while the latter is responsible for the analysis of the faulty tensors produced during the injection and the production of error models. The operator selection phase is managed by the developer, from the list obtained with the Operators Extractor a subset of operators is selected and the campaign is prepared manually by creating the necessary folders and storing the required tensors on the GPU.

3.4. Improving the Framework

While using the framework, we identified some problems related to the technologies used. This section explains the issues and how we decided to deal with them.

- **Fault Injector:** The first major issue we found is due to the fault injector used by the framework. While at the time of the development SASSIFI was, as explained in Section 2.5.2, the state-of-the-art fault injector it is now obsolete. The improvements brought by the newest fault injector, NVBitFI, require us a change of framework.

  The major difference between SASSIFI and NVBitFI that we had to track while migrating was that NVBitFI doesn't need to compile the source code with any particular flag. We were thus able to create a common compiling interface among all programs that allow for easy deployment of the operators if any changes are required. Moreover, for the fault injector, we created a script that automates the entire workflow. It ensures that after fixing the required libraries, the process of writing, compiling, and injecting a test program has been automatized and made clear even for those with no previous knowledge of NVBitFI. It is a fundamental change compared with SASSIFI whose learning curve for the framework was steeper. After our changes, the entire package can be easily transferred and used by other developers if they want to upgrade the existing error model database.

- **Migrating to a different ML framework for fault injection:** The second major issue identified during our work with the framework is related to the usage of Caffe. This library was chosen when the injector was developed due to incompatibilities between SASSIFI and cuDNN, NVIDIA's library for GPUs' accelerated neural networks operations, as explained in Section 2.2.3. With the substitution of SASSIFI with NVBitFI, we decide to migrate from Caffe to cuDNN. The reasons behind this decision are multiple.
  Firstly, Caffe was been created as a Ph.D. project at Berkeley by Yangqing Jia as an open-source product. After his research period at the university, the entire project has been managed by the community, and the amount of support available online has drastically reduced. Moreover, in 2017 a newer version of the library was announced and in 2018 it has been absorbed by PyTorch, an ML framework similar to TensorFlow. This migration means that any future development required by our framework will need to be based on a library that is five years old and no longer developed, making it difficult to obtain any support. One last reason behind our decision was that we wanted to use cuDNN since it is the same library that is used by TensorFlow, thus making it better both in terms of support available and methodologies used. To move from Caffe to cuDNN, we were required to write the code that executes each operator from scratch. Using the online documentation [50], we built all the required scripts. We should note that cuDNN operates at a lower level than Caffe. For this reason, we were required to create special handlers and descriptors for any operation. Due to the initial complexity of this process, we created a skeleton program that explains all the steps required with clarity and made it available with the framework for anyone interested in using it.

- **Migrating to a different ML framework for error simulation:** One final issue we identified is the version of TensorFlow used by the framework when it was initially developed TensorFlow 1.x was still the latest release available, but in September 2019 the newest version was released. This newer version introduces major changes and improvements but not using it does not affect the correctness of the analysis performed in this thesis. Nonetheless, we decided to create a Keras version of the Error Simulator which has been developed as a custom layer that can be inserted into a model. We also developed a script to correctly upload the saved weights for the model after the introduction of said layer. We did not use this version of the error simulator but it is available with the framework and can be used on TensorFlow 2.x CNNs.

In this chapter, we described CLASSES and all its components. We analyzed the workflow of the framework and explained the methodology used to build the error model database. We also discussed in-depth the improvements designed and introduced during this thesis which was the first goal of the work.

4 | Case studies in reliability analysis

The following chapter presents the experimental evaluation performed of the proposed methodology. The chapter is divided into two sections.

In the first one we will describe the case studies adopted highlighting their structure and real-world usage, providing an explanation on why we chose them.

The second part will present the results obtained for each case study with an in-depth analysis of the raw data produced.

To test our framework against the most possible conditions we selected four CNN models that perform different tasks and we used three different dataset to ensure sufficient diversity in conditions. Overall we can split the CNNs considered into two main groups. The first contains those models that perform image classification, while the second includes the applications that perform steering angle detection, a fundamental task for Autonomous Driving Cars.

4.1. Steering Angle Detection

With the advent of CNNs the task of pattern recognition has been revolutionized, prior to their widespread adoption most tasks were performed using an initial stage of handcrafted feature extraction followed by a classifier. The innovation of CNNs is that features are learned automatically from training examples, moreover this class of models has been fundamental in image recognition given the capability of the convolution operation to capture the 2D nature of images.

While the first examples of commercially available CNNs date back twenty years their adoption has seen a huge increase in the last few years for two reasons:

- Large dataset of labeled data such as Large Scale Visual Recognition Challenge (ILSVRC) [51] have become publicly available
- CNNs algorithms have been implemented on parallel architectures such as GPUs

which have hugely reduced the time needed for learning and inference.

The increasing capabilities of CNNs facilitated the development of algorithms that go beyond pattern recognition, the three models we are going to analyze are different implementation of DARPA Autonomous Vehicle 2 (DAVE-2), an end to end CNN proposed at NVIDIA that is capable of learning the entire processing pipeline needed to steer an automobile.

DAVE-2 is an evolution of the original DAVE model [52] built with the goal of avoiding the need to recognize specific human-designated features and thus avoiding a collection of if-then-else rules to modify the model's behavior based on these features.

The model proposed by NVIDIA [53], takes as input an RGB frame recorded by a camera positioned at the front of a vehicle, then it performs a normalization of the values and follows it with five convolutional layers intended to reduce the size of the data and to learn the necessary features. At the end of the convolutional layers we have a flatten operation followed by three fully connected layers that produce a single output which is the predicted steering angle required to keep the vehicle in lane. The two CNNs that we have decided to target all implement this model with some differences between each others, for all of them we used this publicly available dataset [54] of 49 thousands of images with different light conditions, road types and orientations.

4.1.1. PilotNet

PilotNet is a publicly available implementation of the NVIDIA's proposed model, it differs from the original model due to the introduction of dropout layers and the addition of an inverse tangent layer at the end of the model intended to give the network the ability to recover the angle curvature from the visual data instead of having to learn how to convert slopes or tangents into radian measures, the training phase required the model to minimize the mean squared error between the predicted steering angle and the golden version.

Operators

As seen in figure 4.1 the model is composed by five convolutional layers that take a $66 \times 200 \times 3$ input and, with the aid of a Flatten layer, produce a vector of size 1164. This tensor is then fed to four fully connected layers that reduce its dimension to produce the final result.

The output of the last fully connected layer is scaled by a factor of two, a decision taken by the original developer of the model.

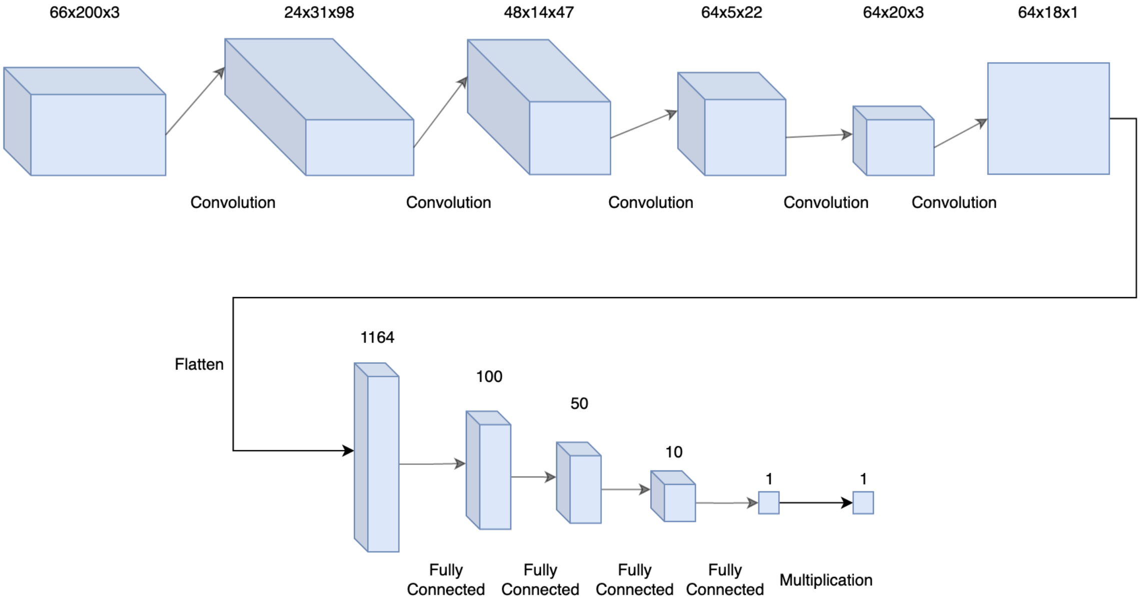

Figure 4.1: PilotNet Model

With our framework we were able to inject all the convolutional operators, the multiplication at the end of the model and the ten BiasAdd operators used by each layer.

Image Selection

To perform the best error simulation campaign possible we selected a set of 45 images that cover all the available cases that range from a clear image on a sunny day 4.2, an image with a strong lens flare 4.3, a street with a dark shadow, vehicles and text on the road 4.4 or a private parking exit with an intense light on the horizon 4.5. These are just four of the possible conditions observed but highlight clearly that the testing performed took into consideration all the possible corner cases.

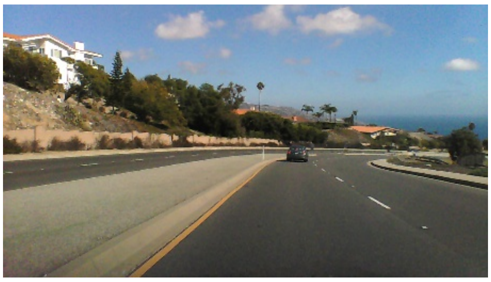

Figure 4.2: Street on a sunny day

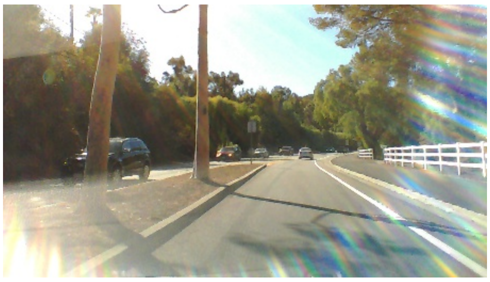

Figure 4.3: Strong lens flare

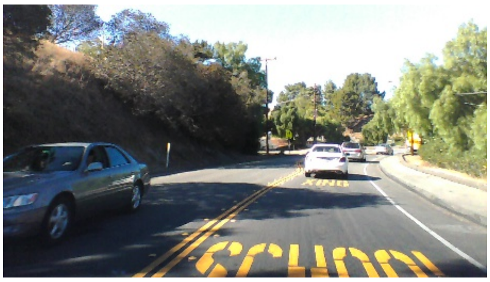

Figure 4.4: Street with shadow and other vehicles

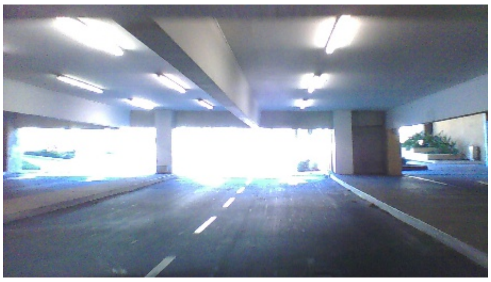

Figure 4.5: Private parking exit with strong light contrast

Output Classification

Differently than an image classification task for this CNN we do not have a correct answer that we are expecting from the model, the authors of the original paper tried to minimize the mean square error between the predicted angle and the golden version provided to the system without giving a range of values inside which any possible outcome would be considered correct.

Nonetheless we have to define a classification of the output to determine if our methodology can produce meaningful results or not, for this reason we propose a 7 ranges classification 4.1 of the absolute difference between the predicted angle and the golden one.

Class 1	Difference between 0 and 10^{-5}
Class 2	Difference between 10^{-5} and 1
Class 3	Difference between 1 and 5
Class 4	Difference between 5 and 10
Class 5	Difference between 10 and 45
Class 6	Difference between 45 and 90
Class 7	Difference greater than 90

Table 4.1: Difference classification for steering angle models

While there is no particular validation that supports this particular set of classes we deemed them broad enough to cover all the available outputs and numerous enough to support our analysis of the results obtained.

Results

For each instance analyzed we completed 5000 runs of the error simulator for each image considered, then we classified the output of each run accordingly to the previously discussed classes.

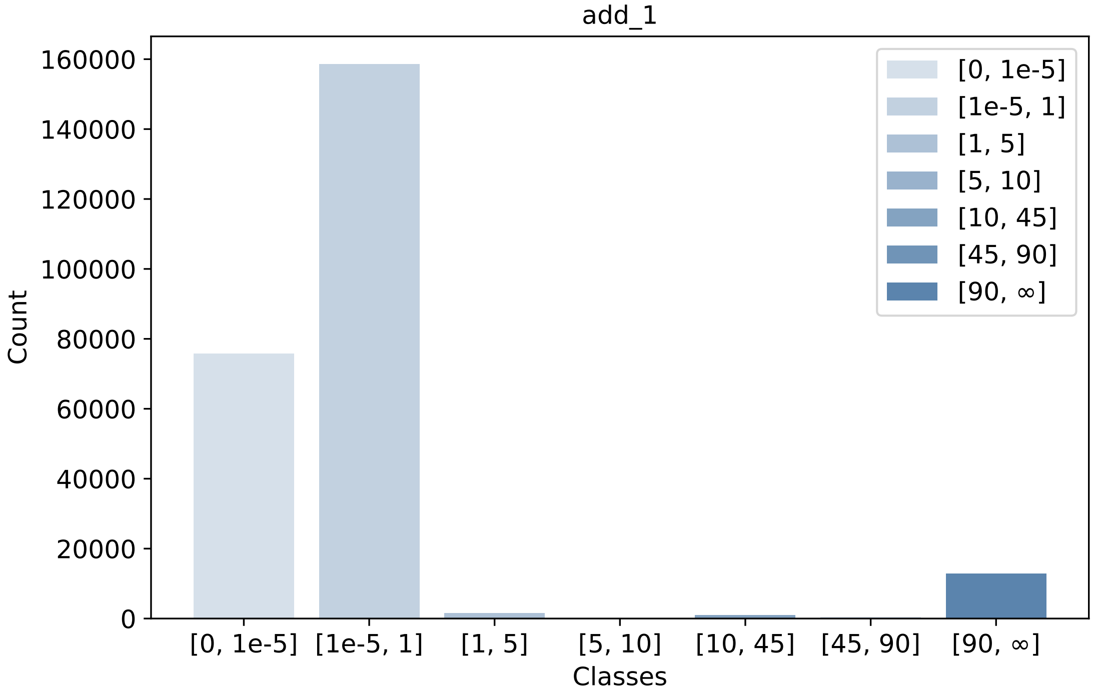

Figure 4.6: First add operator in the net

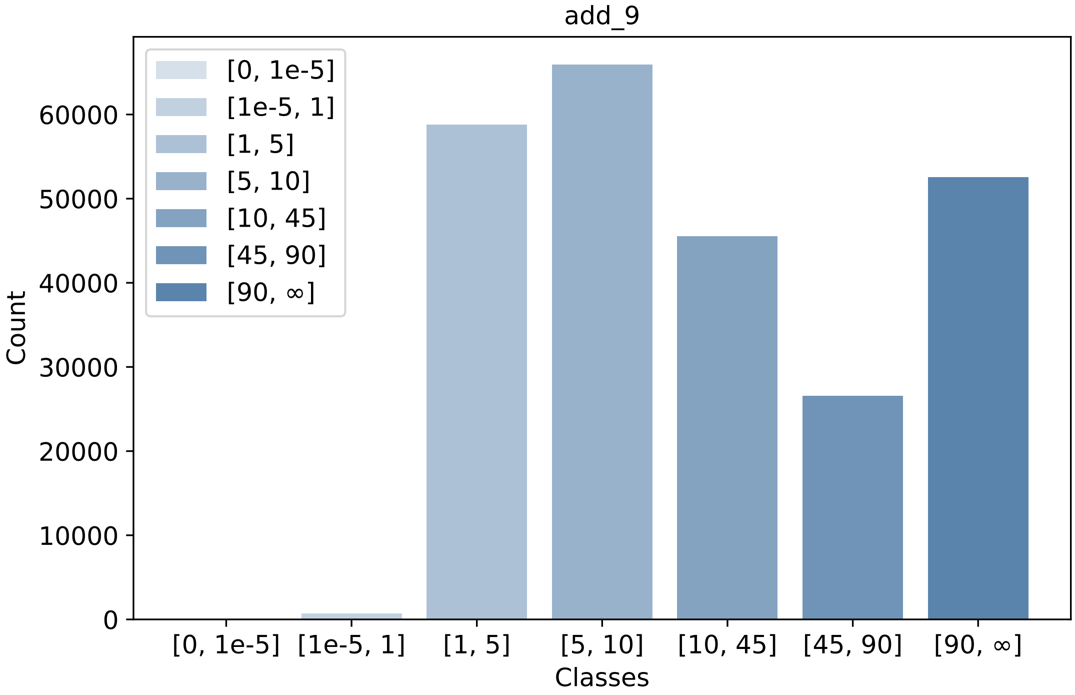

Figure 4.7: Last add operator in the net

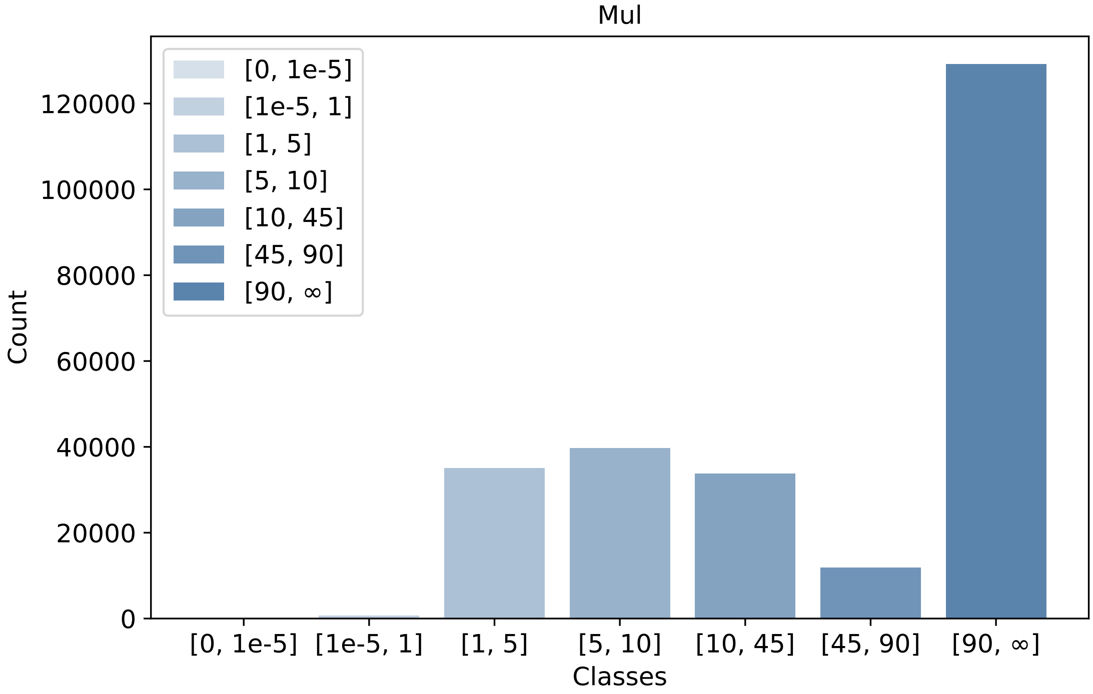

Figure 4.8: Mul operator

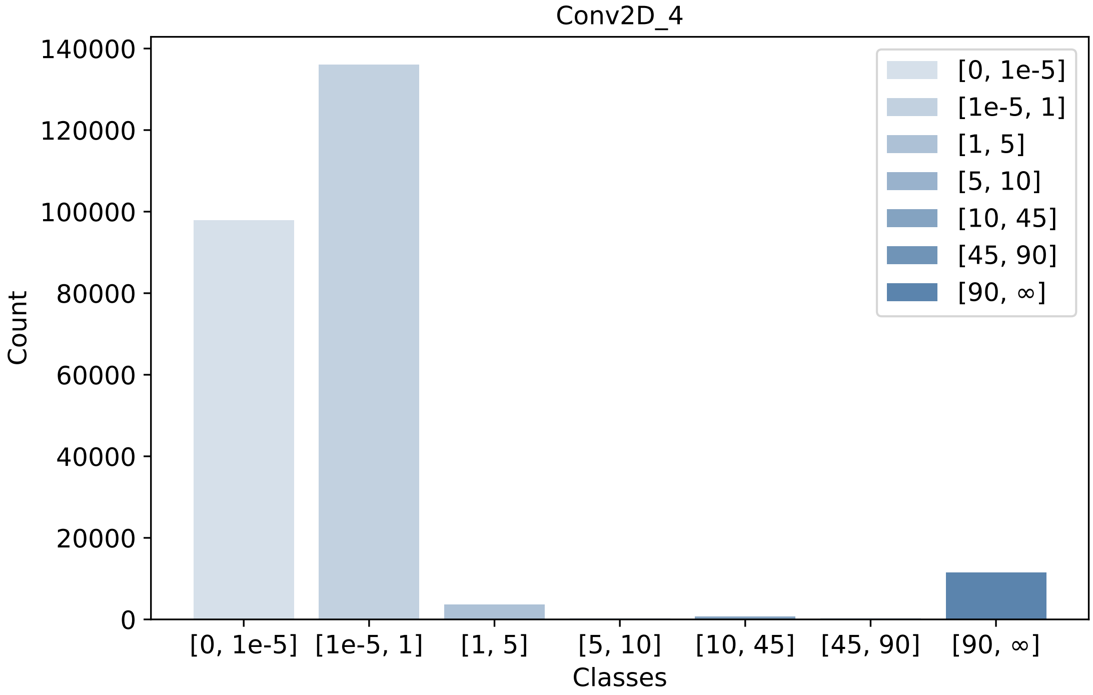

Figure 4.9: Last convolutional operator in the net

From the analysis of this classification we are able to identify two important results.

- **Closer to the output means stronger effects:** the first observation we can make is that a layer closer to the output of the net will be more likely to affect the output of the entire system if suffering from an error. This can be observed by looking at the distribution of classes between the first add operator encountered in the model, Figure 4.6, and the last add operator encountered, Figure 4.7.
  This is due to the fact that, the closer we are to the output of the net the lower is the number of layers that can recover any potential error.
  It is also interesting to observe the last convolutional operator, figure 4.9. Despite being a complex kernel its effect on the output is comparable to the add operator seen in figure 4.6. This is another clear example of the recovery capabilities of this particular model.
- **Not all layers behave similarly:** the second observation we can make is that the type of layer affected by an SDC influences the output of the system.
  This can be understood by observing the aforementioned Add_9 instance, Figure 4.7, in comparison with the Mul instance, Figure 4.8. Despite being nearly at the same depth in the net their influence is vastly different with the former having a distribution of differences spread among all the classes while the latter resides almost entirely in the $[90; \infty]$ range.
  This difference can be explained by looking at the error model database that we have created, shown in Figure 4.10, despite having a comparable cardinality distribution the Add operator presents an higher probability of generating an error belonging to the Zero category, combining this behavior with how this particular instance influence the output we can clearly see that the Mul operator will have more impact on the final value produced.

4.1.2. Comma's AI Steering Wheel Model

The second model belonging to the steering angle detection family that we decide to analyze is the Comma's AI Steering Wheel Model described in [55] which has the same goal as DAVE-2.

Operators

As seen in figure 4.11 the model is a simplification of the PilotNet one we previously described. Only three convolutional layers are used which flatten the $(66 \times 200 \times 3)$

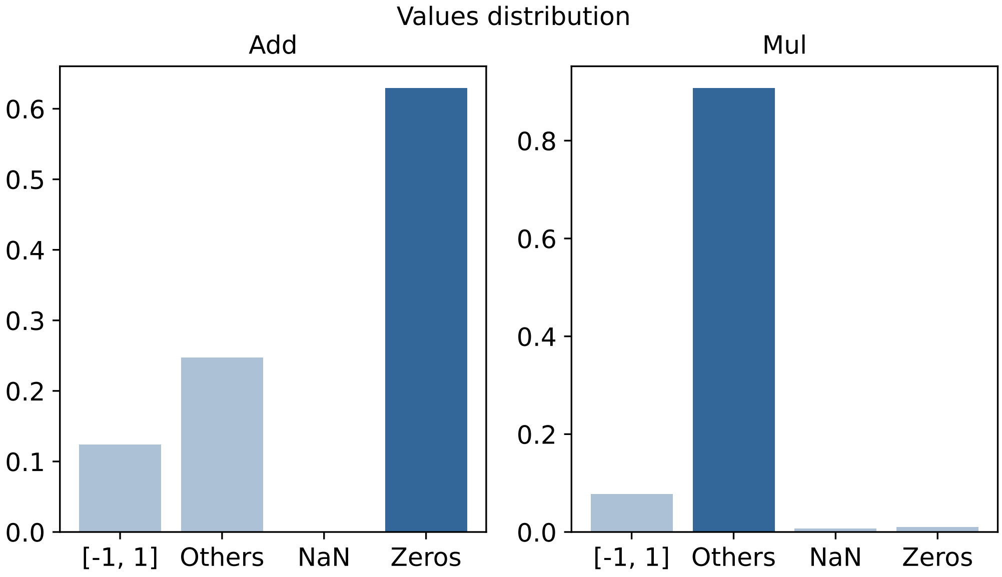

Figure 4.10: Values distribution of Add and Mul error models

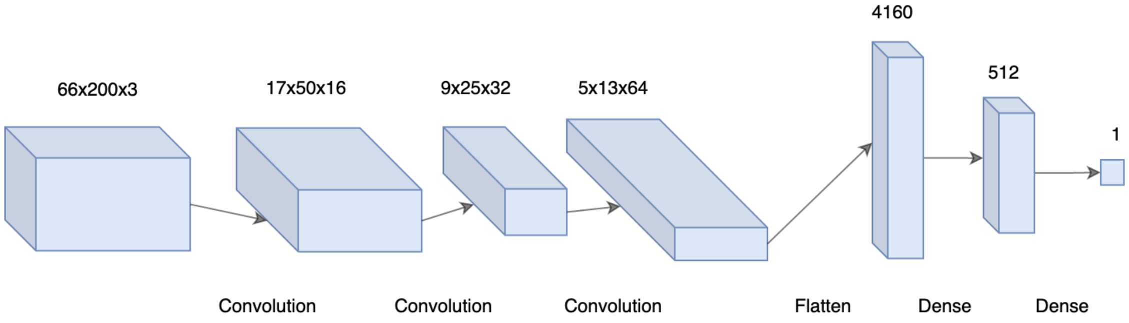

Figure 4.11: Comma's AI Steering Wheel Model

inputs into a tensor of size 4180. This is then fed to two Dense layers that produce the output of the net. Unlike PilotNet the value produced by the last Dense layer is not manipulated any further.
We were able to inject the three convolutional layers and all the five BiasAdd operators linked to each layer of the model.

Image Selection

Since the goal of the net is the same as PilotNet we decided to test the model against the same set of images in order to keep the same corner cases and to be able to compare the robustness of this model against PilotNet.

Results

The results obtained analyzing this model are in line with the ones obtained analyzing PilotNet, the operator with the highest number of differences belonging to the higher classes is the last BiasAdd of the model as seen in Figure 4.12. It is also clear that different layers have different impacts on the final outcome, considering the third convolutional layer we can observe that an error occurring on the BiasAdd operator, Figure 4.15, has an overall lighter impact on the output rather than an error on the convolution operator at the same layer, Figure 4.13.

We can also observe that an ELU activation between the penultimate Dense layer and the last Dense layer highly masks the errors affecting the BiasAdd operator of the second to last Dense, as seen in Figure 4.14 where the most common class of differences is $[10^{-5}, 1]$.

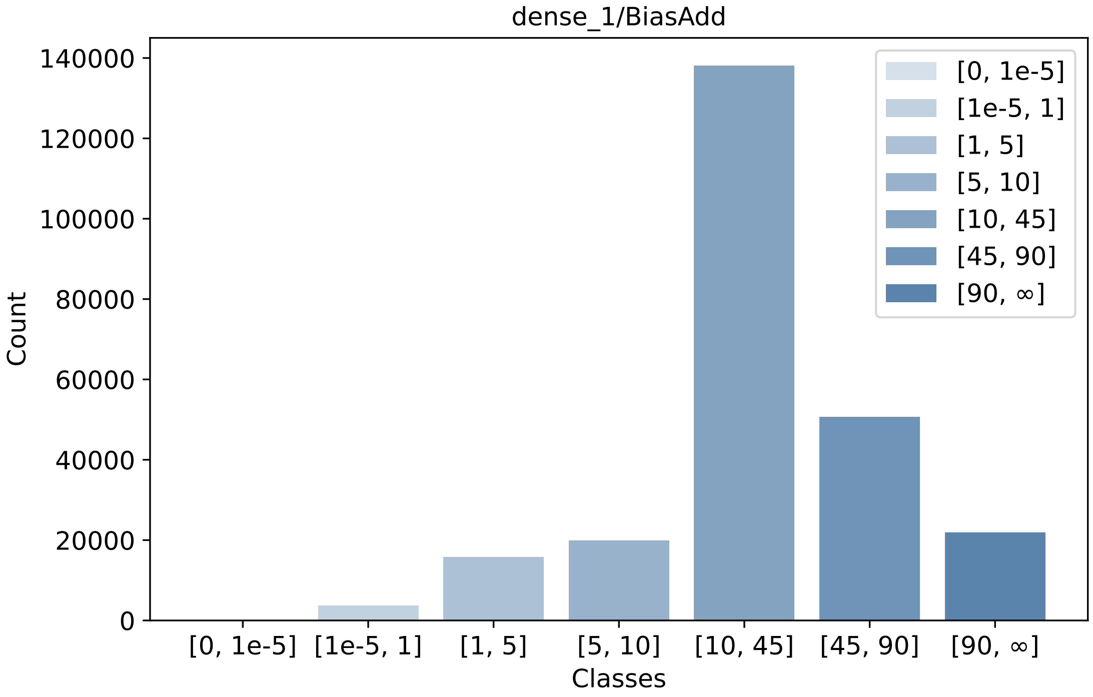

Figure 4.12: Last BiasAdd of the model

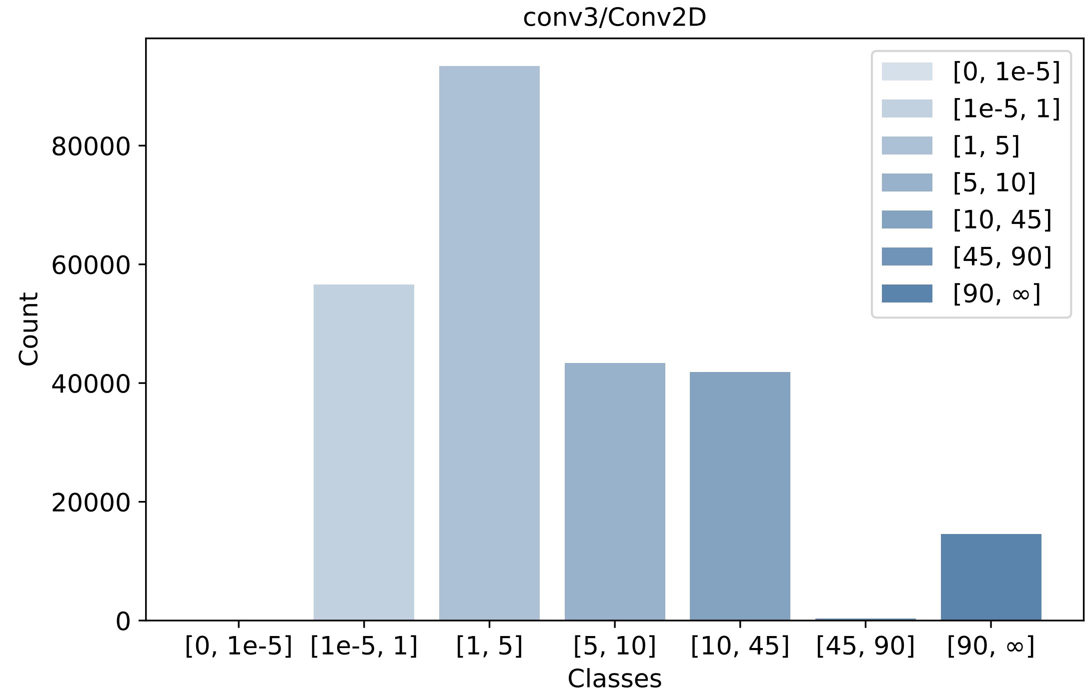

Figure 4.13: Third convolutional layer

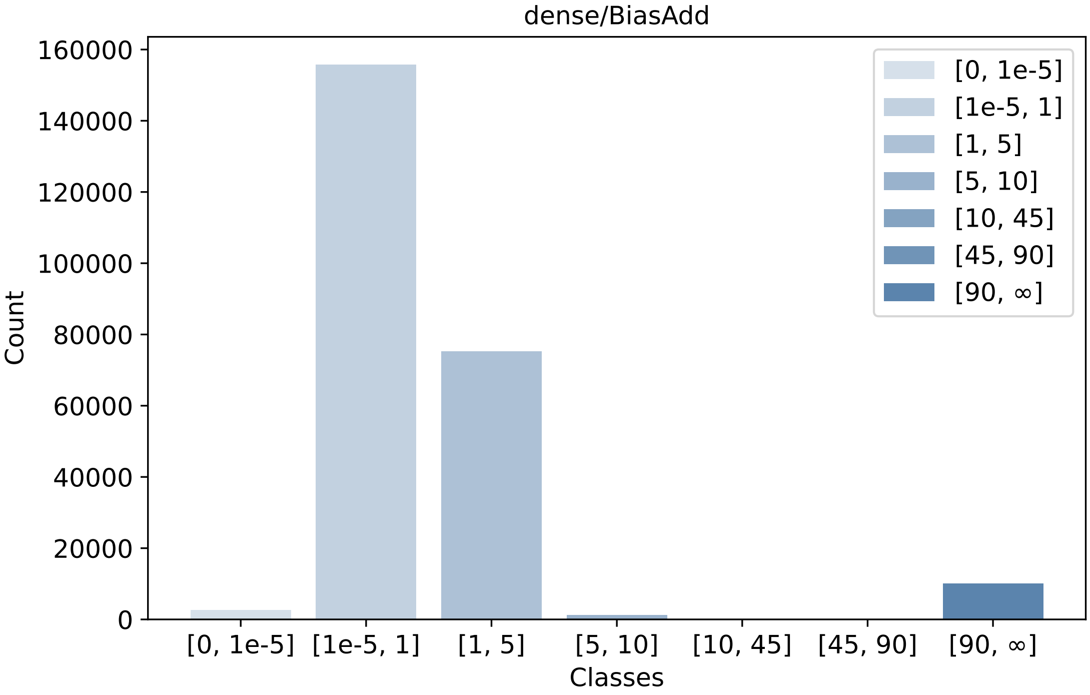

Figure 4.14: Penultimate BiasAdd of the model

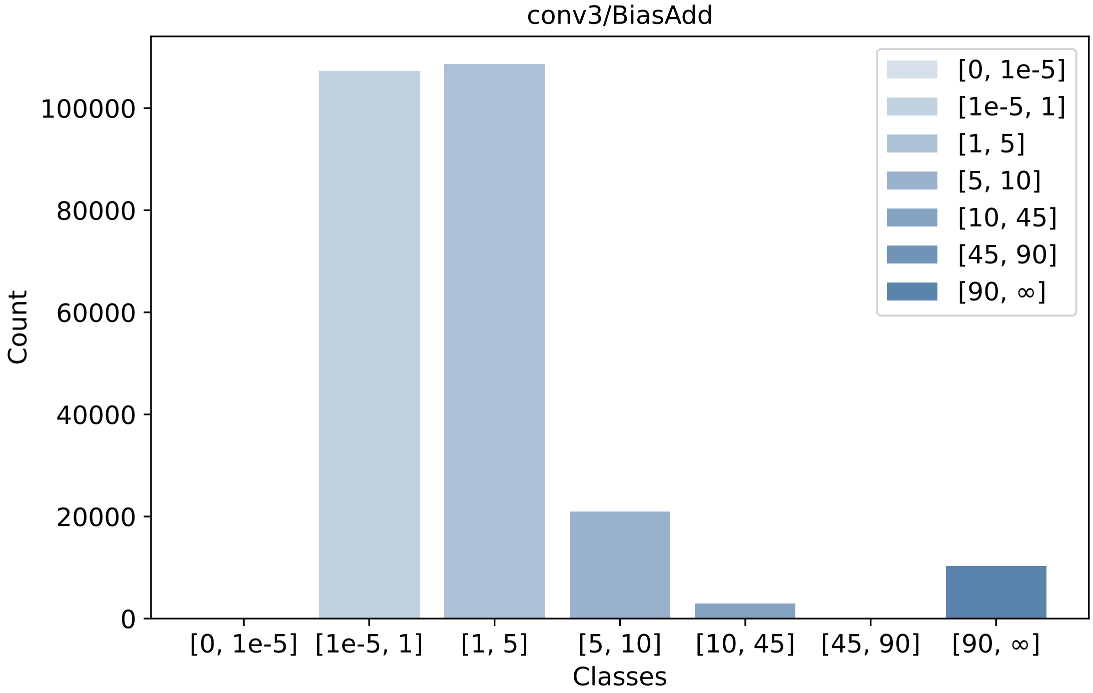

Figure 4.15: BiasAdd operator of third convolutional layer

4.2. Image Classification

One of the key task performed by CNNs is image classification, we live in a world were the diffusion of cameras and sensors has exploded in the last ten years and the request to have reliable ML models capable of correctly analyze and label images is more important than ever. For this reason we decided to target the family of Image Classification models as the second group of CNNs in our analysis, we selected two models that perform this task against two different datasets to experience a wide variety of conditions and to ensure that our conclusions are applicable to different types of models; the models selected for this analysis perform multi-class image classification

4.2.1. CIFAR-10 Model

The first image classification model analyzed is a custom network designed to work on CIFAR10 dataset.

The dataset has been developed by the Canadian Institute For Advanced Research and is one of the most commonly used to train machine learning models, used by many different researchers to support their works such as [56] and [57].

It is composed by 60 thousands images belonging to 10 different classes, equally split among them.

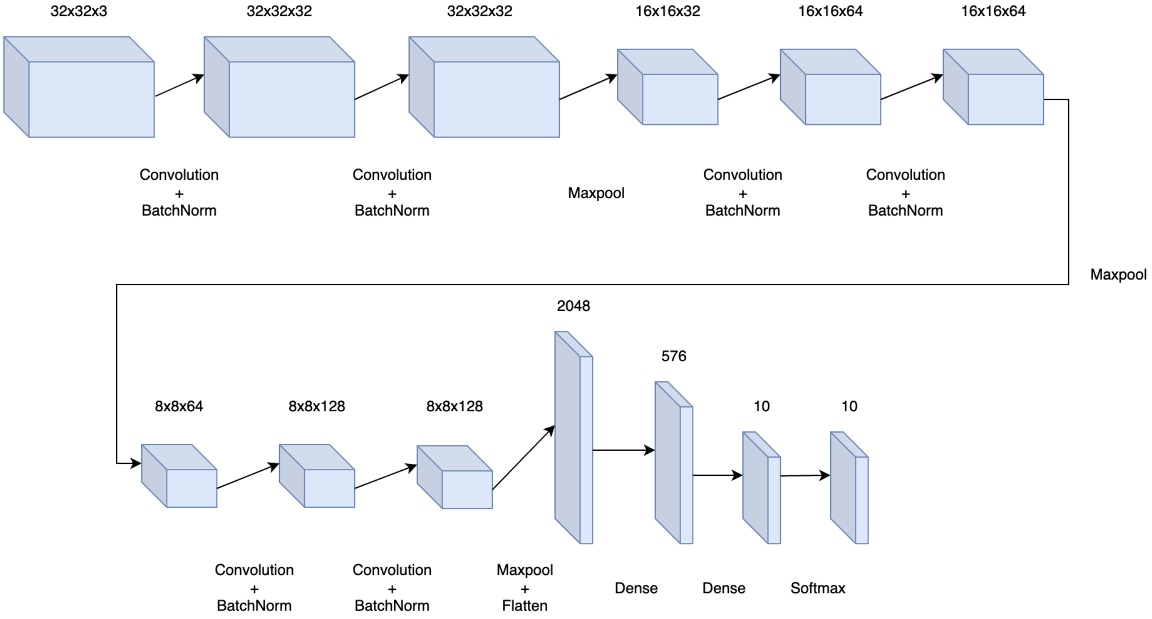

Figure 4.16: Cifar10 Recognition Model Scheme

Operators

A graphical representation of the model can be seen in figure 4.16.

There are six distinct blocks performing a convolution followed by a batch normalization, interleaved among these six blocks we have three maxpool layers used for dimensionality reduction. Finally we have two Dense layers followed by a Softmax operator that converts the produced logits into probabilities.

Given inputs of shape $(32 \times 32 \times 3)$ the model produces an output tensor of size 10 where each value v_i represents the probability that the input belongs to the i-th class.

Our Error Simulator allowed us to inject errors into all the Convolutional layers, all the Batch Normalization layers and the eight BiasAdd operators used by the six blocks of Convolution + BatchNormalization and the two Dense layers.

Input Selection

Having ten different classes of images we decided to select five different images of each class for a total of fifty images to test the model against, this decision ensures a broad

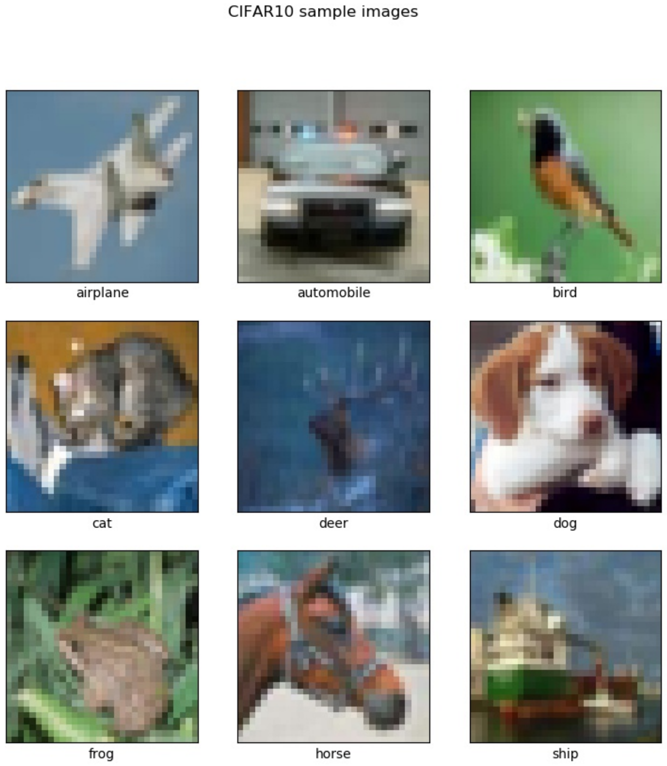

Figure 4.17: CIFAR10 Samples

variety of possible inputs both inter- and intra- classes. Some of the inputs selected can be observed in Figure 4.17, each image has been normalized before feeding it to the model.

Output Classification

Due to the type of task performed by the model we clearly have a correct answer that we are expecting the CNN to produce. For this reason the first immediate classification available for the results of the analysis consists in classifying the number of times a class has been wrongly selected as the correct one in case of a fault. This classification is both immediate in terms of meaning and useful at analyzing the effects that an SDC might have on any given operator.

The second classification we selected for this model is called Top2 and consists in selecting the two highest probabilities for the model and classify their difference. We decided to create ten classes of size 0.1 (i.e. [0.1,0.2]) and then two more, one for storing the cases where the difference is exactly 1 and one for when the probabilities are NaN instead of numbers.

Results

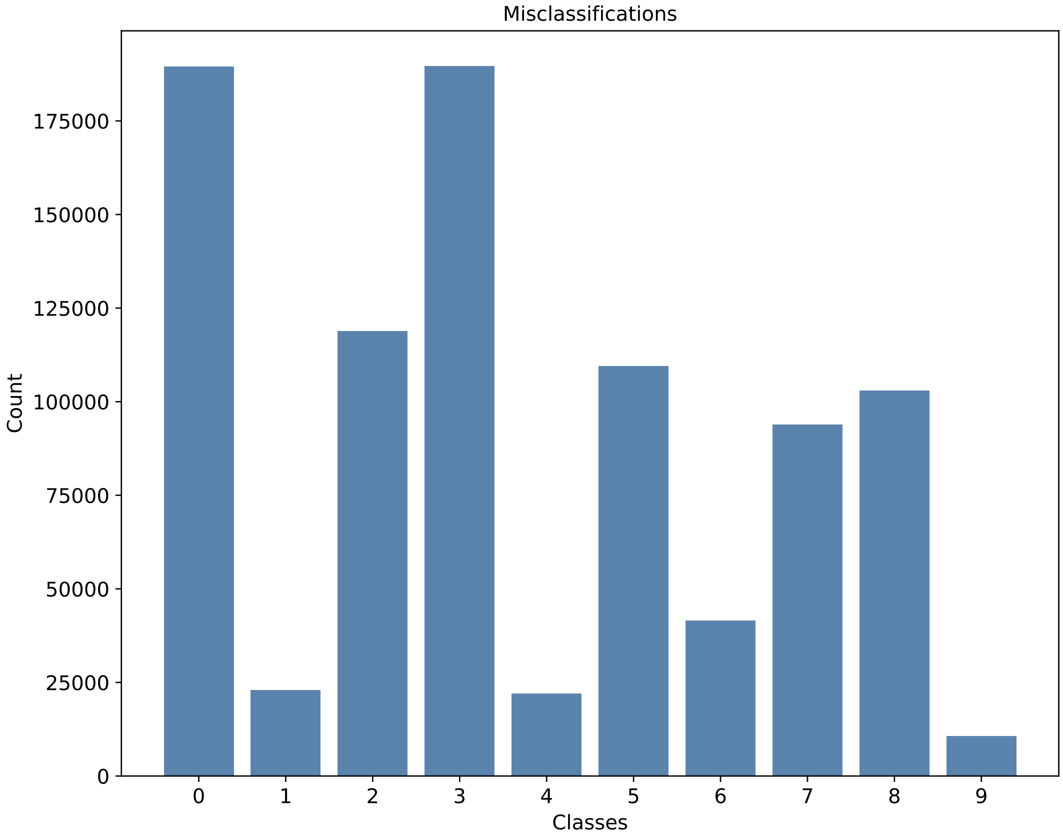

Figure 4.18: Cifar10 Model comprehensive misclassifications

From an observation of the results obtained we can draw some interesting conclusions:

- **Classes to look out for:** Looking at figure 4.18 which represents the overall misclassifications distribution observed in the model we clearly identify two classes with a higher count.
  The higher count of class 0 errors is the easiest one to explain, the class number is obtained by performing an argmax on the probability vector produced by the model. If this vector is entirely composed by NaNs, a situation not so uncommon as we will see later, then the result of the argmax is always 0. For this reason we see an high number of misclassifications toward class 0.
  The high number of times that the model misclassifies toward class 3 is instead more complicated to explain. It might be due to the small dimensions of the input, having such constraint on the placement of errors could lead to a position that is targeted

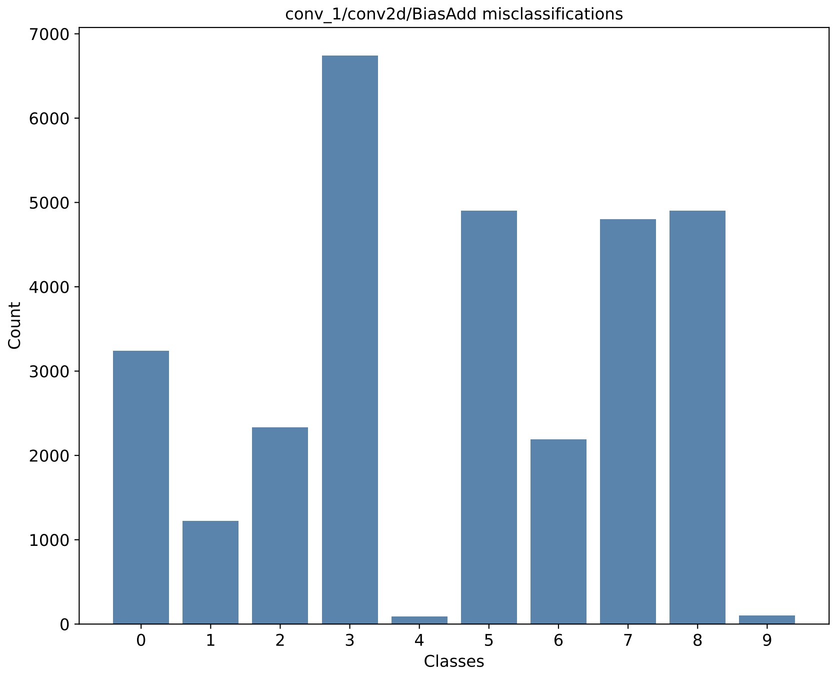

Figure 4.19: Conv1-BiasAdd misclassifications

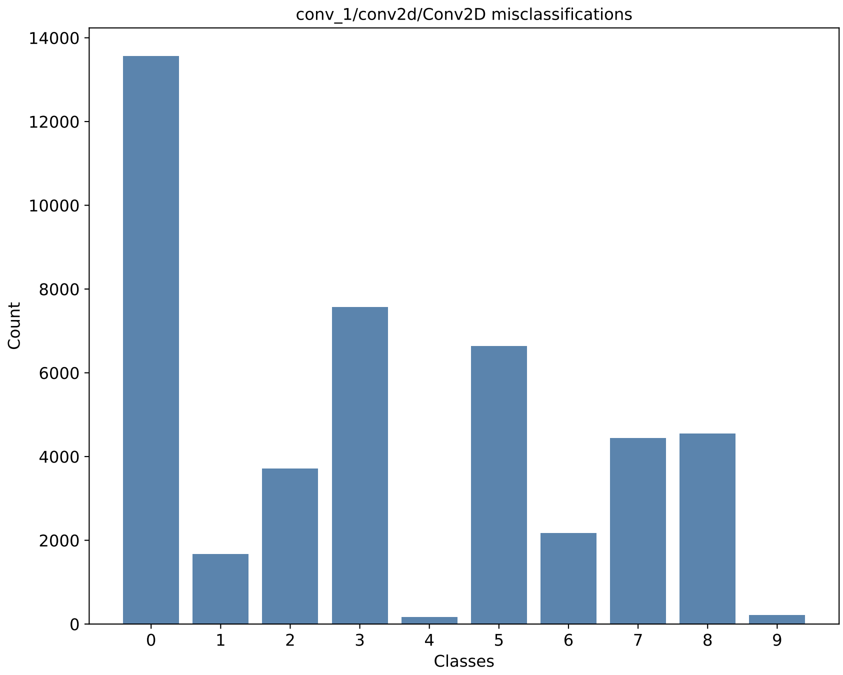

Figure 4.20: Conv1-Convolution misclassifications

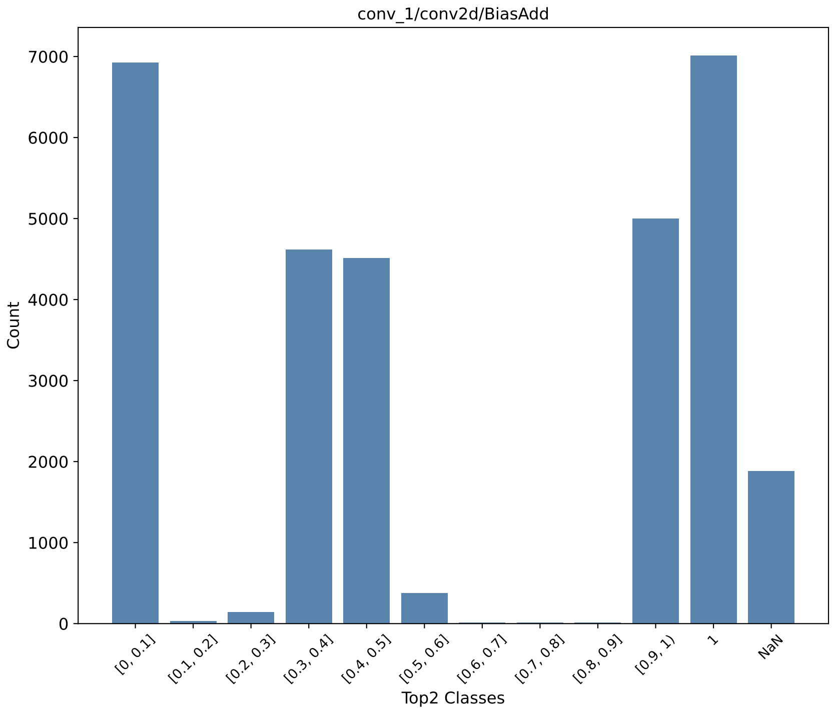

Figure 4.21: Conv1-BiasAdd Top2 distribution

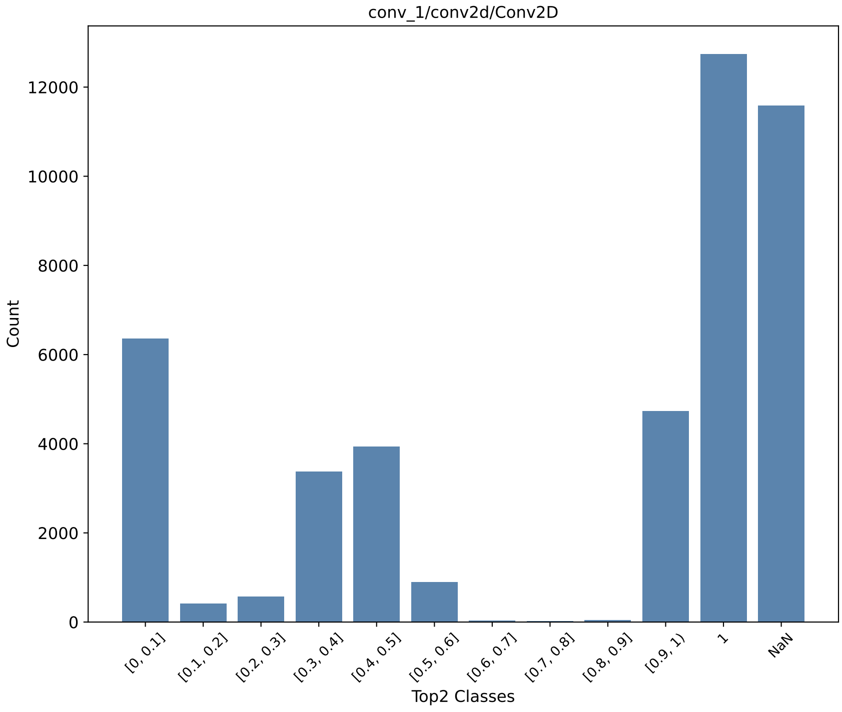

Figure 4.22: Conv1-Convolution Top2 distribution

more frequently and thus leading to a class of misclassifications more common than the others.
This result gives a crucial information, classes like 9 or 1 are correctly predicted in a faulty environment at a consistently higher rate than the others. It means that, whenever the model predicts one of those classes, the developer can be sufficiently sure of the result without needing any other tools to verify.

- **Different layers behave differently:** As we have already observed with the Steering Angle Detection models the type of layer experiencing an error greatly influences the effects it will have on the output.
If we consider the first convolutional layer in the model we can observe the effects of an error striking the BiasAdd operator, figure 4.19, or the Convolution operator, figure 4.20.
The first observation is that an error on the Convolution operator produces a misclassification more frequently, as we can see by observing the y axis range.
Moreover we see an high count of class 0 errors in the Convolution graph, this is due to the fact that an error striking a BiasAdd has a less destructive result with almost no presence of NaNs output, figure 4.21, unlike the more complex operator, figure 4.22.

- **No doubts mean doubts:** All the operators injected present a similar distribution when considering the Top2 indicator. Figure 4.23 and figure 4.24 show two very different operators with a similar behavior. The most common class in the Top2 indicator is the one where a single probability is maxed out at 1.
While this condition is desirable in a fault free environment it is of interest in a faulty environment. The explanation behind this behavior is that despite using normalized error models, the values in this network saturate rapidly. Most of the errors injected are selected through the maxpool layers and ultimately influencing the outcome of the net.

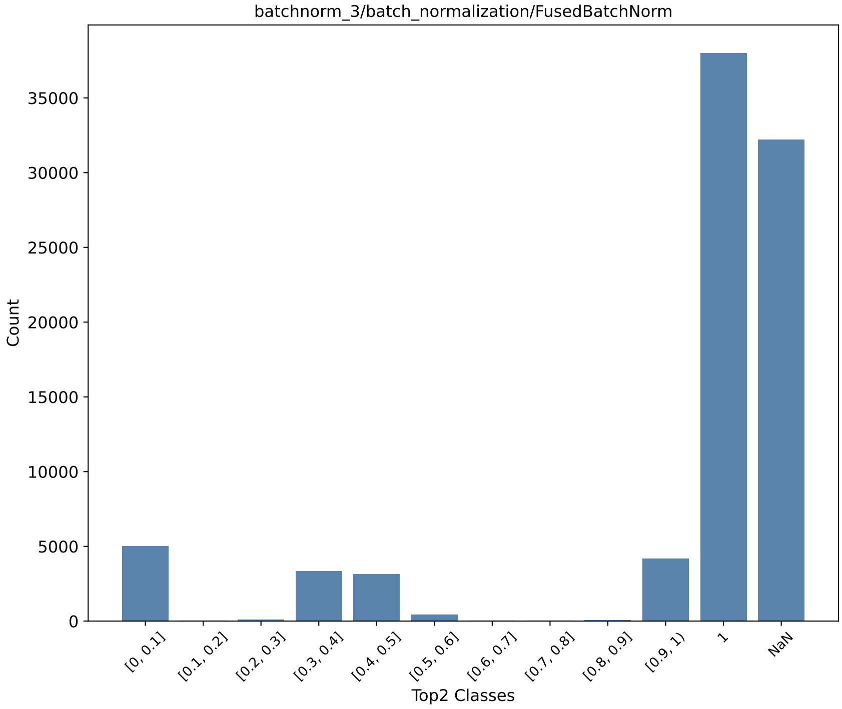

Figure 4.23: BatchNorm3 Top2 distribution

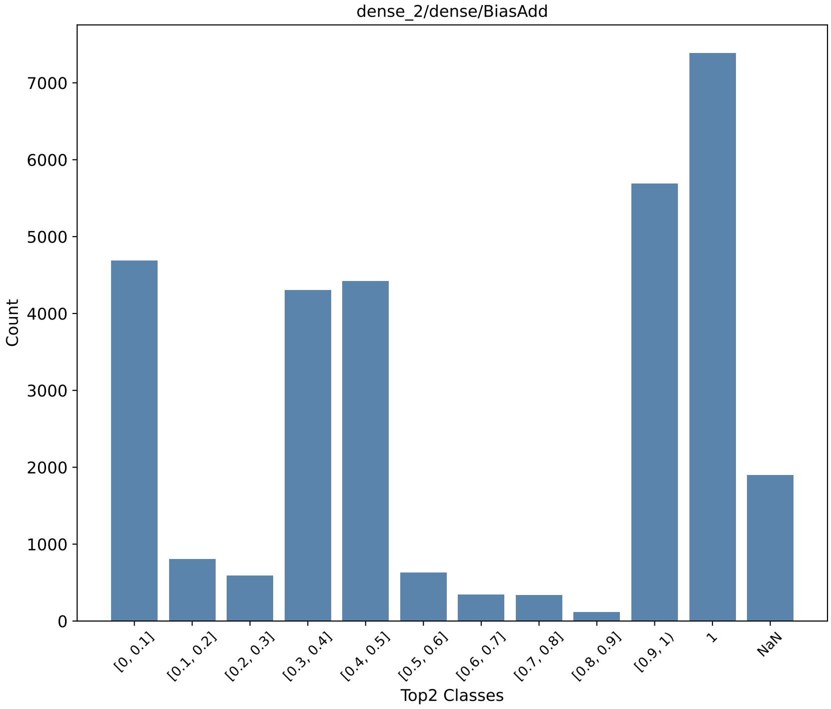

Figure 4.24: Dense2 Top2 distribution

4.2.2. Vgg11

The last model we selected for our analysis is part of the image classification group. It is a slightly simplified implementation of VGG11[58] that takes smaller images as inputs. The number 11 in the name represents the eleven weighted layers of the original model.

Operators

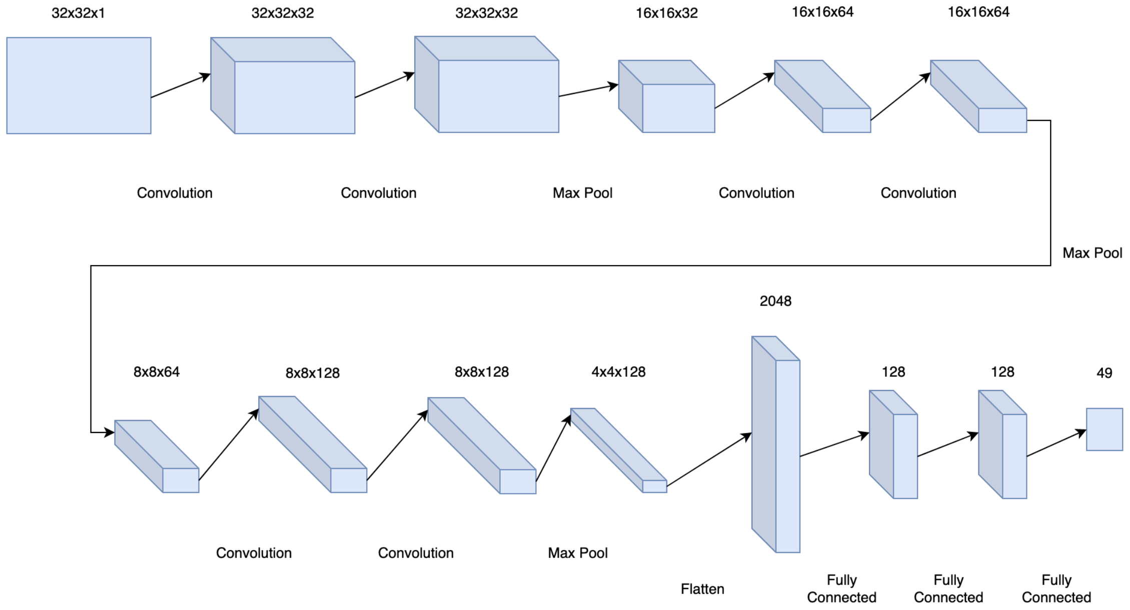

Figure 4.25: VGG11 Model Scheme

A graphical representation of the model can be seen in figure 4.25. The scheme is composed by six convolutional layers with small kernel sizes, (3×3), interleaved with two MaxPool layers. Following this first half we have a Flatten layer and three Fully Connected Layers producing a final output that represents the probabilities for each class. After both the MaxPool layers and all the Fully Connected we have dropout operators to obtain better results.

With our injector we were able to target all the Convolution operators, all the Add operators used by every layer and the Mul operators used by the dropout layers.

Input Selection

The input selection for this model is different than the previously described CIFAR10 dataset. We selected the German Traffic Sign Recognition Benchmark (GTSRB) [59], which is a collection of more than 50 thousands images belonging to 49 different classes,

Figure 4.26: GTSRB Samples

figure 4.26 shows a sample of the available images. We selected this particular dataset for multiple reasons, mostly because of the huge number of classes it represents. This is useful for our analysis because it allows us to observe the model working with diversified inputs, thus triggering many corner cases.

Output Classification

While dealing with a different dataset than the previously discussed CIFAR10 the metrics used to classify the output of the model are the same. We will analyze the misclassifications and the Top2 metric.

Results

The results obtained from the error simulation campaign of this model highlight some interesting conclusions.

- **Not all models react equally:** despite using the same set of operators the behavior of VGG is completely different than what was observed on Cifar. The first

main difference observed is the prevalence of NaN results. Almost all the instances injected resulted in a probability vector entirely composed of NaNs. Since the operators used are the same and the overall mean and standard deviation of the data used is comparable to CIFAR the reason behind this behavior must be searched in the structure of the model itself. Being a very deep model might be a double edged sword: on one side it gives the model time to recover from an error on the other side any error, especially those affecting convolutional layers, has time to spread among neighboring values.
Nonetheless some of the instances produce numbers that are not NaNs. As seen in figure 4.28 and 4.27 the Top2 distribution for those instances shows that the prediction in case of errors is almost always decided by small differences in the produced probabilities. Combining these two observations we can say that this model suffers more from the effects of faults.

- **Misclassifications' behavior follows the rules:** Looking at the misclassifications produced by the model, figure 4.29 and figure 4.30, we find the same behavior observed previously. Overall the misclassifications are spread among all the classes with a couple of them being more targeted than the others and some being nearly always correctly predicted. Overall the same conclusions we have drawn for the previous model can be applied in this case.

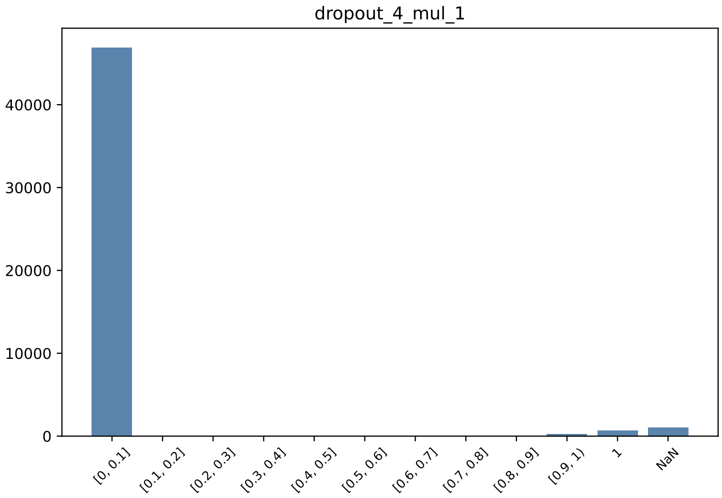

Figure 4.27: Mul Top2 distribution

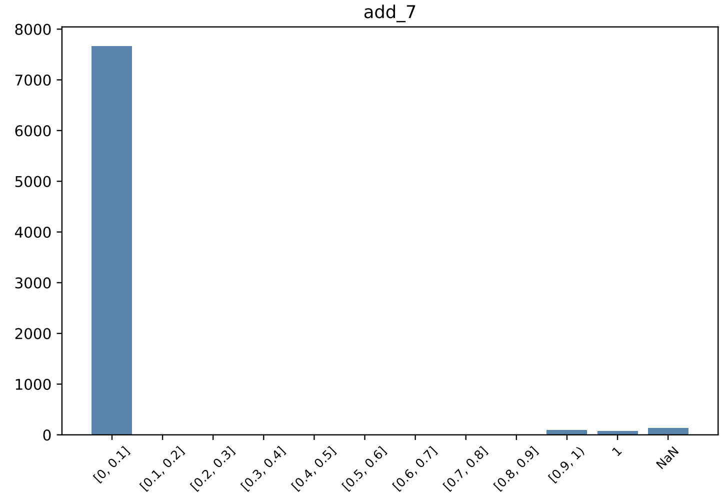

Figure 4.28: Add Top2 distribution

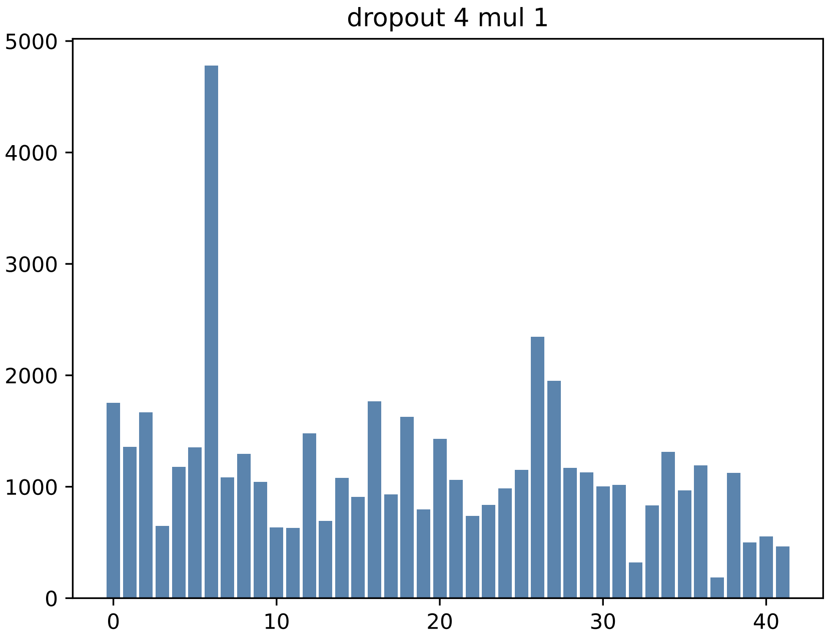

Figure 4.29: Mul misclassifications

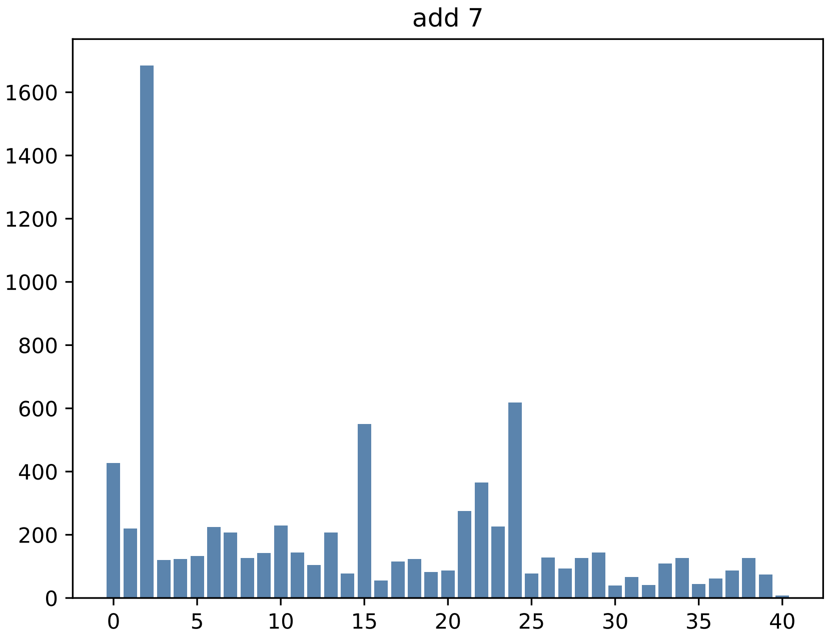

Figure 4.30: Add misclassifications

4.3. Timing Overhead

It remains to discuss the timing overhead introduced by the error simulator.
We thus executed each model on a 2018 MacBook Pro with a 2,3 GHz Quad-Core Intel Core i5 and recorded the time taken for the inference time. Then we run the error simulator for each tuple image and layer, also recording the inference time.
Table 4.2 shows the average execution times recorded. While it is clear that the overhead introduced is not negligible, the recorded times are still low. Moreover, running the error simulator on faster hardware would result in better performances.
With the times presented in Table 4.2, a campaign performing five thousand error simulations for each tuple image and layer would take approximately 23 minutes in the worst case and only 6 minutes in the best scenario.

Model	**Plain Average Time (s)**	**Model+Simulator Average Time (s)**
PilotNet	0.039	0.125
Comma's AI	0.019	0.074
Cifar10	0.016	0.172
Vgg11	0.085	0.272

Table 4.2: Execution times comparison

In this chapter, we presented the error simulation campaign. We discussed the reasons behind the choice of each CNN model and provided an in-depth analysis of the obtained results. We highlighted the process of finding the most vulnerable layers starting from the outputs of CLASSES. Finally, we discussed the overhead introduced by the framework.

5 | Hardening CNNs

The reliability analysis of a CNN is a complex problem due to the many layers that compose any model and the consequently large amount of information produced by an error simulation campaign.

From a literature review, the only methodology that tackles this problem we found is [60], which we will analyze in Section 5.2. The main problem with this approach is that it only uses fault injection, both at the architectural and high levels. While the former is a standard methodology, the latter represents a simplification of the model, and thus, the results obtained might not be the most accurate possible.

For this reason, we propose a methodology for the robustness analysis of CNNs based on error simulation, which is a far more precise technique at the application level. As explained in Chapter 3, the error models used in the simulation phase are generated from a fault injection campaign to be as authentic as possible.

For this reason, we decided to build our methodology based on the observations made at both the application and architectural levels. In this chapter, we will discuss the proposed metric in all its components, and present the results obtained against the CNNs selected in Chapter 4.

5.1. CNN Robustness Metric

CLASSES allows performing reliability analysis at the layer level. For this reason, the robustness metric proposed focuses on layers. Moreover, it accounts for both the application and architectural levels. We decided to maintain the concept of the Layer Vulnerability Factor but changed its definition to adhere to the usable/unusable paradigm rather than the correct/incorrect one.

We first perform an error simulation campaign on each operator, then the different values that compose the final metric are produced and finally, we aggregate them into a single value. The values are:

- **Operator's susceptibility:** The first value produced in the analysis is the susceptibility of each operator. This metric quantifies the information produced at the

architectural level. In particular, it identifies how many times an error striking the operator under analysis results in a corrupted output or not. It is independent of the model under analysis and is an intrinsic property of the operator.
The equation to produce this value is the following

$$s_{err\_i} = \frac{\#errors_i}{\#faults_i} \tag{5.1}$$

$\#errors_i$ is the number of times that the output was corrupted while $\#faults_i$ is the total number of errors injected.

- **Layer Vulnerability Factor:** The second component is the Layer Vulnerability Factor. Introduced in the literature as:

$$LVF_i = \frac{\#sdc_i}{\#faults_i} \tag{5.2}$$

where $\#sdc_i$ represents the number of incorrect results over a fault injection campaign with size $\#faults_i$.
We changed its definition to align the metric to the usability paradigm. Moreover, we decided to apply it at the application level with the results obtained from the error simulation campaign. This decision is due to the need to measure robustness both at the application and architectural levels.

$$\widetilde{LVF_i} = \frac{\#\overline{u_i}}{\#faults_i} \tag{5.3}$$

We thus replaced $\#sdc$ with $\#\overline{u_i}$, that is the number of faults causing unusable results instead of incorrect results and The corresponding robustness of the layer is:

$$R_i = 1 - \widetilde{LVF_i} = 1 - \frac{\#\overline{u_i}}{\#faults_i} \tag{5.4}$$

- **Layer robustness:** The main advantage of CLASSES with respect to other frameworks is the ability to gather information at the application and architectural levels. We thus combined these two sources into a unified metric to measure the robustness of each layer of a given CNN model.
The equation to describe the overall reliability of the CNN against fault affecting

operator i is computed as:

$$R_i = \left(1 - s_{err\_i}\right) + s_{err\_i} \cdot \frac{\#u_i}{\#sim\_errs_i} \tag{5.5}$$

The first addend represents the percentage of cases where the fault did not cause an error on the output, and the second addend represents the percentage of cases where the fault led to a final usable result.

- **Timing weight:** The last parameter to consider is the ratio between the profiled execution time of each operator and the whole CNN.

$$T_i = \frac{\Delta t_i}{\Delta t_{CNN}} \tag{5.6}$$

Intuitively this gives information on the percentage of the execution time allocated for each operator.

We can combine equation 5.5 and equation 5.6 to produce the overall CNN robustness, defined as:

$$R_{CNN} = \sum_{i \in CNN} R_i \cdot T_i \tag{5.7}$$

The robustness of the whole CNN is the sum of each operator's robustness weighted by its allocated execution time.

5.2. Usability aware hardening strategy

In the considered scenario the CNN under analysis can be expressed as a series of operators sequentially executed and accelerated on a GPU. This interpretation allows for multiple hardening techniques to be applied, such as triple modular redundancy, complete duplication of the net, and more.

In our work we propose Duplication with Comparison (DWC) performed at the operator level. This technique requires creating an exact copy of an operator executed in parallel to the original. The results obtained are compared by a custom and newly inserted checking operator, also accelerated on GPU. Using DWC we are capable of detecting possible errors in the output of a single operator in the middle of the inference phase. The developer can then introduce appropriate methods to correct the error.

Considering what we observed in the previous chapter there are multiple immediate approaches to DWC that might not be optimal, let's analyze two of them:

- **Full duplication, compare all operators:** One of the first approaches that we could take would be to duplicate all the available operators and add one checking layer to all of them. While this solution does guarantee the certainty of detecting an error it might be redundant and time wasting. As we have seen in the previous chapter, not all operators have a strong influence on the output of the model. Duplicating those leads to an increase in the overall execution time without having a proportionate increment of robustness.

- **Full duplication, compare the output:** One way to reduce the execution time of the duplicate model would be to remove all the intermediate checking layers and compare only the final outcome of the model. This method ensures a complete robustness of the model and reduces the time of the inference phase with respect than the previous approach, however it still duplicates layers with a low impact on the CNN's output.

Another interesting duplication scheme is presented in [60]. The authors propose a metric called Layer Duplication Impact that measures the probability of a layer improving the CNN reliability after duplication. The equation to produce this value is the following. It is the ratio between the number of instructions that are executed in the layer and the product between the execution time of the layer itself and the Layer Vulnerability Factor. The hardening scheme proposed consists of sorting all the layers based on their LDI and then duplicating them in order. The results of this approach will be compared with our technique in the next section.

Given the observations we just made we decided to perform a complete exploration of the duplication space in order to find the best solutions for the four CNNs under analysis.

5.2.1. Hardening the CNNs under analysis

Algorithm 3: Checking layer

```
Input: shape
Output: boolean result
begin
    tensor_1 = tf.random.uniform(shape, dtype=tf.float32, seed=rand())
    tensor_2 = tf.random.uniform(shape, dtype=tf.float32, seed=rand())
    equality = session.run(tf.math.reduce_all(tf.equal(tensor_1,tensor_2)))
    return equality
end
```

To create the following results we executed the inference phase of the CNNs on a target GPU with time profiling. Then we selected, for each operator, the appropriate checking layer. This choice was performed considering the shape of the output tensor of each layer.

Then we created a custom TensorFlow project that generates two random tensors of the chosen shape and performs the comparison between the two, as seen in algorithm 3.
Duplicating a layer introduces a performance degradation defined as

$$PD_i = \Delta t_i + \Delta t_{c\_i} \tag{5.8}$$

which is the sum of the execution time of the layer and of its checker. Its robustness is set to 1.

$$R_i = 1 \tag{5.9}$$

After recording the comparison time we calculated all the possible combinations of the operators list where each of them was either duplicated or not.
Identifying as H_CNN the set of hardened layers and as N_CNN the set of those that are not the robustness of each possible solution is defined as

$$R_{SH\_CNN} = \sum_{i \in N\_CNN} R_i \cdot \frac{\Delta t_i}{\Delta t_{CNN}} + \sum_{i \in H\_CNN} \frac{\Delta t_i}{\Delta t_{CNN}} \tag{5.10}$$

The overall performance degradation of the hardened solution is defined as

$$PD_{SH\_CNN} = \sum_{i \in H\_CNN} (\Delta t_i + \Delta t_{c\_i}) \tag{5.11}$$

5.3. Results

In this section, we will describe the results obtained by applying the discussed hardening scheme to the target CNNs. We will highlight different solutions observed and discuss the information extracted from these graphs.

5.3.1. PilotNet

The first CNN under analysis is PilotNet, the steering angle detection model. With 16 injectable layers the space of possible combinations is composed by 65 thousands possible combinations. Looking at figure 5.1 we see how the model without any duplicated layers has a robustness of 0.965. This value is already high and is intuitively in line with what we have seen in section 4.1.1. The fact that the layers closer to the output were the most troublesome means that the overall robustness of the net is kept high by the initial layers. Nonetheless, we can aim to improve PilotNet's reliability by duplicating some layers.
In figure 5.1 we can identify two different set of points, the red ones are all the possible

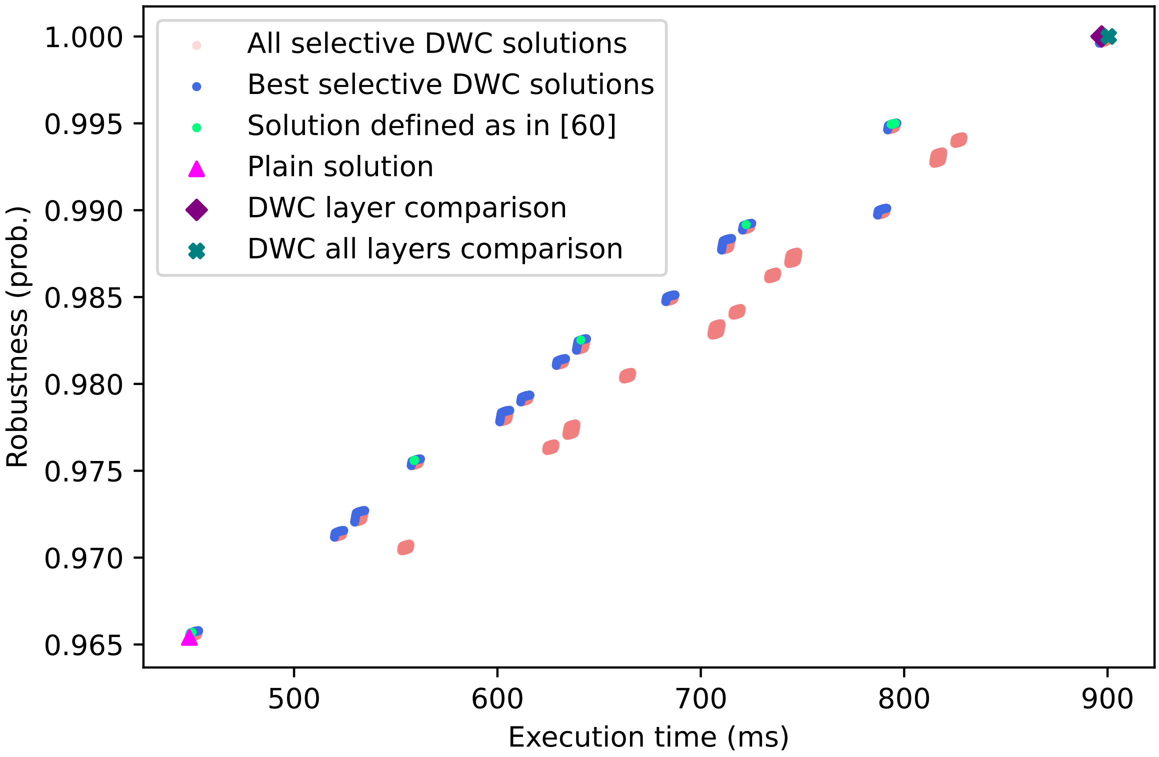

Figure 5.1: PilotNet

DWC's combinations of the model which are dominated according to Pareto's definition, the blue ones instead make up the Pareto front of this scenario. We can see how, for any blue point, there is no other solution in which the robustness increases without an increment in the execution time.

The purple rhombus and the green cross in the graph represents the two conditions we discussed at the beginning of section 5.2, the former being full duplication with comparison of the output and the latter full duplication with comparison on all layers. As seen in the image, both solutions have a robustness of 1, which is correct by definition. However, what is interesting to note is that the execution time between the two cases doesn't vary too much. Considering how much time is spent in the inference phase we find that the five convolutional layers take 90% of it. Instead the time spent by the checking layers does not take more than 1% of the inference time. This disparity means that the time increment of any solution with respect to the original model will be decided almost entirely by which layer will be duplicated.

All the blue points are local optimal solutions of the problem and a developer interested in increasing the robustness of its model should specify a time threshold and select the result with the highest robustness with an acceptable execution time.

5.3.2. Comma's AI Steering Wheel Model

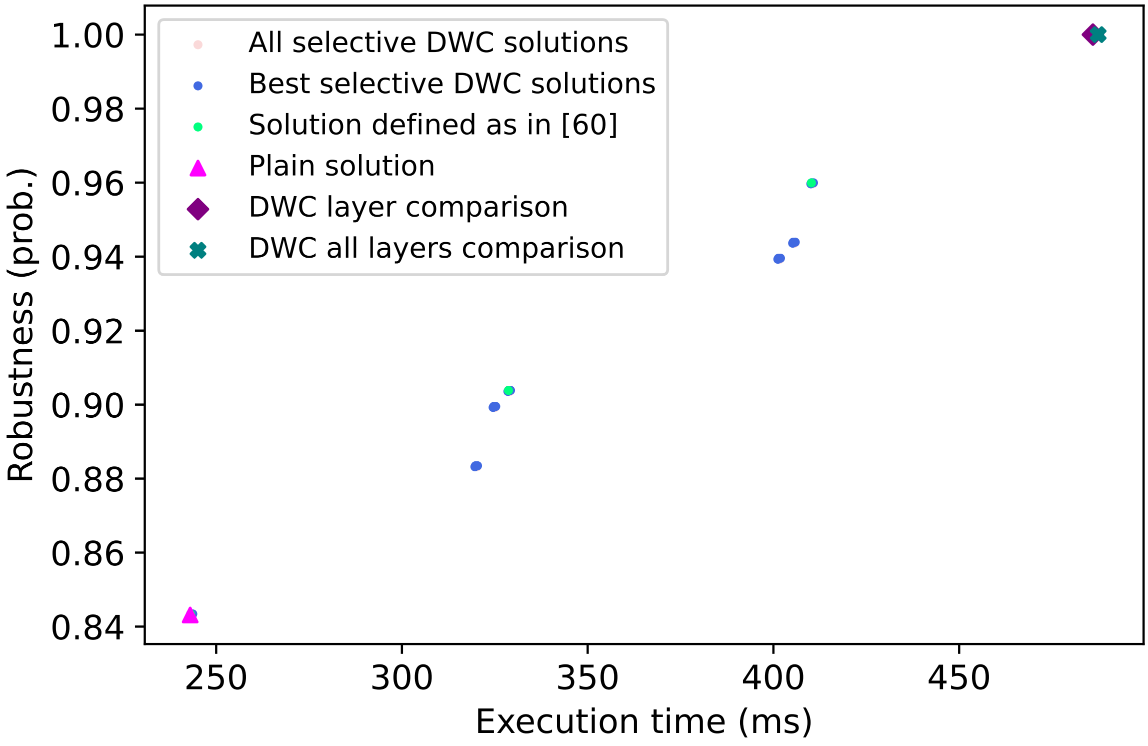

Figure 5.2: Comma's AI

Comma's AI model is the second CNN considered in our analysis. Due to a much lower number of injectable layers, the produced result, figure 5.2, is composed almost entirely of Pareto optimal solutions. Like the previous model, the entirety of the time spent by this CNN is due to the convolutional layers, with the checking layers being almost irrelevant. It is interesting to analyze the green points in the graph, representing the solution proposed in [60]. Due to the low number of layers in the model, the optimal results are almost identical between the two approaches. A considerable difference can be observed instead in CNNs with a higher level of depth. Figure 5.4 represents a clear example where the number of optimal solutions produced by our methodology is far greater than the ones identified by the other approach.

5.3.3. CIFAR-10 Recognition Model

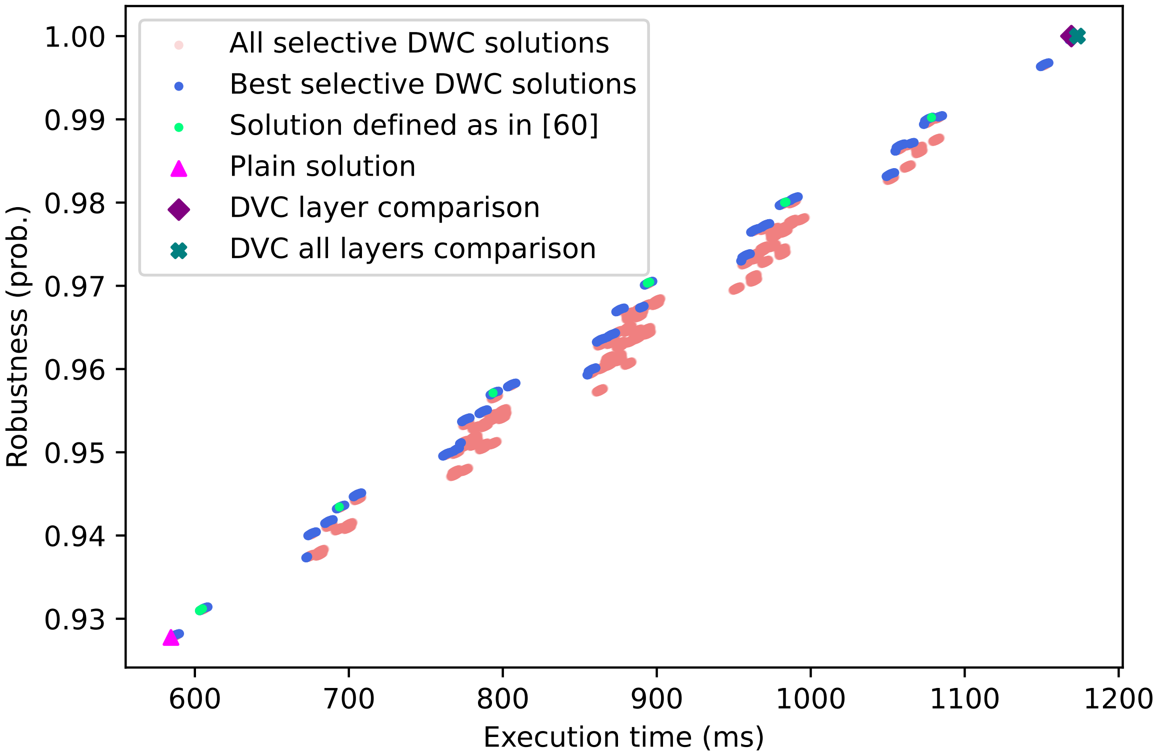

Figure 5.3: CIFAR-10 model

With CIFAR-10 recognition model we can observe a behavior that was already slightly visible in PilotNet. It is possible to highlight five different groups of solutions clearly separated one to the other. The existence of these five groups can be explained by looking at the individual execution times of the layers, table 5.1, it is immediately clear how the convolution operators are the most time demanding ones. This means that depending on how many of them are duplicated in a solution the execution time will change, with 600ms being the baseline we have 700ms with one convolutional layer duplicated, 800ms with two and so on.

Like the two other models observed the checking layers' execution time is much lower than the execution time of the layers, meaning that the two immediate solutions of full duplication with comparison have a nearly identical recorded time.

Instance	Execution time (ms)
Convolution 1	99,8
Convolution 2	93,4
Convolution 3	99,6
Convolution 4	87,4
Convolution 5	88,4
Convolution 6	94,0
BatchNorm 1	18,1
BatchNorm 2	0,5
BatchNorm 3	0,5
BatchNorm 4	0,5
BatchNorm 5	0,8
BatchNorm 6	0,5
Dense 1	0,2
Dense 2	0,16

Table 5.1: Cifar-10 layers' execution time

5.3.4. VGG 11

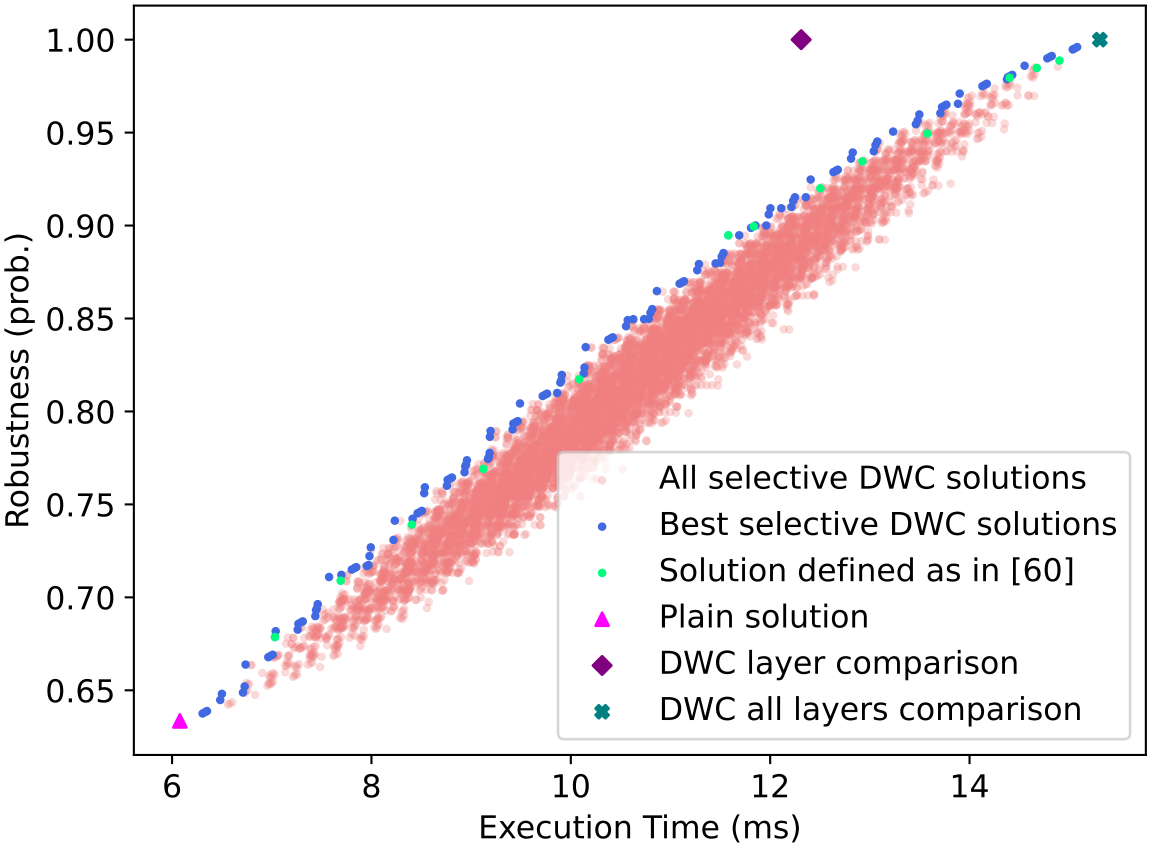

Figure 5.4: VGG-11 model

The analysis of VGG-11 model is different than the previous models. Looking at figure 5.4 it is already clear that the observations we made are not applicable to this model.
The first main difference is a much lower execution time with a baseline of 6ms, a value 100 times lower than the other image recognition model. It means that each layer has almost the same weight in terms of execution time. It leads to a distribution of the optimal solutions that is not in the form of groups but instead an increasing monotone curve. Moreover, a lower execution time means that the checking layers have a higher impact on the timing of the proposed solutions. We can observe this phenomenon by analyzing the "DWC with comparison on last layer" and "DWC with comparison at all layers" solutions. The timing of the former is highly lower than the latter, with a difference of more than 2 ms. Checking all the intermediate outputs increases the execution time by almost 25% than comparing only the outcome of the model.

5.4. Conclusions

Using the results of the error simulation campaign, we were able to develop a Layer Vulnerability Factor that defines the robustness of a given Convolutional Neural Network. This metric considers the execution time of each layer of the network, its intrinsic probability of producing a faulty output if a fault occurs, and its effects on the outcome of the model.
We then discussed how to use the metric to create a hardening solution that, through the duplication of a set of layers, increases the overall robustness of the model while keeping the execution time under a certain threshold. As we defined in section 1.1, our primary goal was to provide a robust and precise technique for the reliability analysis of any given CNN. This task has been completed with the results of this chapter, where we demonstrated that using CLASSES, a developer is able to easily target an ML model and discover its most vulnerable layers. A knowledge that can be used to develop a hardening strategy that greatly improves the robustness of the model. From the analysis of the graphs produced, we can make some interesting observations.

- **Some models are intrinsically robust.** Looking at figure 5.3, we see how the robustness of the base model is already really high, almost 0.93. It is an intrinsic property of this particular model due to how it was built, and it does not depend on the depth of the net itself. Looking at figure 5.4, we have a deep model with much lower base robustness or, looking at figure 5.1, the opposite, a model with a low number of layers and high base robustness. It means that, in order to design a reliable model, the developer must carefully choose how the net is constructed and then use our technique to improve the base result.

- **Great improvements might come at low costs.** Figure 5.3 is a clear example of how improvements might not be expensive. Reaching maximum robustness of 1 would require an execution time of approximately 1200 ms. We can reduce the execution by nearly 200ms, almost 17%, while keeping the robustness over 0.98.

In this chapter, we described our proposed approach for the robustness analysis of Convolutional Neural Networks. We discussed each component of the metric in-depth and provided explanations of how we selected them. Finally, we analyzed the results of the methodology applied to the four CNNs selected.

6 | Conclusions and future developments

In recent years we saw a widespread increment in the usage of Convolutional Neural Networks for perception functionalities, both in safety-critical systems and not. Among all the safety-critical ones, we have discussed the importance of Autonomous Driving Systems, a group that is gaining even more relevance with newer models of cars that are every day more technological than the previous ones.

This improvement, coupled with the spread of GPUs among publicly available systems, saw an exponential increment in performance, making ADS one of the core components of any vehicle. An increment of performance and diffusion came with the pressing need for a precise and robust reliability analysis technique. Creating such a technique was the primary goal of this thesis.

We thus selected four different CNNs, all relevant to the tasks performed by ADSs, and executed a complete error simulation campaign on each of them. Starting from the results of this campaign, we created a reliability analysis metric that calculates the robustness of the given CNN.

The proposed metric works by calculating the intrinsic reliability of each layer of the network starting from the results of the architectural error injection campaign. It then uses a custom oracle to weight each layer based on its effects on the outcome of the network.

The granularity of our approach allowed us to propose a hardening technique that uses the produced metric to identify a set of solutions where the robustness of the model is increased while keeping the execution time under a certain threshold. By proposing this hardening solution, we demonstrate that our metric can be used to assess the reliability of a CNN and also identify possible solutions given a duplication scheme of the model's layers. While building this metric, we also improved the available Error Simulator tool making it more robust in terms of both technologies used and usability for the end-user. The improvement of the framework was the secondary goal of the thesis.

We can affirm that both goals were successful. While working on the thesis, we identified some interesting future developments that we want to discuss.

6.1. Future developements

Possible future developments of this thesis can be divided into two different categories. The first one contains those improvements regarding the error simulator framework, both in terms of usage and technologies. The second one consists of the developements of the proposed metric.

6.1.1. Error Simulation Framework

- **A better analyzer:** The process of analyzing the corrupted tensors produced during the architectural fault injection campaign is highly dependent on a human factor. In particular, the most crucial aspect is the recognition of spatial patterns. We performed the analysis manually, identifying the most common spatial patterns. Then we developed a custom script that automatically recognizes and characterizes those patterns. This approach is time demanding and prone to errors since the user tasked with the recognition of the patterns might miss some of them.
  For this reason, an interesting development would be to automatize the recognition process. A possible approach would be to apply clustering algorithms to the corrupted tensors considering only the positioning of the errors and not the values.
  We already begun exploring this possibility paired with the usage of Auto Encoders to reduce the tensors' dimensionality.

- **TensorFlow 2:** As we discussed in section 3.4 we already developed a working error simulator that targets Keras models. An interesting development would be to integrate it inside the simulation framework. Moreover we think it would be interesting to evaluate the performances and the results of the methodology described in this thesis if performed using a newer version of TensorFlow.

- **Target multiple GPUs:** With the introduction of cuDNN, we ensured that the scripts used in the architectural error injection phase are executable on any NVIDIA GPU. The next step for improving the tool requires adapting it for different GPU architectures such as Intel or AMD.
  Doing so requires introducing a new fault injection tool that targets the required architecture. The part of the framework executed after the error injection campaign is the same and doesn't require any modification.

6.1.2. Hardening CNNs

- **Introduce new metrics:** The proposed Layer Vulnerability Factor describes a ratio between the robustness of the model and its execution time. We chose these two metrics because of the working scenario considered. However, it might be of interest to link the robustness of the CNN with other metrics such as power consumption or area of the chip. One possible future development of the proposed work would be introducing different working scenarios with different metrics used to build the Layer Vulnerability Factor.
- **Expand the number of CNNs analyzed:** In this thesis, we focused our analysis on four different Convolutional Neural Network models due to the limited time available. While we demonstrated the effectiveness of our approach, we believe that the analysis should be expanded to other commonly used CNNs. In particular, it would be interesting to target bigger models to observe the effects of faults on networks with very deep structures.

Bibliography

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

[2] Mark Campbell, Magnus Egerstedt, Jonathan How, and Richard Murray. Autonomous driving in urban environments: Approaches, lessons and challenges. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 368:4649–72, 10 2010. doi: 10.1098/rsta.2010.0110.

[3] J3016_202104: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles - SAE International. https://www.sae.org/standards/content/j3016_202104.

[4] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750, December 1996. ISSN 1558-1578. doi: 10.1109/23.556861.

[5] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv:1502.03167 [cs]*, March 2015.

[6] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, 117(4):500–544, August 1952. ISSN 0022-3751.

[7] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical Evaluation of Rectified Activations in Convolutional Network. *arXiv:1505.00853 [cs, stat]*, November 2015.

[8] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR. URL `https://proceedings.mlr.press/v15/glorot11a.html`.

[9] Brownlee Jason. A gentle introduction to the rectified lin-

ear unit (relu). https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/, 2019. (Accessed on 15/01/2022).

[10] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2. Chapter 4: Artificial Neural Networks.

[11] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. URL https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[12] François Chollet et al. Keras. https://keras.io, 2015.

[13] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient Primitives for Deep Learning. *arXiv:1410.0759 [cs]*, December 2014. URL http://arxiv.org/abs/1410.0759. arXiv: 1410.0759.

[15] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast training of convolutional networks through ffts, 2013. URL https://arxiv.org/abs/1312.5851.

[16] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. page 7.

[17] Andrew Anderson, Aravind Vasudevan, Cormac Keane, and David Gregg. High-performance low-memory lowering: Gemm-based algorithms for dnn convolution. In *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 99–106, 2020. doi: 10.1109/SBAC-PAD49847.2020.00024.

[18] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23 (4):14–19, 2003. doi: 10.1109/MM.2003.1225959.

[19] D. Binder, E. C. Smith, and A. B. Holman. Satellite anomalies from galactic cosmic

rays. *IEEE Transactions on Nuclear Science*, 22(6):2675–2680, 1975. doi: 10.1109/TNS.1975.4328188.

[20] Timothy C. May and Murray H. Woods. A new physical mechanism for soft errors in dynamic memories. In *16th International Reliability Physics Symposium*, pages 33–40, 1978. doi: 10.1109/IRPS.1978.362815.

[21] M. Nicolaidis. *Soft Errors In Modern Electronic Systems*. 01 2011. ISBN 978-1-4419-6992-7. doi: 10.1007/978-1-4419-6993-4.

[22] J. Maiz and N. Seifert. Introduction to the special issue on soft errors and data integrity in terrestrial computer systems. *IEEE Transactions on Device and Materials Reliability*, 5(3):303–304, 2005. doi: 10.1109/TDMR.2005.858326.

[23] Tech Powerup. Nvidia geforce gtx titan. `https://www.techpowerup.com/gpu-specs/geforce-gtx-titan.c1996`, 2013. (Accessed on 30/04/2022).

[24] Tech Powerup. Nvidia geforce rtx 3080. `https://www.techpowerup.com/gpu-specs/geforce-rtx-3080.c3621`, 2021. (Accessed on 30/04/2022).

[25] Edward Wyrwas, Kenneth A LaBel, Michael Campola, and Martha O'Bryan. Guidance on standardizing gpu test approaches. In *2018 IEEE Radiation Effects Data Workshop (REDW)*, pages 1–4, 2018. doi: 10.1109/NSREC.2018.8584282.

[26] E. J. Wyrwas. Proton Testing of nVidia GTX 1050 GPU, May 2017.

[27] NVIDIA. Cuda-gdb. `https://docs.nvidia.com/cuda/cuda-gdb/index.html`, 2019. (Accessed on 20/04/2022).

[28] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. Gpuqin: A methodology 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.

[29] NVIDIA. Llfi-gpu. `https://github.com/DependableSystemsLab/LLFI-GPU`, 2019. (Accessed on 20/04/2022).

[30] Daniel Oliveira, Vinicius Frattin, Philippe Navaux, Israel Koren, and Paolo Rech. Carol-fi: An efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators. In *Proceedings of the Computing Frontiers Conference*, CF'17, page 295–298, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344876. doi: 10.1145/3075564.3075598. URL `https://doi.org/10.1145/3075564.3075598`.

[31] Zitao Chen, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications. *arXiv:2004.01743 [cs, stat]*, April 2020.

[32] Dependable Systems Lab. Tensorfi design principles. `https://github.com/DependableSystemsLab/TensorFI/wiki/Design-Principles`, 2019. (Accessed on 30/04/2022).

[33] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W. Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258, 2017. doi: 10.1109/ISPASS.2017.7975296.

[34] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. Flexible software profiling of gpu architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 185–197, 2015. doi: 10.1145/2749469.2750375.

[35] Timothy Tsai, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, and Stephen 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, 2021. doi: 10.1109/DSN48987.2021.00041.

[36] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. Nvbit: A dynamic binary instrumentation framework for nvidia gpus. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 372–383, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358307. URL `https://doi.org/10.1145/3352460.3358307`.

[37] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. *BinFI*: An efficient fault injector for safety-critical machine learning systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pages 1–23, New York, NY, USA, November 2019. Association for Computing Machinery. ISBN 978-1-4503-6229-0. doi: 10.1145/3295500.3356177.

[38] Abdul Rehman Anwer, Guanpeng Li, Karthik Pattabiraman, Michael Sullivan, Timothy Tsai, and Siva Kumar Sastry Hari. Gpu-trident: Efficient modeling of er-

ror propagation in gpu programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

[39] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. Modeling soft-error propagation in programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 27–38, 2018. doi: 10.1109/DSN.2018.00016.

[40] Fernando F. dos Santos, Josie E. Rodriguez Condia, Luigi Carro, Matteo Sonza Reorda, and Paolo Rech. Revealing gpus vulnerabilities by combining register-transfer and software-level fault injection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 292–304, 2021. doi: 10.1109/DSN48987.2021.00042.

[41] Josie E. Rodriguez Condia, Boyang Du, Matteo Sonza Reorda, and Luca Sterpone. FlexGripPlus: An improved GPGPU model to support reliability analysis. *Microelectronics Reliability*, 109:113660, June 2020. ISSN 0026-2714. doi: 10.1016/j.microrel.2020.113660.

[42] B. Du, Josie E. Rodriguez Condia, M. Sonza Reorda, and L. Sterpone. On the evaluation of seu effects in gpgpus. In *2019 IEEE Latin American Test Symposium (LATS)*, pages 1–6, 2019. doi: 10.1109/LATW.2019.8704643.

[43] Fernando dos Santos, Luigi Carro, and P. Rech. Kernel and layer vulnerability factor to evaluate object detection reliability in gpus. *IET Computers & Digital Techniques*, 13, 05 2019. doi: 10.1049/iet-cdt.2018.5026.

[44] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450351140. doi: 10.1145/3126908.3126964. URL `https://doi.org/10.1145/3126908.3126964`.

[45] Ali Jahanshahi. Tinycnn: A tiny modular CNN accelerator for embedded FPGA. *CoRR*, abs/1911.06777, 2019. URL `http://arxiv.org/abs/1911.06777`.

[46] Fernando Fernandes dos Santos, Pedro Foletto Pimenta, Caio Lunardi, Lucas Draghetti, Luigi Carro, David Kaeli, and Paolo Rech. Analyzing and increasing

the reliability of convolutional neural networks on gpus. *IEEE Transactions on Reliability*, 68(2):663–677, 2019. doi: 10.1109/TR.2018.2878387.

[47] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W. Fletcher, Sarita V. Adve, Charbel Sakr, Naresh Shanbhag, Pavlo Molchanov, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. Optimizing selective protection for cnn resilience. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pages 127–138, 2021. doi: 10.1109/ISSRE52982.2021.00025.

[48] Fatemeh Ayatolahi, Behrooz Sangchoolie, Roger Johansson, and Johan Karlsson. A study of the impact of single bit-flip and double bit-flip errors on program execution. In Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaâniche, editors, *Computer Safety, Reliability, and Security*, pages 265–276, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40793-2.

[49] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, mar 1991. ISSN 0360-0300. doi: 10.1145/103162.103163. URL `https://doi.org/10.1145/103162.103163`.

[50] NVIDIA. Nvidia cudnn documentation. `https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html`, 2022. (Accessed on 01/03/2022).

[51] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[52] DARPA-IPTO. Autonomous off-road vehicle control using end-to-end learning. `http://net-scale.com/doc/net-scale-dave-report.pdf`, 2004. (Accessed on 10/03/2022).

[53] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to End Learning for Self-Driving Cars, April 2016.

[54] Sully Chen. Driving dataset. `https://github.com/SullyChen/driving-datasets`, 2021. (Accessed on 10/03/2022).

[55] Eder Santana and George Hotz. Learning a Driving Simulator. *arXiv:1608.01230 [cs, stat]*, August 2016.

[56] Ekin Dogus Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation policies from data. 2019. URL `https://arxiv.org/pdf/1805.09501.pdf`.

[57] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. *arXiv:1811.06965 [cs]*, July 2019.

[58] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition, April 2015.

[59] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, (0):–, 2012. ISSN 0893-6080. doi: 10.1016/j.neunet.2012.02.016. URL `http://www.sciencedirect.com/science/article/pii/S0893608012000457`.

[60] Fernando Fernandes dos Santos, Luigi Carro, and Paolo Rech. Kernel and layer vulnerability factor to evaluate object detection reliability in gpus. *IET Computers & Digital Techniques*, 13(3):178–186, 2019. doi: https://doi.org/10.1049/iet-cdt.2018.5026. URL `https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-cdt.2018.5026`.

List of Figures

1.1 Correct labelling . . . 2
1.2 Wrong placement . . . 2
1.3 Wrong labelling . . . 2

2.1 CNN example . . . 8
2.2 Representation of a tensor storing a RGB image . . . 9
2.3 ReLU . . . 12
2.4 Leaky ReLU . . . 12
2.5 PReLU . . . 12
2.6 ELU . . . 13
2.7 Sigmoid . . . 14
2.8 SM architecture . . . 18
2.9 Thread Memory . . . 19
2.10 SASSI instrumentation flow . . . 27
2.11 CUDA Software Stack . . . 28
2.12 NVBit tool compilation and usage . . . 29

3.1 Framework Architecture - Initiation Phase . . . 38
3.2 Framework Architecture - Usage . . . 38
3.3 Offsets vector . . . 45
3.4 Strides vector . . . 45
3.5 Intersection of architectural faults and error models . . . 46
3.6 Add . . . 48
3.7 Exp . . . 48
3.8 Convolution 1 . . . 49
3.9 BatchNormalization . . . 49
3.10 Add domain values . . . 52
3.11 Exp domain values . . . 52
3.12 BatchNorm domain values . . . 53
3.13 Convolution 1 domain values . . . 53

3.14 Error Simulator . . . 55
3.15 Framework Architecture Technologies - Initiation Phase . . . 56
3.16 Framework Architecture Technologies - Usage . . . 56

4.1 PilotNet Model . . . 63
4.2 Street on a sunny day . . . 63
4.3 Strong lens flare . . . 63
4.4 Street with shadow and other vehicles . . . 64
4.5 Private parking exit with strong light contrast . . . 64
4.6 First add operator in the net . . . 65
4.7 Last add operator in the net . . . 65
4.8 Mul operator . . . 66
4.9 Last convolutional operator in the net . . . 66
4.10 Values distribution of Add and Mul error models . . . 68
4.11 Comma's AI Steering Wheel Model . . . 68
4.12 Last BiasAdd of the model . . . 69
4.13 Third convolutional layer . . . 70
4.14 Penultimate BiasAdd of the model . . . 70
4.15 BiasAdd operator of third convolutional layer . . . 71
4.16 Cifar10 Recognition Model Scheme . . . 72
4.17 CIFAR10 Samples . . . 73
4.18 Cifar10 Model comprehensive misclassifications . . . 74
4.19 Conv1-BiasAdd misclassifications . . . 75
4.20 Conv1-Convolution misclassifications . . . 75
4.21 Conv1-BiasAdd Top2 distribution . . . 76
4.22 Conv1-Convolution Top2 distribution . . . 76
4.23 BatchNorm3 Top2 distribution . . . 78
4.24 Dense2 Top2 distribution . . . 78
4.25 VGG11 Model Scheme . . . 79
4.26 GTSRB Samples . . . 80
4.27 Mul Top2 distribution . . . 82
4.28 Add Top2 distribution . . . 82
4.29 Mul misclassifications . . . 83
4.30 Add misclassifications . . . 83

5.1 PilotNet . . . 90
5.2 Comma's AI . . . 91
5.3 CIFAR-10 model . . . 92

5.4 VGG-11 model 93

List of Tables

2.1 Different ReLU implementations 13
2.2 NVBitFI injection's possible errors 30

3.1 Set of operators selected for the error modelling 43

4.1 Difference classification for steering angle models 64
4.2 Execution times comparison 84

5.1 Cifar-10 layers' execution time 93

List of Symbols

Variable	**Description**
s_{err_i}	Operator's susceptibility
R_i	Reliability against operator i
T_i	Timing weight of operator i
R_{CNN}	CNN Robustness
PD_i	Performance degradation introduced by layer i
R_{SH_CNN}	Hardened CNN Robustness