POLITECNICO DI MILANO

School of Industrial and Information Engineering Department of Electronics, Information and Bioengineering Master of Science Degree in Computer Science and Engineering

Effective and Inexpensive Fault Detection in VGG-16 Inference

Advisor: Prof. Giacomo Boracchi
Co-Advisor: Dott. Diego Stucchi

Master Thesis by:
Gabriele Gavarini, 912948

A.Y. 2020-2021

Ai miei genitori e a Margherita, per il supporto durante gli anni.

Abstract

An increasing number of applications take advantage of Convolutional Neural Networks due to their effectiveness in image vision tasks. However, these models can be affected by critical faults that change their prediction. Previous works have studied what happens when critical faults impact a LeNet5 network. In particular, they have proposed an architecture for a fault detector composed of two networks, where the auxiliary model acts as a controller for the main network. This Master Thesis aims to expand the same analysis to a larger network such as VGG-16. Moreover, we will propose an effective and inexpensive fault detector that analyzes the vector score of VGG-16 to detect critical faults, without the need for an auxiliary network.

Ringraziamenti

In primo luogo, voglio ringraziare la mia famiglia, i miei amici e Margherita, senza i quali non sarei mai riuscito ad arrivare dove sono ora. Ringrazio anche Giacomo e Diego per il loro costante supporto e per le possibilità che mi hanno offerto. Allo stesso modo ringrazio anche Ernesto e Annachiara, senza i quali non sarei mai riuscito a completare questo lavoro. Un ringraziamento finale va anche ad Andrea, compagno di avventura in questa mia ultima tappa universitaria.

Contents

1 Introduction **1**

2 Theoretical Background **5**
2.1 Convolutional Neural Networks 5
2.1.1 Neural Network and Deep Neural Network 5
2.1.2 Training a Neural Network 7
2.1.3 Training with batches 9
2.1.4 Convolutional Neural Network 9
2.1.5 VGG 12
2.1.6 MobileNet 14
2.2 Knowledge Distillation 15
2.2.1 K-D loss 16
2.2.2 Training with Knowledge Distillation 16
2.3 Faults in Convolutional Neural Networks 17
2.3.1 Classification of Faults 17
2.3.2 Critical Data Corruptions 18
2.3.3 Fault Tolerance 18
2.3.4 Faults in Neural Network Accelerators 19
2.3.5 Impact of faults on weights 19
2.4 Fault Injections 20
2.4.1 Hardware Fault Injections 20
2.4.2 Software Fault Injections 21

3 Problem Formulation **23**

4 Proposed Solutions **25**
4.1 Auxiliary-Based Fault Detectors 29
4.1.1 Auxiliary architecture: VGG-Like 31
4.1.2 Auxiliary architecture: MobileNet 34

4.2 Score-Based Fault Detectors 34
4.3 Proposed Solutions Recap 37

5 Experimental Setup **39**
5.1 Selection of a Fault Injector 39
5.2 Limitations of the Fault Injector 40
5.3 Extension of the Fault Injector 41
5.3.1 N2D2 42
5.3.2 Python / PyTorch 42
5.3.3 Fault Injector 43
5.4 Running a fault injection campaign 44
5.5 Metrics Computation 45
5.5.1 Fault Impact 45
5.6 Auxiliary Networks 45
5.6.1 Dataset Split 45
5.6.2 Training the Auxiliary Model 46
5.6.3 Compute model prediction 47
5.7 Fault Detector 47
5.7.1 Threshold computation 47
5.7.2 Fault Detector Evaluation 48

6 Results **51**
6.1 Fault Injections Result 51
6.2 Auxiliary Models Training 53
6.3 Auxiliary-Based Fault Detector Results 56
6.3.1 Norm of the Difference 56
6.3.2 K-D Loss 59
6.4 Score-Based Fault Detectors Results 61
6.5 Final Considerations 63

7 Conclusion and Future Works **65**
7.1 Contributions 65
7.2 Future Works 66

Chapter 1

Introduction

An increasing number of applications exploit Deep Neural Networks (DNN) due to their effectiveness in task such as Image Classification [1] [2], Image Segmentation [3] [4], or Speech Detection [5]. Convolutional Neural Networks (CNN), a specific kind of DNNs, are among the leading technologies in computer vision. While very effective, CNNs are composed of millions of parameters and, as such, are computationally expensive. For this reason, researchers have focused on improving the effectiveness of Convolutional Neural Networks. On the one hand, networks have become smaller and more accurate, while on the other, new hardware solutions have been developed to speed up their computation. Hardware accelerators - such as GPUs, ASICs, or FPGAs - improve the performances of Convolutional Neural Networks by taking advantage of their natural predisposition to parallelization. In a CNN, a convolutional layer applies the same transformation - with the same parameters - to all its inputs. Consequently, a hardware accelerator that paralyzes these operations greatly decreases the time needed to compute a network prediction [6].

Unfortunately, accelerators are vulnerable to different sources of faults, such as radiation-induced errors, aging, or temperature [7]. Moreover, due to the high level of parallelization involved, they are prone to propagate these errors to multiple output elements: accelerators are more susceptible to malfunctions than regular hardware architectures. The lower reliability of this components is of particular concern, as highlighted in [8] and [9], in safety-critical systems, like in the automotive sector. When a fault changes the prediction of a network we talk about critical faults: this type of faults are dangerous as they are not easy to identify [6].

As a result, more researchers have begun to focus on the reliability of

convolutional neural networks [6] [10]. We can distinguish between a passive and an active approach to this problem. A system with *active fault tolerance* is designed to deal with faults without having a component dedicated to this task. Consequently, reliability is achieved through some type of redundancies [11]. An example of this is the Algorithm-Based Fault Tolerance, where a checksum is added to every operation. This process adds significant overheads, increasing the computational cost of the network. Conversely, a system with *passive fault tolerance* takes advantage of a specifically designed fault detector to identify the faults. Additionally, it employees a supervisor element to deal with the faults that have been identified [6] [10]. Recognizing whether the output of a network is the result of a hardware malfunction is not an easy task. Ideally, the best fault detector would be a replica of the network. By comparing the network prediction with the replica prediction, it is easy to identify a misclassification caused by a critical fault. However, this is prohibitively expensive to implement in practice. While some works treat the fault detector as a black-box [12] [13], others presented an architecture based on a smaller network replica [14]. In this latter case, the fault detector compares the network output with the output of its smaller version - the auxiliary network - to detect the presence of a fault. As the auxiliary is not a complete replica of the network, the fault detector can not simply compare the output of the two. Being the auxiliary network an approximate model, the prediction of the two networks may be different even without a critical fault. Consequently, the fault detector looks at the norm of the difference between the vector scores of the two models.

In our Master Thesis, we aim to expand the work presented in [14], as we have collaborated closely with its author. Since results were shown only for LeNet5, a small convolutional network, our work aims to study the performance of the replica-based fault detector on VGG-16 [1], a state-of-the-art Convolutional Neural Network composed of more than 100 million parameters. The first thing we are going to introduce is the implementation of an Auxiliary-Based Fault Detector for VGG-16. In particular, we will present different alternatives for the auxiliary network and compare their performances. The design of these architectures is not trivial: it requires a careful balance between model accuracy and computational cost. On the one hand, the auxiliary model needs to be small enough not to impact negatively VGG-16 performances. On the other hand, since this additional component acts as a controller for the main network, their vector scores need to be as close as possible. For this reason, we have developed four different architectures for the auxiliary network: two models have the same structure as VGG-16 with fewer layers, the other two are MobileNets [2]. We propose

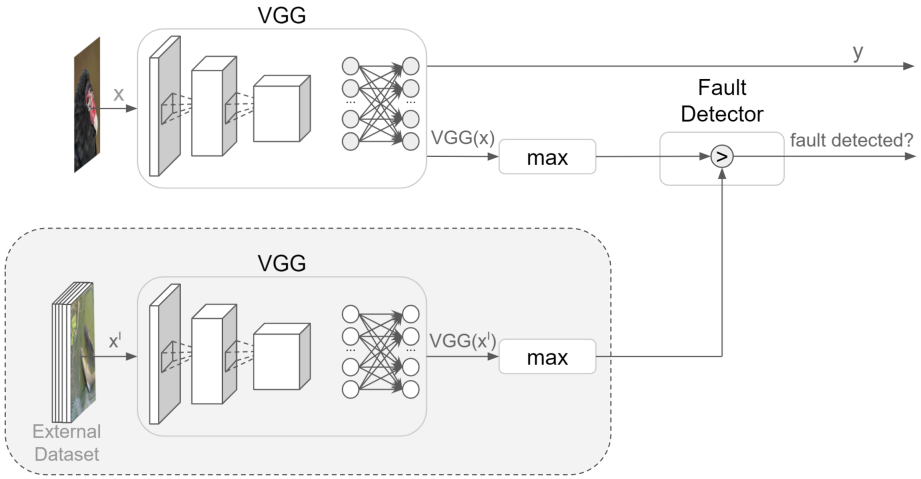

Figure 1.1: The architecture of the proposed fault detector. The Fault Detector compares the maximum of the vector score $VGG(X)$ with the maximum value ever reached by a component of VGG-16 vector score when predicting the images of the External Dataset. The computation of the maximum value needs to be done only once before using the Fault Detector.

these architectures as they reduce the number of parameters and operations of VGG-16. These models have been trained using knowledge distillation [15] to have an auxiliary network with a vector score as close as possible to one of the main network.

Additionally, we are going to introduce a more efficient solution, which we call *Score-Based Fault Detector*, as it looks only at the vector score of VGG-16 to detect a fault. The new fault detector we propose doesn't require a replica of the network. It works by comparing the maximum element of this vector score with the maximum value ever reached by the vector score when VGG-16 was used to predict images from an external dataset, as shown in *Figure 1.1*. When a fault impacts the network, we expect it to modify its vector score. Moreover, a critical fault that changes the prediction, changes the order of the element of the vector score. Consequently, we can expect that the component of maximum value in the vector score is a good element to inspect when we want to detect a fault. Our idea is to record, before performing any prediction, the maximum value ever reached by a component of the vector score when VGG-16 is evaluated over a fixed dataset. If for a prediction, the value of the top element of the vector score is much higher than this value, there is a high chance that it is the result of a critical fault.

Our experiments show that the fault detector we are going to propose is more effective and less expensive than the one employing a network replica. In particular, we are going to show that the *Score-based Fault Detector* has better accuracy, precision, and recall than its *Auxiliary-Based* counterpart. These measurements have been obtained for multiple faulty runs where we inject different numbers of faults in VGG-16.

The thesis is structured as follows:

- in Chapter 2 we introduce background concepts necessary to understand our work. We will introduce CNNs and VGG-16, other than talk about why faults occurs and what is their impact;
- in Chapter 3 we formally introduce our problem;
- in Chapter 4 we present 2 Auxiliary-Based Fault Detectors that require a smaller replica of the network, and 2 Score-Based Fault Detectors that work without the need of an additional model;
- in Chapter 5 we introduce our experimental setup, with a focus on the implementation of a fault injector, a tool used to simulate the presence of faults in a neural network;
- in Chapter 6 we present the results of our experiments, comparing the different models and showing why the one based on the max is the best fault detector;
- in Chapter 7 we summarize our work, detailing possible directions for future studies.

Chapter 2

Theoretical Background

In this chapter we are going to introduce some key concepts that are useful to understand the content of the thesis. Particularly, we will first introduce neural networks and *convolutional neural networks*. After that, we are going to talk about the *VGG-16*, which is the primary work of this thesis. We will then discuss how faults may happen in neural network *accelerators* and their impact. Finally, we will briefly talk about *fault injection*: techniques used to simulate this kind of malfunctions.

2.1 Convolutional Neural Networks

In this section we are going to briefly introduce neural networks. After a general introduction to the topic, we will talk about *Convolutional Neural Networks* and how they work. After that, we will focus on one of the most important architecture, VGG-16, that, despite being outperformed by more recent networks, is still an excellent model for image classification.

2.1.1 Neural Network and Deep Neural Network

A neural network is a non-linear model that takes that can be used for *regression* and for *classification*. In a classification task, we want to decide to which of the available class assign a given input. Let Δ be the set of the possible classes and $t \in \Delta$ the *target* class for input $x \in \mathbb{R}^N$. Our objective is to find a function $\bar{Y} : \mathbb{R}^N \to \Delta$ such that:

$$\bar{Y}(x) = t \tag{2.1}$$

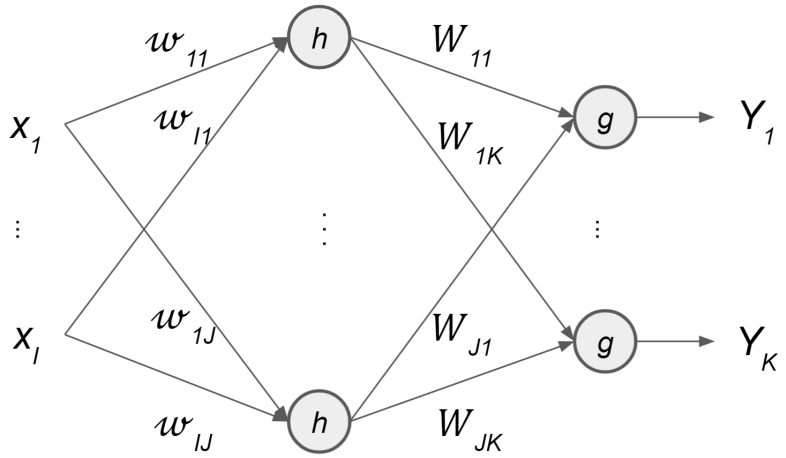

Figure 2.1: Architecture of a feed forward neural network composed of a single hidden layer.

As shown in *Figure 2.1*, a neural network is a graph where every node is called a *neuron*. If the graph is *directed and acyclic*, then the architecture it is called a *feed-forward neural network*. We can group neurons in *layers*: an *input layer* containing the input neurons, one or more intermediate *hidden layers*, and an *output layer* containing all the output nodes. One of the most important results is the *Universal Approximation Theorem* [16], which states that a feed-forward neural network composed of a single hidden layer is capable of approximating any continuous function $f : \mathbb{R}^n \to \mathbb{R}^m$.

As we have seen before, a neuron is the smallest component of a neural network. To better understand how they work, let's analyze in detail what a neuron does. A neuron (*Figure 2.2*) computes a *linear combination* of its input by applying a non linear transformation h, called the *activation function*. Moreover, not all the inputs have the same importance, and this is represented by weighting each input with a *weight* w_i. Additionally, every neuron has a dummy input whose value is fixed to one, and its associated weight is called the *bias*. In literature, it is often referred to as w_0. Therefore, the output of a neuron with inputs $x =< x_1.., x_I >$ is defined as:

$$y(x) = h(\sum_{i=1}^{I} w_i x_i + w_0) \tag{2.2}$$

We can extend this formulation to the whole architecture. Let's focus

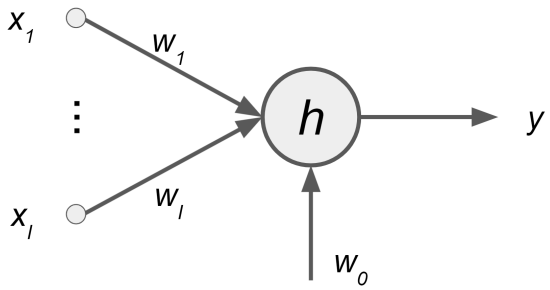

Figure 2.2: Anatomy of a neuron

on a feed-forward neural network composed of a single hidden layer, like the one shown in *Figure 2.1*. The J neurons intermediate layer applies a non-linear transformation h with weights $w =< w_1...w_I >$, while the neurons of the output layers apply a non-linear transformation g with weights $W =< W_1...W_J >$. Let θ be the set of all the weights of the network. Given an input whose components are $X =< X_1...X_I >$, the k-th component of the *vector score* $Y_k(x, \theta)$ is:

$$Y_k(x, \theta) = g(\sum_{j=1}^{J} W_j h(\sum_{i=1}^{I} w_i x_i + w_0) + W_0) \tag{2.3}$$

From this vector we can retrieve the prediction of the network $\bar{Y}$ as the component of maximum value, that is:

$$\bar{Y}(x, \theta) = \arg\max_k [Y(x, \theta)]_k \tag{2.4}$$

2.1.2 Training a Neural Network

Training a model means find the optimal values of the weights in order to minimize some kind of pre-defined *loss function*. Typically, the most used loss function for multi-class classification is the *categorical cross-entropy* $E(x|t, \theta)$ and for a set of weights θ is defined as:

$$E(x|t, \theta) = \sum_{k=1}^{K} t \ln Y_k(\theta, x) \tag{2.5}$$

where (x, t) is a pair input, target. The goal is to find a set of weights θ that minimizes the loss function for all the pairs (x, t) of our training dataset.

Notice that a neural network is *non-linear model* and the loss can be a *non-convex* function: finding its minimum is not trivial.

The method used to find the right set of weights is called *backpropagation*. The weights of the network are updated using *stochastic gradient descent*: fore every wight of the network $w \in \theta$, we subtract from an value, the gradient of the loss function multiplied by a parameter α called the *learning rate*:

$$w^{k+1} = w^k - \alpha \frac{\partial E}{\partial w}\bigg|_{w=w^k} \tag{2.6}$$

where w^k is the value of the weight for the current iteration while w^{k+1} is the value of the weight for the next iteration. The computation of the loss -and generally of the output- is often referred to as the *forward pass*. We call a pass over the whole dataset an *epoch*.

The training of the network stops when we have found a satisfactory approximation. Notice that finding the set of weights for which the loss is close to zero is rarely the right approach: by doing so, we risk to *overfit* the data to the training set. More precisely, overfitting means that the model is very good at approximating the function for inputs it has been trained on, but does not perform as well with inputs it has never seen before. In this case, we say that the network is *biased* toward the training set.

To evaluate the actual performance of the model we need an external dataset, composed of images that are not given as input to the network during training. This dataset, called the *validation dataset*, is what is used to estimate the loss function with no bias. Since reserving a portion of the dataset can hurt the training process, we want to minimize the amount of data assigned to the validation dataset.

A solution to this problem can be found in *cross-validation*. The idea is to split the dataset in multiple *folds*, and to train the model over all but one fold, using the remaining one as the validation dataset. At the end of the epoch, we switch the fold in the validation set with one of the fold in the training set. When using k folds this method is known as *k-fold cross-validation*.

While using techniques such as cross-validation, we can employ various model selection techniques. The most used ones in neural network is *early stopping*: when the loss over the validation set does not decrease for a given number of epochs - called the *patience* - we stop the training. Notice that we don't need to use the same loss for training and for evaluation. In fact, we do not need a loss at all for the evaluation phase: we can track other values such, for example, the network accuracy.

2.1.3 Training with batches

In order to compute the weights update of a neural network we need to compute the gradient of the loss function with respect to the weights of each input image multiple times. However, computing the gradient for all the inputs simultaneously is not practical, mainly due to memory limitations. Therefore, input data are split into sub-groups, called *batches*. By doing so, we are no longer computing the gradient over all the inputs, but only over a batch.

Using batches has other implications. In fact, the computation of the gradient is faster as it requires less operations. However, using small batches means that the intermediate computations are less accurate, slowing down the gradient convergence. Therefore, choosing the right batch size is crucial to speed up the training.

2.1.4 Convolutional Neural Network

Despite the ability of feed-forward neural networks of approximating every continuous function with a single hidden layer , this architecture is not always the best choice for learning a given mapping. The seminal work of Yann LeCun [17] has shown that some tasks are better suited to *Deep Neural Networks* (*DNN*); feed-forward neural networks composed of multiple hidden layers. *LeNet* [17], the network he proposed, is an example of this (*Figure 2.3*).

Deep Neural Networks are an excellent tool for *image recognition*, where we want to assign a label that describes the input image. This task is typically achieved by a specific kind of DNN, called *Convolutional Neural Network*. This models is mainly composed by two kind of layers: *convolutional layers* and *fully-connected layer*. The latter are the same we have

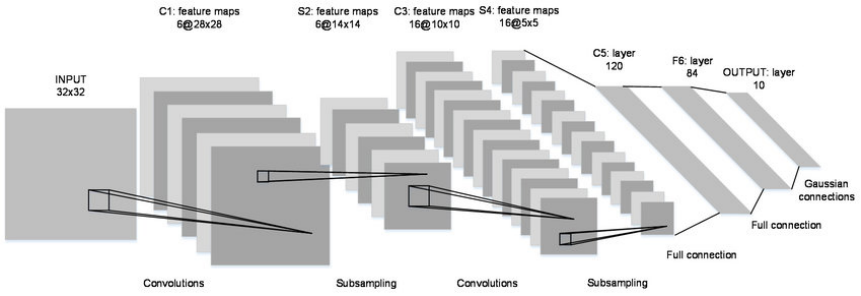

Figure 2.3: LeNet, image from [17]

being discussing until now and, as the name suggest, are composed by neurons that have a connection with all the neurons of the previous layer and all of the next layer. To better understand how convolutional layers work, let's first introduce the *convolution* operation.

Convolution is a filtering operation that consists in applying to an image a *filter* -or *kernel*- of size N applied to the pixel in position (r, c) of the image. Conceptually, by applying the convolution operation we want to multiply the value of the pixel of a neighborhood of the target position with the value of the filter.

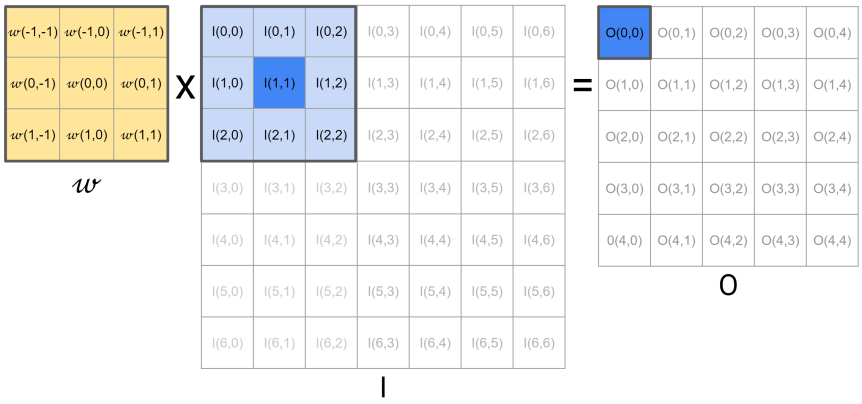

Figure 2.4: A 3×3 *convolutional* filter w applied to a 7×7 image I, resulting in an 5×5 output image O.

More specifically, let's consider an input image I of width W and height H, composed by only one channel (i.e. it has depth of 1). This can be represented as a $1 \times W \times H$ tensor. We apply to it a filter w of size N, represented by a $1 \times N \times N$ tensor. The output of the convolution O applied to position (r, c) is a *linear combination* of the image and the filter:

$$O(r,c) = \sum_{x,y=-\lfloor \frac{N}{2} \rfloor}^{\lfloor \frac{N}{2} \rfloor} w(x,y) I(r+x, c+y) \tag{2.7}$$

By applying the same kernel to every pixel of the input image, we obtain a filtered image. We can see the application of a filtering operation in *Figure 2.4*. Since the convolution is a linear combination of its inputs, we can easily implements a neuron that works as a convolutional filter. A network in which some of its neuron act as a convolutional filter is called *convolutional neural network*. We can split the components of these models into 4 groups:

convolutional layers, *activation functions*, *pooling layers*, and *fully connected layers*.

In a *convolution layer* the output is a linear combination of all the values in a region of the input. As previously discussed, a filter is applied through the whole spatial extent of the input. We can choose to apply a filter only to certain pixels: we call *stride* the distance between two application of the convolution operation. It is possible to apply multiple filters: different filters yield multiple output channels. While each filter must have the same depth as the input image, the depth of the output is dictated by the number of filters applied. Therefore, we say that a convolutional layer has a *large depth extent* because it covers the whole volume fed as input. On the other hand, they have a *small spatial extent*.

An *activation layer* is used to add non-linearity into the network. Among other things, the non-linearity of the output function is what allows neural networks to work as a universal function approximator. Typically, the activation function of the last layer of a convolutional neural network is a *softmax* function that transforms a vector - whose component are called the *logits* - to a probability distribution. Given a vector $z \in \mathbb{R}^J$, we define a softmax function $s : \mathbb{R}^J \to \mathbb{R}^J$ as:

$$s_i(z) = \frac{e^{\frac{z_i(x)}{\tau}}}{\sum_j e^{\frac{z_j(x)}{\tau}}} \tag{2.8}$$

where τ is the *temperature*, a parameter used to decide how sharp the probability distribution should be.

While convolutional layers greatly increased the depth of the volume and slightly decreased its spatial extension, the *pooling layers* are used to reduce the spatial extension. The pooling layer operates independently on every depth slice of the input and resizes it spatially, often using the max operation: this class of layers is called *maxpooling*.

Another example of a pooling layer is the *Global Average Pooling* (*GAP*) *layer*, first proposed in [18]. In a classification task where there are N output classes, a GAP layer transforms a $W \times H \times N$ tensor in a vector of N components, as shown in *Figure 2.5*. The purpose of this layer is to avoid the presence of a fully connected portion of the network. Furthermore, it transforms the output of a convolutional layer into something that can be fed directly to a softmax layer. By using this layer, we increase the efficiency of the network. This layer applies an *average* to each of the channels of the input tensor. This means that the element in position i of the output tensor is the average of all the elements of channel i in the input tensor.

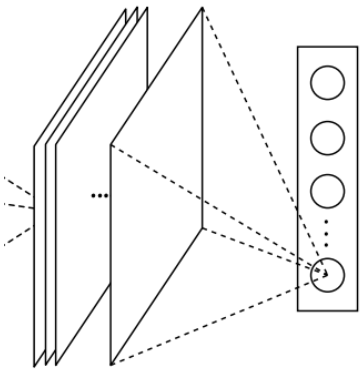

Figure 2.5: A GAP layer, image from [18]

The main idea behind convolutional neural networks is that a convolutional layer will act as a *feature extractor*: the filters are used to learn -and later find- certain regions of pixel in the input image that are associated with a feature. Multiple features describe the class to which the image belongs to. The concept of feature can be better understood by introducing the *receptive field*. In a regular fully connected layer, the value of every single pixel in the output image is dependent on the whole input image. However, due to the properties discussed before, in a convolutional neural network, a pixel in the output image depends only on a small region of the input. This region is called the receptive field of that output. As we traverse the depth of a convolutional neural network, the size of the receptive field tend to increase. This is due to maxpooling and convolution operations, as well as having a stride factor greater than 1.

2.1.5 VGG

The network introduced by LeCun in 1998, despite working well with the small MNIST dataset it was conceived for, was far too simple to deal with bigger images. Starting in 2010, a competition called *the ImageNet Large Scale Visual Recognition Challenge* (*ILSVRC*) was launched to encourage the research of deep neural networks capable of dealing with bigger input images. The organizer of this competition provided a dataset, often called the *Imagenet Dataset*. To this day, this dataset is one of the most used, mainly due to its size: it contains more than 14 million labeled images split

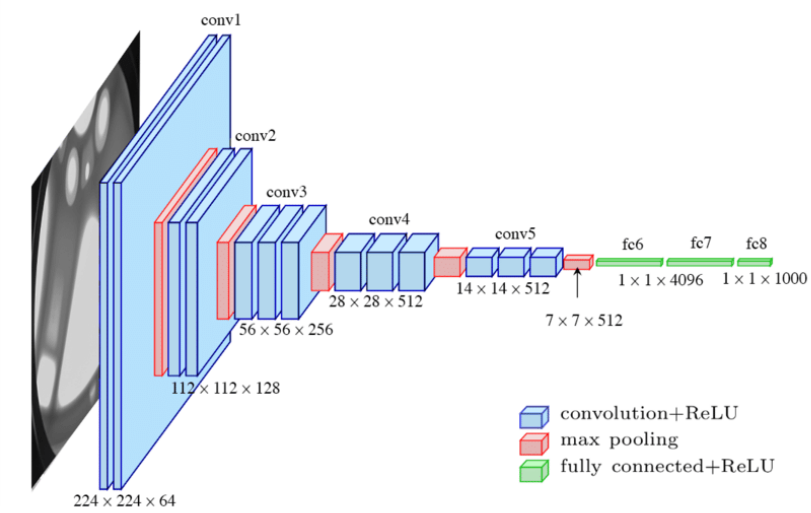

Figure 2.6: VGG-16, image from [1]

in 1000 non-overlapping classes.

One of the most effective network to compete in the ILSVRC is *VGG* (Visual Geometry Group), developed in 2015 [1]. This network was able to achieve impressive results when it was first presented, reaching a 23.7% top-1 validation error and a 6.8% top-5 validation error. In *Figure 2.6*, we can see a scheme for the network: it is composed by multiple convolutional, maxpooling and fully connected layers. At first, we can split the network in two segments, a fully connected part and a convolutional one. Furthermore, we can group layers of the convolutional part into different sections, each formed by a certain number of convolutional layers and by a maxpooling layer.

Various topologies of VGG were presented, each characterized by a certain number of layers. The one shown in *Figure 2.6* is called VGG-16 as it contains 16 different layers. *VGG-16* takes as input a $224 \times 224 \times 3$ image and output a 1000-sized vector. The predicted class is the one associated with the component of highest value. The activation function of the last layer is a *softmax* rather than a relu. This is crucial as, in our thesis, we are going to analyze the output of the network before the application of the softmax activation function.

Since Imagenet is composed of images of varying size, to input them to VGG we must re-scale and re-size them. First, images are re-scaled to a

fixed size of 256×256, then they are *center-cropped* to the actual input size of 224×224. An additional pre-processing step is to *mean-center* the data: we subtract the mean RGB value -computed on the training set- to each pixel of the input image.

2.1.6 MobileNet

Another network that was developed for the ILSVRC competition is MobileNet [2]. This model is an efficient convolutional neural network designed for mobile or embedded applications. This efficiency is achieved trough *depth-wise separable convolution* and by defining an hyper-parameter α called the *width multiplier*.

In a standard convolutional layer, we apply a $N \times N \times C_I \times C_O$ filter to the image, where N is the dimension of the kernel, C_I is the number of input channels and C_O is the number of output channels. By applying this filter, we transform an input tensor of size $W_I \times H_I \times C_I$ to an output tensor of size $W_O \times H_O \times C_O$. The cost of this operation is:

$$N \times N \times C_I \times C_O \times H_O \times W_O \tag{2.9}$$

In a depth-wise separable convolution layer, we split this operation in two parts, as shown in *Figure 2.7*. First, we apply a *depth-wise convolutional filter* of size $N \times N \times C_I$. By applying this filter we are not modifying the depth of the output tensor. In standard convolution we apply C_O filters,

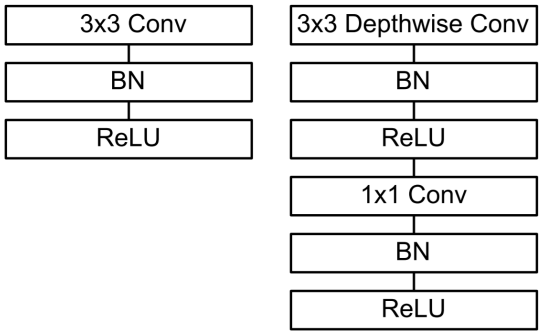

Figure 2.7: On the left we can see a traditional convolutional layer with a batch-normalization layer and a relu activation function. On the right, we can see the *depth-wise convolutional layer* and the *point-wise convolutional layer*m each followed by a batch-normalization and a relu. Image from [2]

each of depth C_I, to our image. In this case, instead, we are only applying a single filter of depth C_I. The dimension of the output of this layer is $W_O \times H_O \times C_I$. In this case, the number of operations is:

$$N \times N \times C_I \times H_O \times W_O \tag{2.10}$$

To obtain an output of the same size of the standard convolution, we need to apply an additional operations, called the *point-wise convolution*, where we apply C_O 1×1 filters to the input tensor, therefore the cost is:

$$C_I \times C_O \times H_O \times W_O \tag{2.11}$$

By combining (2.10) and (2.11) we obtain the total cost of the *depth-wise separable convolution*:

$$N \times N \times C_I \times H_O \times W_O + C_I \times C_O \times H_O \times W_O \tag{2.12}$$

By dividing (2.9) by (2.12) we obtain the reduction in number of operation. This value is:

$$\frac{1}{C_O} + \frac{1}{H_O \times W_O} \tag{2.13}$$

Furthermore, the *width multiplier* hyper-parameter $\alpha \in (0, 1]$ can be used to further decrease the number of operations. For a given layer and a given α, the number of input channels C_I becomes αC_I and the number of output channels C_O becomes αC_O. Using this parameter, the computational cost of the *depth-wise separable convolution* becomes:

$$N \times N \times \alpha C_I \times H_O \times W_O + \alpha C_I \times \alpha C_O \times H_O \times W_O \tag{2.14}$$

Choosing an $\alpha < 1$ decreases the accuracy of the network. A MobileNet with $\alpha = 1$ reaches a validation accuracy over the Imagenet Dataset of 70.6%.

2.2 Knowledge Distillation

Knowledge distillation is a technique that consist in training a neural network, called the *student*, to closely follow another trained model: the *teacher* [15]. That is, during training, we want the student model to minimize a custom loss - often referred as the *k-d loss* - that is computed comparing the prediction of the student with the prediction of the teacher. Knowledge distillation is often employed to perform *model compression*: this technique allows us to create a smaller version of a network without losing its accuracy.

2.2.1 K-D loss

Given an input x with a target label t, a multi-class student model S and a multi-class teacher model T, we define the *k-d loss* $E_{KD}(x|t)$ as the *cross-entropy* loss between the output of the two models:

$$E_{KD}(x|t) = \sum_i S_i(x|t) \log T_i(x|t) \tag{2.15}$$

In most neural networks, the output layer is typically a *softmax* layer, converting *logits* to pseudo-probabilities (2.8). The *temperature* τ is very important in knowledge distillation, since can to manipulate how the logits are converted into probability distribution. High values of τ smooth the distribution of $s(z)$ over all its components, while values closer to 1 return sharper distributions.

In knowledge distillation, we use a high value of the softmax temperature during training, both for the student and the teacher model. We do this for two reasons. First, this allows us to reduce the information loss resulting from the application of the softmax. Second, the gradient of the loss function has a smaller variance, allowing for larger values of the learning rate [15]. During testing, the temperature is reverted back to 1 for the student model. In this case we are only interested in the prediction and not in the relationship between the components of the output.

Algorithm 1 Knowledge Distillation Training

```
1:  function KnowledgeDistillationTraining(S, T, τ)
2:      Set S and T temperature to τ
3:      for every epoch do
4:          for x ∈ X do
5:              t = T(x)
6:              s = S(x)
7:              E_KD(x) = Σ_i^I s_i log t_i
8:              backpropagation(S, E_KD(x))
9:          end for
10:     end for
11: end function
```

2.2.2 Training with Knowledge Distillation

The complete algorithm for training a student model with distillation learning can be found in *Algorithm 1.* In line 2 we setup the models by setting

the temperature to the desired value. In line 3-8, for each epoch, we cycle trough all the images of the dataset. For every input, we make a prediction for the student model and for the teacher model, obtaining the probability distribution for the two networks (line 5-6). In line 7 We use these vectors to compute the k-d loss. With this value we can compute the updated value of the student network weights trough *backpropagation* (line 8).

2.3 Faults in Convolutional Neural Networks

In this section we are going to introduce the most common faults in neural networks and give a brief explanation on why convolutional neural networks are more prone to errors than other architectures. In particular, we are going to introduce the concept of *silent data corruptions* (*SDC*) and show how faults occurring in *neural network accelerators* are related to this kind of errors.

2.3.1 Classification of Faults

A definition of fault can be found in [10]: "*A fault is an anomalous physical condition in a system that gives rise to an error. An error is a manifestation of a fault in a system, the deviation from the expected output, in which the logical state of an element differs from its intended value*". In literature, faults are split in two macro-classes: *permanent* and *transient*. This subdivisions is based on the duration of the malfunction.

Permanent faults are malfunctions typically caused by a hardware failure. As the name suggests, a fault of this kind is constant in time: once it happens, the fault will always be there. Therefore, when such a defect present itself, the only way to get permanently rid of it is to substitute the failing component.

Instead, *transient fault* are the result of external disturbances and, as such, they last for a brief span of time. In fact, this kind of malfunction can be fixed at software level. They are often referred to as *soft errors*.

To model permanent faults, one of the most accepted technique found in literature is *stuck-at* values, that is: a data or control line is stuck at 1 (*stuck-at high*) or at 0 (*stuck-at low*). Transient faults, instead, can be modeled with a random *bit-flip*: a bit of a registry, or of a memory location, is flipped to the opposite of its original value.

What is particularly interesting to us, is to analyze how a permanent fault impacts a neural network. This kind of malfunction can modify the results of certain operations (by corrupting register values) or change some

element stored in memory. We are interested in studying this latter case. In particular, we are going to examine what happens when one or more faults modifies the value of a network weights during the prediction phase.

2.3.2 Critical Data Corruptions

A neural network can react to a malfunction by changing its output. We typically define a *faulty network* as a network that has been affected by a fault, while a *clean network* is a network that is not influenced by faults. If the vector score $Y^F(x) \in \mathbb{R}^J$ of a faulty network $Y^F(x)$ for a given input is different from the output of the corresponding clean network over the same input $Y^C(x) \in \mathbb{R}^J$ we say that there is a *data corruption* affecting input x. Furthermore, if this changes the prediction of the network, we talk about *critical data corruption* [6].

This kind of corruption is the most interesting to study, as it changes the prediction of the network but is not easily detectable. As such, it is often referred to as *Silent Data Corruption*. While it may change the vector score and even the prediction of the network, it doesn't affect anything else. Without knowing the prediction of the network when it is unaffected by faults, it is impossible to immediately recognize an output as critically corrupted.

2.3.3 Fault Tolerance

An increasing number of papers focus on the creation of an architecture that is capable of handling faults. The proposed solutions can be grouped into *passive* and *active* fault tolerant architectures [10].

Passive solutions employ architectures that don't explicitly detect faults. In fact, there is no additional structure designed to detect faults and no additional structure to deal with the detected malfunctions. Rather, they employ network redundancies or error correcting codes to deal with faulty connections. This kind of solution often requires more computational power, as the idea is to introduce some kind of inefficiency, either by duplicating some neurons, or by increasing the number - or the complexity - of operations.

On the other hand, *active* solutions do not change the structure of the network itself, but they introduce special *fault detection* tools and *controller* components. Notably, the most explored topic in this kind of solutions is the realization of the controller, while the detector is often considered as a black box by most authors [12][13]. However, some works explore how to implement a fault detector: for example in [19] a solution consisting in fixed

probing vectors to identify the presence of faults is proposed. The literature dealing with fault detection is limited and quite dated, as the application of this kind of techniques is not documented for complex architectures such as convolutional neural networks.

2.3.4 Faults in Neural Network Accelerators

In the last few years new hardware architecture have been developed to increase the performances of deep neural networks. These computing elements, often referred to as *DNN accelerators*, are composed by multiple simple processing units that work on parallel over a given input.

The convolutional layer takes advantage of this component the most: since we need to compute the same convolution operation - with the same parameters - for all the elements of the input tensor, we can easily parallelize this task. However, this speed increase has a drawback in terms of reliability. Cheaper computing units and memories increase the possibility of a fault, while the parallelization of the operations tend to propagate a single fault, compromising the ability of a network to deal with it [6].

2.3.5 Impact of faults on weights

As we discussed, most of the fault that affect a neural network modify the value of the weight stored in memory. Temporary faults are modeled with a bit-flip while permanent faults are modeled with a stuck-at.

Independently of the nature of the fault, when it modifies a bit of the weight it will have a different effect, depending on which data type is used and the position of the bit itself. We are going to analyze what happens when the weight is stored as a *float32*, that is, a float occupying 32 bits. This is the most used format to store neural network weights.

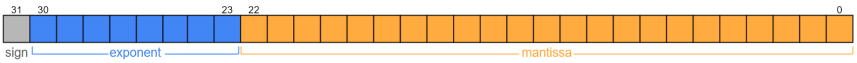

Figure 2.8: How a float32 is formatted

According to IEEE 754 [20], a float32 is composed of three components: a *sign*, an *exponent* and a *mantissa*. In particular, as shown in *Figure 2.8*:

- b_{31}: one bit is reserved to store the sign;
- $b_{30}...b_{23}$: eight bits to store the value of the exponent;
- $b_{22}...b_0$: 23 bits are used to store the value of the mantissa.

Given this representation, we can compute the decimal equivalent of a float32 binary representation as:

$$-1^{b_{31}} \cdot 2^{(b_{30}\ldots b_{23})_2 - 127} \cdot (1 + \sum_{i=0}^{22} b_{23-i} \cdot 2^{-i}) \tag{2.16}$$

As anticipated, faults have a different effect depending on which bit has been affected. In fact, when the fault hits a bit of the exponent the value of the weight might "*explode*" as its value increases significantly. Consequently, this fault might induce data corruption on an high number of inputs [21]. Conversely, faults impacting bits of the mantissa do not have the same effect, and the number of corrupted data tends to be much lower.

Sometimes, as a result of a fault, the network outputs can not be represented as number. In case a *vector score* contains *NaN* (not a number) values or infinite values, we can not talk about *Silent Data Corruption*, as it is easy to identify the presence a fault. For this reason, fault that have this effect should be filtered out in an analysis of the fault impact.

2.4 Fault Injections

Finally, in this section we are going to discuss *fault injections*, that is, how faults are simulated, both on hardware and in software. We are going to present an overview of which tools are available to simulate this kind of fault on a large scale and provide a brief overview of their advantages and disadvantages.

2.4.1 Hardware Fault Injections

A fault injection campaign can be carried out by replicating certain physical phenomenon that can lead to critical faults on the hardware. Among the experiments worth mentioning we cite the *neutron bean experiments* [22]. In this case, DNN-Accelerators are bombarded with neutron bean to simulate the effect of radiations on this kind of device. While this approach is likely to be the most accurate simulation of a malfunction, there are some drawbacks. The first is that an experiment like this is very expensive and time consuming. Moreover, the impact of a fault is intrinsically linked to the architecture of the accelerator we are using.

2.4.2 Software Fault Injections

Software fault injections can be inexpensive, especially in small networks. There are various approaches to simulate malfunctions. We can group fault injection methodologies under two categories.

First, we can simulate a fault *directly on the accelerator*. In this case, with libraries such as *NVBitFI* [23], *SASSIFI* [24] or *LLFI* [25], we can simulate a fault either in a registry or a memory location. This kind of tools are very powerful, and are used to simulate many kind of malfunctions in very different scenarios. As such, adapting them to work with neural network libraries, such as Tensorflow or PyTorch requires a non-trivial effort.

Conversely, more specific tools simulate fault *directly on the neural network*, modifying the values of the weights and the results of some operations. These instruments, such as the fault injector developed by *Politecnico di Torino* [21] or *TensorFI* [26] are developed to quickly simulate a fault injection. While simple to use, they are rather new. As such, documentation is scarce, and, more importantly, they are few works in literature that use them.

Chapter 3

Problem Formulation

As we have discussed, convolutional neural networks take advantage of *DNN accelerators*. Executing a large network, such as *VGG-16*, on this hardware, greatly reduces the time required to compute a prediction. Unfortunately, an accelerator is more likely to malfunction than a traditional hardware component such as a CPU. When a *fault* hits a neural network that is being executed on this kind of hardware, it can change the model prediction.

Let D denote VGG-16 and let $D(x) \in \mathbb{R}^{1000}$ denote the vector score computed by the network over input x. VGG-16 is used to solve a *classification problem*, where we want to assign a class to an input image. Let d denote the *predicted label* of the network for input x, defined as the component of the vector score of maximum value, that is:

$$d = \arg\max_j [D(x)]_j \tag{3.1}$$

The prediction of VGG-16 may be altered by the presence of one or more critical faults that modify the value of the weights. Let *VGG-16 faulty* be an instance of VGG-16 executed on malfunctioning hardware. The function associated to this network is D^f. Conversely, let *VGG-16 clean* be an instance of the network executed on an accelerator unaffected by fault. The function relative to this model is D^c.

We recall that we have defined a *critical fault* as a fault that changes the prediction of the network it affects. More precisely, a fault affecting a faulty instance of VGG-16 is critical when:

$$\arg\max_j [D^f(x)]_j \neq \arg\max_j [D^c(x)]_j \tag{3.2}$$

When a critical faults affects VGG-16, it worsen its performances, decreasing the accuracy of the network. As such, we are interested in designing a *fault detector FD* that take as input $D^f(x)$ and, tells us whether D^f is affected by a critical fault. The fault detector can be seen as a function $FD(D^f) : \mathbb{R}^{1000} \rightarrow \{0, 1\}$:

$$FD(D^f(x)) = \begin{cases} 1 & \text{critical fault has occurred} \\ 0 & \text{otherwise} \end{cases} \tag{3.3}$$

Designing a fault detector is a task that, in convolutional neural networks, has been often overlooked. This is mainly due two reasons:

- *Focus on passive error correction*: most academic works have focused on passive error correction, with varying results. These techniques often require additional operations or redundant architectures and therefore have a considerable computational cost;
- *Fault Detection as a Black Box*: in active fault detection, with detector-controller architectures, the focus is put on the latter. The detector is often treated as a *black box*[12] [13]. As such, there are few works that focus on designing the actual fault detector [19]. Notably, most of the literature on this topic is quite dated and is does not take into account convolutional neural networks.

Therefore, our goal is to design a fault injector that is:

- *Agnostic*: the fault detector has *no knowledge* of the clean network vector score $D^c(x)$;
- *Effective*: the accuracy, precision and recall of the fault detector must be high in different scenarios;
- *Inexpensive*: the computational cost of the fault detector must be low enough that it does not impact the execution of the network it is analyzing.

Chapter 4

Proposed Solutions

In order to identify whether a fault leads to a *critical data corruption* for the prediction of an input $x \in X$, we are going to look at the corresponding vector score $D^f(x)$. When a fault modifies the value of the weights, we expect one of the following:

- There is no data corruption.
- *Non-critical data corruption*: the fault changes the vector score but doesn't change the prediction;
- *Critical data corruption*: the fault alters the vector score and changes which component has the maximum value (3.2). This means that the prediction changes.

Clearly, we are only interested in identifying the last case. The best option would be to compare - for an input x - the prediction of the faulty network $D^f(x)$ with the prediction of the same clean network $D^c(x)$. The only possible way to achieve this in a real-world scenario is by *replicating* the whole network (*Figure 4.1*). Obviously, this is not a good solution as: (i) the computational cost is very high and (ii) there is no certainty that the replica network is not affected by a critical fault.

A possible solution, is to replace the replica with a simpler version of the network (*Figure 4.2*). In this case the *auxiliary network* is *smaller* than the main network and has a *similar* vector score. The auxiliary network acts as a supervisor for the main model: by comparing the vector scores of the two networks, we should be able to identify whether a fault changed the prediction of the main network. Differently from the previous case, if the auxiliary network has significantly less operations, we can ignore the

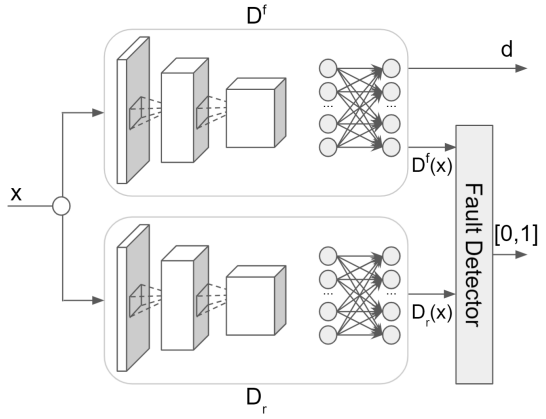

Figure 4.1: A fault detector that compares the vector score of a faulty network $D^f(x)$ with the vector score $D^c_r(x)$ of a replica.

computational cost. Furthermore, we can consider the auxiliary element as not affected by a fault if it is composed by remarkably less parameters, since the probability of a critical fault happening on the auxiliary network is much smaller than the probability of it hitting the main one. We call *Auxiliary-Based Fault Detector* a detector that uses an *auxiliary* network to flag *critical data corruptions*.

The other solution is to design a fault detector that uses only the vector score of VGG-16 (*Figure 4.3*). We call a fault detector that works without an

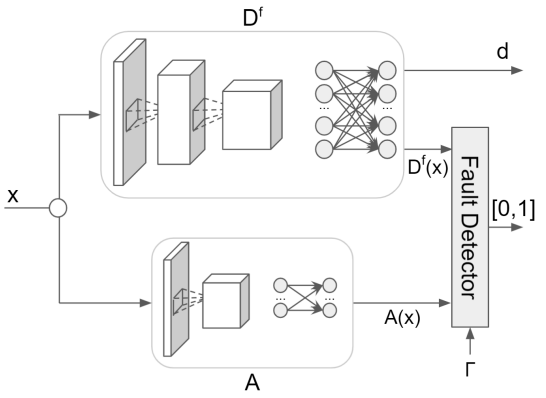

Figure 4.2: An *Auxiliary-Based Fault Detector* that compares the output of the faulty network $D^f(x)$ and the output of a smaller auxiliary network $A(x)$ with some external threshold value Γ.

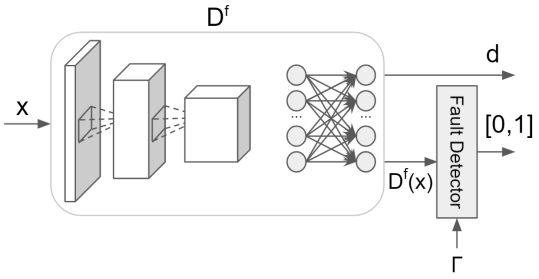

Figure 4.3: A *Score-Based Fault Detector* that compares the vector score of the faulty network $D^f(x)$ with some external threshold value Γ.

auxiliary network *Score-Based Fault Detector*, as they use only the vector score of the default network. Regardless of the presence of an auxiliary network, we design the fault detectors to work in the same way.

Let $i_x \in \mathbb{R}^M$ be the input of the fault detector corresponding to a VGG-16 input $x \in X$. The fault detector applies a function $F : \mathbb{R}^M \to \mathbb{R}$ to its input and compares the resulting value to a related *threshold* Γ_F, as shown in *Figure 4.4*. This means that we can re-write the equation of the function associated with the fault detector (3.3) as:

$$FD(i_x) = \begin{cases} 1 & \text{if } F(i_x) > \Gamma_F \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

The threshold value Γ_F is computed evaluating F over all the inputs $i_{\bar{x}} \; \forall \; \bar{x} \in \bar{X}$, where $\bar{X}$ is a dataset such that $\bar{X} \cap X = \emptyset$. Let K be the average value

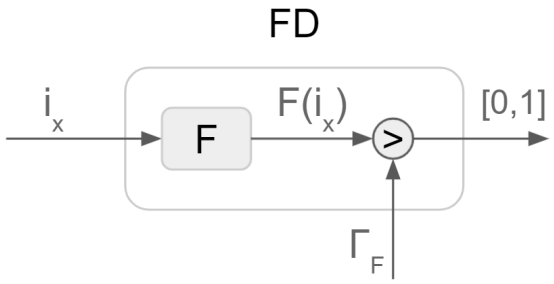

Figure 4.4: The fault detector *FD* evaluates a function F on its input i_x and compares it with a threshold Γ_F

of $F(i_{\bar{x}}) \; \forall \; \bar{x} \in \bar{X}$, we define the threshold Γ_f as:

$$\Gamma_F = \gamma \cdot K \tag{4.2}$$

Where $\gamma > 0$ is a multiplicative constant used to fine-tune the value of the threshold. The function used to compute the threshold is show in *Algorithm 2*. In this case F is the function computed by the fault detector over its input i_x, where $\bar{X}$ is the external dataset and γ is the multiplicative constant. This function needs to be called when we initialize the fault detector.

Algorithm 2 Compute Threshold

```
1: function ComputeThreshold(F, X̄, γ)
2:     K = average of F(i_x̄) ∀ x̄ ∈ X̄
3:     Γ_F = γ · K
4:     return Γ_F
5: end function
```

The fault detector executes the function DetectFaults(i_x, F) illustrated in *Algorithm 3*. In line 1 we initialize the threshold by calling the ComputeThreshold function. In line 3-7 we check whether the function F, evaluated for a given input i_x, is greater than the the threshold and, in that case, return 1.

Algorithm 3 Detect Faults

```
1: Initialize:
       Given γ > 0 and a dataset X̄
       Set Γ_F = ComputeThreshold(F, X̄, γ)
2: function DetectFaults(i_x, F)
3:     if F(i_x) > Γ_F then
4:         return 1
5:     else
6:         return 0
7:     end if
8: end function
```

In Section 4.1 we are going to present 2 thresholds for the *Auxiliary-Based Fault Detector*. Furthermore, we are going to present 4 different auxiliary network architectures. In Section 4.2 we are going to present 2 thresholds for the *Score-Based Fault Detectors*. This means that, in the end, we are going to present 10 different fault detectors.

4.1 Auxiliary-Based Fault Detectors

In this section we discuss 4 different architectures for an auxiliary network that tries to mimic the vector score of the main network (*Figure 4.2*). Clearly, we want these networks to be as small and computationally inexpensive as possible.

We refer to the *default* (or *main*) *network* as D and to the *auxiliary network* as A. We indicate with $D(x)$ the vector score computed by the main network over input $x \in X$, while with $A(x)$ the vector score computed by the auxiliary network over the same input. Our aim is to have a auxiliary network with a vector score as similar as possible to the one of the main network.

For this reason we are going to train the auxiliary networks using *knowledge distillation*. This allows us to have an auxiliary network that is trained to minimize the *k-d loss* (2.15). As a result, the auxiliary network vector score is similar to the main network vector score. A complete explanation can be found in Chapter 2. Having defined the loss function, we now define the auxiliary network topologies. We present two main classes of architectures. The first class uses a simplified version of VGG-16. The second class uses a *MobileNet* [2] with a very small α as auxiliary network. We are going to discuss these architectures in the following sections. For the reminder of this section we will discuss the architecture of the fault detectors.

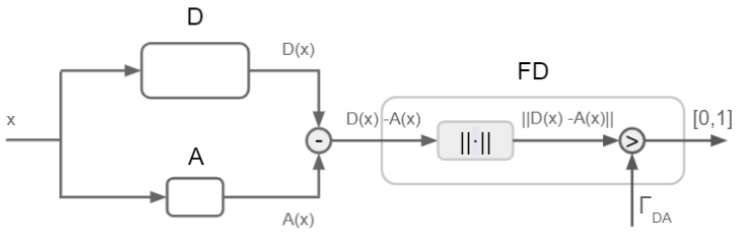

Figure 4.5: The Auxiliary-Based Fault Detector that uses the norm of the difference $||D(x) - A(x)||$.

The first fault detector, shown in *Figure 4.5* , is a *norm-based* fault detector that uses the norm of the the difference between the two networks vector scores as function F. To be more precise, the fault detector takes as input i_x, for $x \in X$ the difference between the vector scores:

$$i_x = D(x) - A(x) \tag{4.3}$$

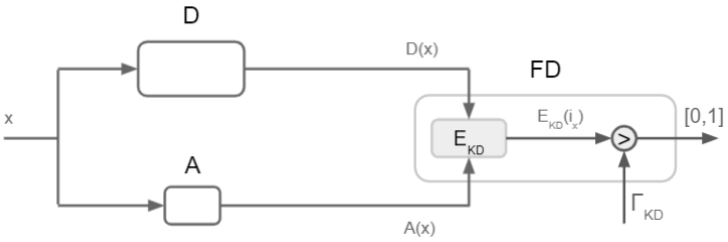

Figure 4.6: The Auxiliary-Based Fault Detector that uses the k-d loss $||E_{KD}(x)||$.

And applies to it a function $F_{DA} : \mathbb{R}^{1000} \to \mathbb{R}$ defined as:

$$F_{DA}(i_x) = ||i_x|| \tag{4.4}$$

From (4.2), (4.3) and (4.4) we obtain the equation for the computation of a threshold Γ_{DA}:

$$\Gamma_{DA} = \gamma \cdot \frac{\sum_{\bar{x} \in \bar{X}} ||D(\bar{x}) - A(\bar{x})||}{|\bar{X}|} \tag{4.5}$$

where $\bar{X}$ is a dataset such that $X \cap \bar{X} = \emptyset$.

In this case, the function implemented by the norm-based fault detector is a slight modification of the one described in *Algorithm 3*. In *Algorithm 4*, we can see the implementation of DetectFaultDA.

Algorithm 4 Detect Faults DA

Initialize:
    Given $\gamma > 0$ and a dataset $\bar{X}$
    Set $\Gamma_{DA} =$ ComputeThreshold$(||\cdot||, \bar{X}, \gamma)$
function DetectFaultsDA($D(x)$, $A(x)$, F)
    if $||D(x) - A(x)|| > \Gamma_{DA}$ **then**
        return 1
    else
        return 0
    end if
end function

The second fault detector we are going to introduced for the *Auxiliary-Based Fault Detector* is illustrated in fig. 4.6. In this case, the fault detector uses as function F the *k-d loss* $E_{KD}(x)$ (2.15). In particular, we are going to

redefine the k-d loss for two vector scores $D(x), A(x) \in \mathbb{R}^{1000}$ as :

$$E_{KD}(D(x), A(x)) = \sum_{i=1}^{1000} [D(x)]_i \log[A(x)]_i \tag{4.6}$$

We define this a *k-d loss-based fault detector*. The inputs of this fault detector are the two vector scores $A(x)$ and $D(x)$ for a given input $x \in X$. The function F is defined as:

$$F_{DA}(i_x) = E_{KD}(D(x), A(x)) \tag{4.7}$$

Where we use as student model the auxiliary network A and as teacher the default model D. From (4.2) and (4.7) we obtain the equation for the computation of a threshold Γ_{KD}:

$$\Gamma_{KD} = \gamma \cdot \frac{\sum_{\bar{x} \in \bar{X}} E_{KD}(D(\bar{x}), A(\bar{x}))}{|\bar{X}|} \tag{4.8}$$

Even in this case we can outline the function associated with the fault detector by modifying the general case described in *Algorithm 3*. The new function, DetectFaultsKD is shown in *Algorithm 5*.

Algorithm 5 Detect Faults KD

Initialize:
    Given $\gamma > 0$ and a dataset $\bar{X}$
    Set $\Gamma_{KD} = \text{ComputeThreshold}(E_{KD}(\cdot), \bar{X}, \gamma)$
function DetectFaultsKD($D(x)$, $A(x)$, F)
    Let D be the teacher model and A the student model
    if $E_{KD}(D(x), A(x)) > \Gamma_{KD}$ **then**
        return 1
    else
        return 0
    end if
end function

In the following subsections we are going to define the architecture of the auxiliary networks.

4.1.1 Auxiliary architecture: VGG-Like

The idea behind the *VGG-Like* models is to use an auxiliary network that is topologically similar to the main network. We think that a similar approach

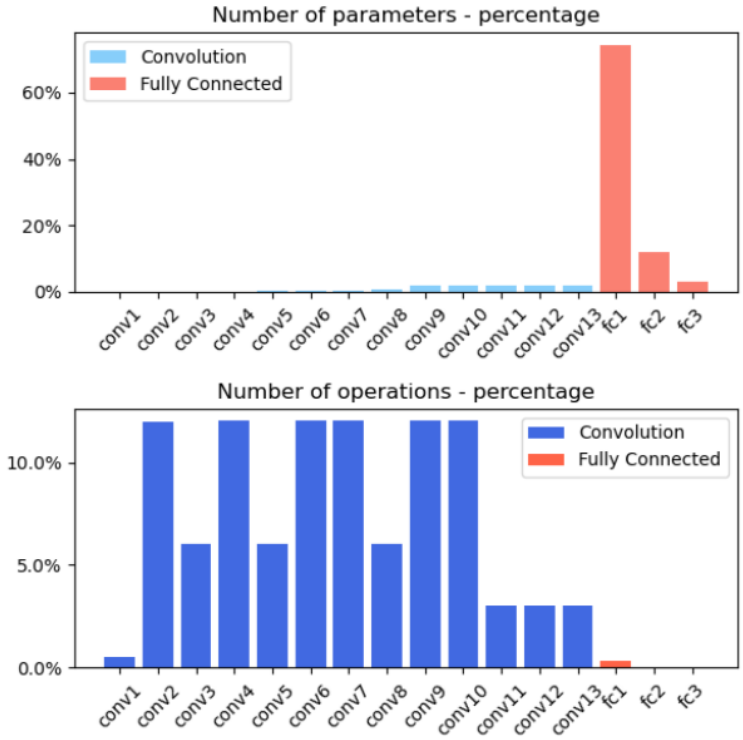

Figure 4.7: In this picture we can see the number of operations (top) and parameters (bottom) for each layer of VGG-16.

will favour the auxiliary model to produce vector scores close to the one computed by VGG. In this case, the challenge is to reduce the number of operations and parameters whilst maintaining a network that is somewhat significant.

As we can see in *Figure 4.7*, most of the operations of VGG are in the convolutional layers. Instead, the majority of parameters are stored in the first fully connected layer. Therefore, to reduce the computational complexity we should reduce the number of convolutional layers, while to decrease the number of parameters we should either reduce the size of the output of the last convolutional layer or reduce the number of nodes in the first fully connected layer.

We are going to present two different architectures. The first one, dubbed *VGG Slim* (Figure 4.8 - left), uses only one convolutional layer for each convolutional-pooling block to decrease the number of operations. In par-

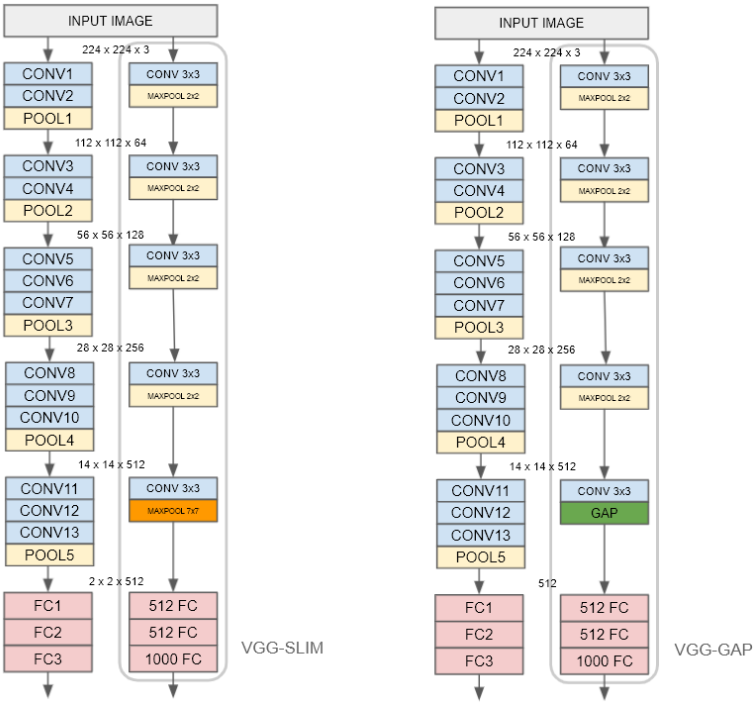

Figure 4.8: In this picture we can see the architecture for VGG-slim (left) and VGG-GAP (right). To reduce the number of operations, both models have less convolutional layer. To reduce the number of parameters, VGG-Slim reduces the size of the first fully-connected layer by using a stronger maxpooling, while VGG-GAP uses a GAP layer on the last convolutional layer.

ticular, it uses a stronger max-pooling (i.e. higher dimension of the operator) to decrease the number of parameters. The second architecture, called *VGG GAP* (Figure 4.8 - right) uses the same technique for the number of operations. However, it uses a *GAP* layer at the end of the convolutional part of the network to decrease the number of parameters. Both the networks use smaller fully connected layers.

We can see in *Figure 4.9* the number of operations and parameters of this two architectures compared with VGG-16. In particular, VGG Slim doesn't decrease significantly the number of operations while it decreases more the number of parameters. VGG GAP does better. As shown in the next section, the best result are obtained by using MobileNet.

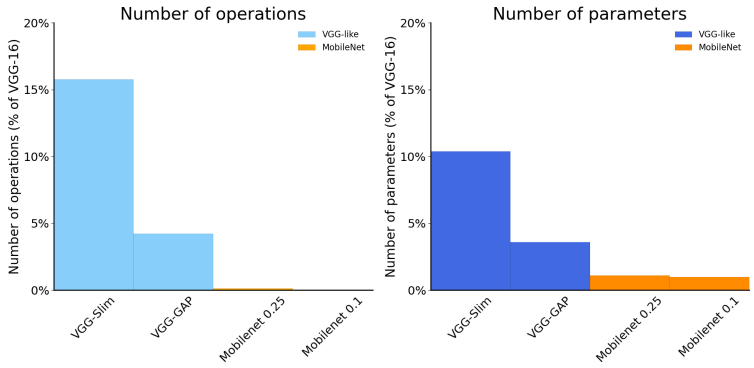

Figure 4.9: In this picture we can see the number of operations (left) and parameters (right) of the proposed auxiliary network architectures compared with VGG-16.

4.1.2 Auxiliary architecture: MobileNet

MobileNet is a convolutional neural network optimized for mobile devices that offers an accuracy similar to the one of VGG-16 with fewer parameters. In particular, for this topology we can specify the *width multiplier* α. If we take $\alpha < 1$ we can reduce the number of filters, therefore reducing the number of parameters and operations. This allows us to *trade-off* accuracy for a simpler network.

We will present two MobileNets, one using $\alpha = 0.1$ and the other using $\alpha = 0.25$ As we can see in *Figure 4.9*, the number of operations is much lower than the one of VGG-16. Similarly, as we have few parameters, we can maintain the hypothesis that the probability of a critical fault happening in the auxiliary network is negligible compared to the probability of it happening on the main network.

4.2 Score-Based Fault Detectors

As seen in literature, fault tends to introduce *critical data corruption* when they hit the most significant bits of the weight exponent [21]. We recall that weights are stored as *float32* in 4 bytes. Let's consider the case of a weight where a fault hits bit in position 30 flipping its value to 1. If the weight was positive - bit in position 31 is 0 - then the new value of the weight is in the range of 10^{38}. This value is clearly much higher than the average value of a weight. As such, we expect that the vector score computed by a network

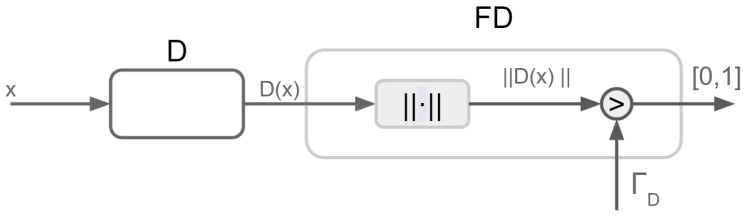

Figure 4.10: The Score-Based Fault Detector that uses the norm $||D(x)||$.

affected by this kind of faults is higher than a vector score computed by a clean network. Of course, this is not the only kind of fault that can alter significantly the vector score, but it gives us a rough idea of how many fault occur to produce at least a critical data corruption.

As we expect critical faults to have such a high impact on the value of the weight, and therefore of the outputs, the first *Score-Based Fault Detector* we present uses the norm of the vector score as function F. This solution, shown in fig. 4.10, is a *norm-based* fault detector. Clearly, for all the the *Score-Based Fault Detectors* it holds that:

$$i_x = D(x) \tag{4.9}$$

where i_x is the input of the fault detector and $D(x)$ is the vector score of VGG-16 for input $x \in X$. As we said, the function associated with fault detector $F : \mathbb{R}^{1000} \to \mathbb{R}$ is:

$$F(i_x) = ||i_x|| \tag{4.10}$$

From (4.2), (4.9) and (4.10) we can obtain the equation of the threshold Γ_D:

$$\Gamma_D = \gamma \cdot \frac{\sum_{\bar{x} \in \bar{X}} ||D^c(\bar{x})||}{|\bar{X}|} \tag{4.11}$$

Where $\gamma > 0$ is a constant multiplier and $\bar{X}$ is a dataset such that $X \cap \bar{X} = \emptyset$.

As for the *Auxiliary-Based Fault Detectors*, the function associated with this *Score-Based Fault Detector* can be implemented by slightly modifying the function shown in *Algorithm 3*. The pseudo-code for DetectFaultD is shown in *Algorithm 6*.

However, the norm might not be the best choice to compress the vector score. In fact, the same fault can have a different outcome. Imagine that, as a result of the fault, an element of the output of the second to last fully

Algorithm 6 Detect Faults D

Initialize:
  Given $\gamma > 0$ and a dataset $\bar{X}$
  Set $\Gamma_D = \text{ComputeThreshold}(||\cdot||, \bar{X}, \gamma)$
function DetectFaultsD($D(x)$, F)
  if $||D(x)|| > \Gamma_D$ **then**
    return 1
  else
    return 0
  end if
end function

connected layer is in the order of 10^{38}. If this element is multiplied by a negative weight, an element in the vector score might have a value in the order of -10^{38}. This is due to the fact that the last fully connected layer doesn't have a *relu* activation function that 'cleans' negative values.

Unless this negative component is the one corresponding to the predicted label, it doesn't change the network prediction. In this case, the fault is *not critical*. However, since this component increases the value of the norm, it is very likely that our previous fault detector flags the vector score as *critically corrupted*. Therefore, for this kind of faults, we need to apply a different function F to the vector score. Since we want to identify very high numbers, it makes sense to analyze the *maximum* element of the vector score. Furthermore, this idea is coherent with the selection of the prediction, making this a very promising way to identify *critical faults*.

Therefore, we define a second *Score-Based Fault Detector*, shown in fig. 4.11 that uses as function $F : \mathbb{R}^{1000} \to \mathbb{R}$ the maximum element of the vector score:

$$F(i_x) = \max_j [D(x)]_j \tag{4.12}$$

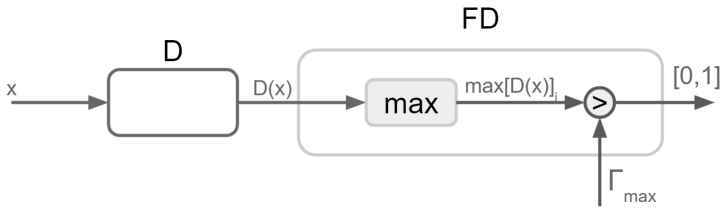

Figure 4.11: The Score-Based Fault Detector that uses $\max_j [D(x)]_j$.

However, in this case we will not use (4.2) to compute the threshold. Since we are using a maximum operator, it is more coherent not to use an average, but to select the maximum value ever recorded for $F(\bar{x})\ \forall \bar{x} \in \bar{X}$. Therefore, the threshold $\Gamma_{\max}$ for this fault detector is:

$$\Gamma_{\max} = \gamma \cdot \max_{\bar{x} \in \bar{X}} (\max_{j} [D(\bar{x})]_j) \tag{4.13}$$

For the same reason, we need to modify the function used to compute the threshold ComputeThreshold shown in *algorithm 2*. In *algorithm 7* we present a new function, called ComputeThresholdMax that computes (4.13).

Algorithm 7 Compute Threshold Max

```
1: function ComputeThresholdMax(X̄, γ)
2:     K = max_{x̄∈X̄}(max_j [D(x̄)]_j)
3:     Γ_max = γ · K
4:     return Γ_max
5: end function
```

Similarly to what described in algorithm 3, we can define a new DetectFaultsMax function that is employed by this fault detector. The pseudo-code for this procedure is shown in algorithm 8.

Algorithm 8 Detect Faults Max

```
1: Initialize:
       Given γ > 0 and a dataset X̄
       Set Γ_max = ComputeThresholdMax(X̄, γ)
2: function DetectFaultsMax(D(x), F)
3:     if max_j [D(x)]_j > Γ_max then
4:         return 1
5:     else
6:         return 0
7:     end if
8: end function
```

4.3 Proposed Solutions Recap

In total, we are going to present 10 different fault detectors, as shown in *Table 4.1*. Eight of them are *Auxiliary-Based Fault Detectors* while the

other two are *Score-Based Fault Detectors*. We are going to compare them and choose the best one.

Since *Auxiliary-Based Fault Detectors* use an auxiliary network as a controller, we expect good results in terms of fault detection accuracy. However, if critical faults change significantly the value of the elements in the vector scores, the contribution of an auxiliary network is marginal. Moreover, we expect different results based on the architecture of the auxiliary network. VGG-like models are likely to have a lower k-d loss: it is possible that this models perform better when using the loss-based fault detectors.

On the other hand, *Score-Based Fault Detectors* are *inexpensive* as they do not require any extra operations beside the computation of a norm or of a maximum. Contrarily, *Auxiliary-Based Fault Detectors* are quite expensive. If the accuracy of both classes of methods is similar, Score-Based detectors will be the best fault detectors due to their inexpensiveness. In particular, we expect the max-based fault detector to perform better than the norm-based one. As discussed in Section 4.2, there are some vector scores that may be wrongly identified as critically corrupted by the norm-based detector but not by the max-based one.

Typology	*F*	*Threshold*	*Auxiliary Model*
Auxiliary-Based	$\|\|D(x) - A(x)\|\|$	T_{DA}	VGG Slim
Auxiliary-Based	$\|\|D(x) - A(x)\|\|$	T_{DA}	VGG GAP
Auxiliary-Based	$\|\|D(x) - A(x)\|\|$	T_{DA}	MobileNet ($\alpha = 0.1$)
Auxiliary-Based	$\|\|D(x) - A(x)\|\|$	T_{DA}	MobileNet ($\alpha = 0.25$)
Auxiliary-Based	$E_{KD}(A(x), D(x))$	T_{KD}	VGG Slim
Auxiliary-Based	$E_{KD}(A(x), D(x))$	T_{KD}	VGG GAP
Auxiliary-Based	$E_{KD}(A(x), D(x))$	T_{KD}	MobileNet ($\alpha = 0.1$)
Auxiliary-Based	$E_{KD}(A(x), D(x))$	T_{KD}	MobileNet ($\alpha = 0.25$)
Score-Based	$\|\|D(x)\|\|$	T_D	-
Score-Based	$\max_j [D(x)]_j$	$T_{\max}$	-

Table 4.1: List of the proposed solutions

Chapter 5

Experimental Setup

In this chapter, we will discuss the tool used to execute the fault injection. Furthermore, we are going to talk about our experimental setup, from the training of the auxiliary networks to the implementation of the Fault Detectors.

In particular, we will first analyze why and how we chose extended the Fault Injector provided by *Politecnico di Torino*. Secondarily, we are going to provide a detailed overview of how we structure a faulty run. After that, we will present the setup for the computation of the metrics we are interested in. Finally, we will introduce the implementation and evaluation of the fault detector itself.

5.1 Selection of a Fault Injector

The fault injector we have used in our experiment is an extension of the one provided to us by *Politecnico di Torino*, which we will refer to as *PdT-FI*. While there are many other tools for this task, few provide a simple and streamlined process for fault injection on a neural network. In fact, before choosing this fault injector, we have explored other solutions.

The first fault injectors we have worked with were *NVBitFI* [23] and its predecessor *SASSIFI* [24]. These tools, developed by *NV Labs*, are designed to simulate actual malfunctions on a graphic card and are not application-specific. They can inject, among other things, a fault at run time in the GPU memory, simulating an actual temporary fault. However, they are not designed with neural networks in mind. Using them with either PyTorch or Tensorflow would have required us to re-write part of their libraries. We would have had to modify some CUDA functions - those used to com-

pute neural network operations on the GPU - to make them compliant with NVBitFI and SASSIFI requirements. This task was deemed too complex for this thesis, leading us to abandon these tools. Another tool we tried was *LLFI*[25]. However, we discarded it for reason similar to the one presented for NVBitFI and SASSIFI.

Finally, we began experimenting with *Politecnico di Torino*'s fault injector. Differently from the tools we have presented before, it is designed to work with neural networks only. In its initial state, it worked only with LeNet5, a small convolutional network [17]. However, it is straightforward to use, as it just requires inserting the number of desired faults to simulate a *fault injection campaign*. This tool doesn't run code on the GPU, as it is written in *C*. However, it simulates stuck-at faults caused by a malfunction on an accelerator. Moreover, by creating two *threads* - one for the injection and one for the evaluation - it allows us to inject faults in between two predictions. This fault injection capability is all we need for our experiment. Seen how simple it was to use, we began a collaboration with *Politecnico di Torino*. Moreover, we extended the tool to work with large networks, and in particular, with VGG-16.

5.2 Limitations of the Fault Injector

One of the accomplishment of this thesis is the extension of *PdT-FI* in order to make it work with VGG-16. We came up with a pipelined process to extend the fault injector: this procedure can be repeated to allow the fault injector to work with other architectures.

To better understand how we proceeded to implement this extension, we must first introduce *N2D2* [27]. N2D2 is an open-source framework for deep neural network simulation developed by CEA LIST. Our fault injector, PdT-FI, is based upon this framework. N2D2 allows us to specify a network topology either through a costume `.ini` file or by using the *ONNX* framework [28]. Furthermore, we can define a dataset, a validation/training split, and train the network. We can then use the trained model to compute a prediction for a new image. Importantly, N2D2 allows us to *export* the network. That is, we can create a *stand-alone*, *compilable* *C* code that can be used independently from N2D2 to execute the network. PdT-FI takes advantage of this export functionality to obtain the *C* code of a trained neural network. This code can be modified to inject faults into the network and simulate malfunctions.

Since N2D2 exports *C* code, the network runs on the CPU and not on

the GPU. The original algorithm was designed to run on a small integrated board that required *C* code. However, this was not an issue, since the network used was LeNet5, with very few parameters and was fast to train even without any accelerator. Unfortunately, as VGG-16 has more than 100 million parameters, it is very expensive to train without GPU support. Therefore, we chose not to train VGG-16 using N2D2. We preferred to do it in PyTorch and then export the weights in the same format used by N2D2. Furthermore, we have more freedom to pre-process the images with PyTorch than with N2D2. As such, the pre-processing is done in python as well. The resulting database is exported to N2D2 afterward.

5.3 Extension of the Fault Injector

In *Figure 5.1*, we can see the pipelined procedure that needs to be to extend the fault injector to any given network. We can do this thanks to some python script we have written. We adopted the same method to extend the original fault injector from LeNet5 to VGG-16.

The first step consists in the creation of the network in N2D2 and in the *C* export. During the second steps we train the network in Python/PyTorch and export of the weights. Furthermore, in this step we create and export a pre-processed dataset. Finally, in the last step, we merge the file we have

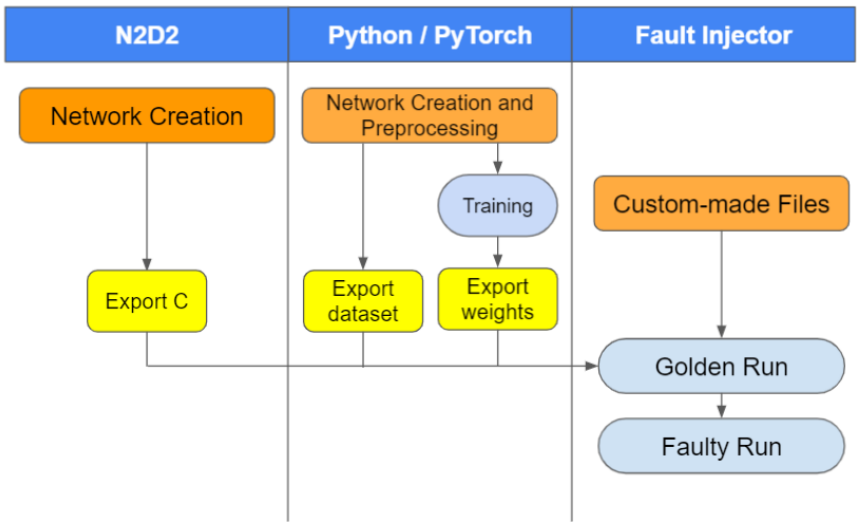

Figure 5.1: In this image we can see what operations need to be done in order to implement the fault injector for a new architecture

created in the previous steps with some custom-made files. At the end, we have a working fault injector, capable of executing *faulty* and *clean* (or *golden*) runs. In the following sections we are going to discuss into detail the fault injector extension procedure.

5.3.1 N2D2

The first task is to create the network in N2D2. As we already mentioned, this framework allows us to export a C compilable starting from a `.ini` file that describes the network. The output of this first step is just a combination of `.c` and `.h` files. The related compiled executable allows us to run inferences with the network. Notice that we need to repeat this step only if one wants to change the topology of the network.

5.3.2 Python / PyTorch

In this step, we train the network in PyTorch and export the weights in a format compatible with N2D2. Furthermore, we pre-process the images and create a dataset consisting of *`.ppm`* files, the format used by N2D2 for its input images.

Since we are using VGG-16, we did not train the network by ourselves, but we got pre-trained weights. To understand how these weights, which can be easily obtainable from the PyTorch API, are exported to N2D2, we must first understand how this framework stores the weights in the C exports. The files generated by N2D2 contain, among others, one `.c` file for each layer of the network. Each of these files contains the tensor of the weights, stored in a $W \times H \times C \times I$ matrix. The information about the layer, such as the width W, the height H, the number of channels C and the number of the input channels I, is stored in a header file having the same name as the `.c` one. In order to export the weights, we have to create a `.c` file, similar to the one generated by N2D2, starting from the values we obtained from PyTorch. As the process is quite linear and tedious, we won't go into detail on how this was achieved. The only other thing worth mentioning about this procedure is that we have to invert the dimension of the tensor storing the weights since in PyTorch this tensor is of size $C \times I \times W \times H$.

N2D2 is quite limited in terms of image pre-processing, therefore we carry out this process directly in python and export the processed dataset. As we discussed in chapter Chapter 2, the pre-processing step used for VGG are a mean normalization and a center crop. Both of them can be easily achieved by using some pre-processing function provided by the PyTorch API. Since

this pre-processing step transforms the images from a `.png` format to a 3-dimensional tensor and N2D2 only accepts `.ppm` images, we need to convert the tensors to the target format. A `.ppm` image contains a header that specifies the size of the image and the number of colors [29]. After that, it stores 2 bytes for every pixel of the image, each containing the encoding for the color of that pixel. N2D2 extends this standard, by using *4 bytes* for each pixel. This is done because this framework is not interested in storing images, but rather a (possibly pre-processed) float32 representation of the image. Furthermore, it adds a byte at the end that is used to encode the *label* of the image. As such, saving the matrix representation of a pre-processed image is quite simple: we just need to obtain the binary representation of each value and create a file containing the *header*, the binary representation of the *image*, and the binary representation of the *label*.

At the end of this step, we have a *dataset* of pre-processed `.ppm` images and a collection of `.c` files containing the *weights* of the trained network. These procedures need to be repeated whenever we change the dataset, the pre-processing functions, or the topology of the network.

5.3.3 Fault Injector

Finally, we are going to add together the files we have generated, the dataset, and some files from the PdT-FI that we have modified. While the original fault injector only needed a few tweaks to simulate malfunctions on VGG-16, its performances were abysmal. As the number of parameters was over ten thousand times greater than the one for LeNet5, we had to address issues caused by that.

We will only discuss the most significant changes made to some of the files of PdT-FI, as smaller corrections can be considered bug fixing. In particular:

- We created a file containing the global declaration of all the weight matrices. Previously, there were multiple declarations and definitions: the memory used to store these parameters was way higher than it should have been. Furthermore, weights were sometimes explicitly allocated inside function calls. By creating a single global declaration for these parameters and by having unique definitions, we decreased the memory usage and increased the performances. Our fault injector resulted ten times faster than the original one;

- We modified the main file `n2d2_test.c` that contained the main loop for the execution of the fault injector. In particular, we added the

possibility of saving intermediate outputs and a way to log the performances of our tool;

- We modified the python scripts that generated the fault list. Among the changes, we added the possibility to generate a fault in a layer with a probability proportional to the number of parameters in that layer. In other words, a fault has the same possibility of affecting every bit of every weight;
- We added a new python script that automates the fault injection procedures. In particular, for each run, it generates a new *seed*. This value is used to create a fault list. For very run, it executes multiple injection campaigns, each with a subset of faults form the list. We defined this way of injecting malfunctions *incremental fault injection*.

These files are added together with the ones generated at the previous step, thus obtaining our fault injections. The only file that needs to be modified if we want to change the topology of the network is the one generating the fault list, as we need to specify the parameters of each layer.

Once we have completed all the steps, we can perform a *golden run* and a *faulty run*. The golden run is used to obtained the predictions (and the vector score) over the dataset when there is *no fault injected* into the network parameters. Conversely, he faulty runs are used to obtain the same parameters when we simulate malfunctions in the model.

5.4 Running a fault injection campaign

Given our tool, the trained network weights and the dataset, the steps for running a simulation are the followings:

1. *Golden run*: Test the network over the training dataset;
2. Generate a random *seed* and use this number to create a *fault list* F consisting of N_F faults;
3. Select a subset of the fault list $f \subset F$ consisting of n_f faults;
4. For each seed and for each subset of the fault list:
    i. *Fault injection campaign*: Inject the fault from the set f into the network weights;
    ii. *Faulty run*: Compute the *vector score* $D^f(x)$ and the prediction $\arg\max D^f(x)$ for every image $x \in X$.

5.5 Metrics Computation

Having defined the fault injector and the steps of the simulations we are going to perform, we now discuss how we measure the impact of fault injections in VGG-16 . As we discussed in Chapter 3, we are looking for *critical data corruption*. That is, we want to find out when a fault occurred and when it changed the prediction for some input images. In particular, we are going to measure the impact of multiple faults through a reduction in *accuracy* of the network. Furthermore, we introduce a new metric, the *fault impact rate* to measure how many correct prediction were changed as a result of certain combinations of faults.

5.5.1 Fault Impact

To measure how impactful a set of fault is, we are going to introduce a new parameter, the *impact rate* r_i defined as:

$$r_i = \frac{\sum_{\dot{x} \in \dot{X}} \delta(\arg\max(D^c(\dot{x})) - \arg\max(D^f(\dot{x})))}{|\dot{X}|}, \tag{5.1}$$

where $\dot{X}$ is a subset of X such that $D^c(\dot{x}) = t \; \forall \; \dot{x} \in \dot{X}$, where t is the target output for input $\dot{x}$. This value is a way to measure how many correct prediction were *critically corrupted* thanks to fault injection campaign f.

We are going to measure this value for every run and for every number of faults, measuring how this values varies when we increase the number of faults. We expect it to be a monotonically increasing function. However, we are interested to see if the relation between the impact rate and the number of faults is comparable in every run.

5.6 Auxiliary Networks

Now that the framework for the fault injection is ready, we need to talk about the implementation of the *fault detectors*, starting from the training of the auxiliary networks for the *Auxiliary-Based Fault Detectors*.

Initially, we are going to discuss how we split the dataset, then we will talk about the training process.

5.6.1 Dataset Split

We recall that we are using the *Imagenet* dataset. In particular, since a labeled version of the test dataset is not available, we will use the validation

dataset to carry out our experiment. From now on, we will refer to this validation dataset simply as Imagenet. This dataset contains 50000 images, evenly distributed in 1000 classes. As summarized in *Table 5.1*, we split the Imagenet dataset in 3:

1. *Training dataset*: consists of 30000 images. It is used to perform the training of the auxiliary model.

2. *Validation dataset*: consists of 10000 images. It is used to perform *model selection*. Furthermore, this dataset is also used to compute the metrics analyzed by the fault injector. That is, the norm of the average value of the difference $||D(x) - A(X)||$ and the average k-d loss $E_{KD}(x)$ for the *Auxiliary-Based Fault Detectors*, the average norm $||D(x)||$ and the max value $\max D(x)$ for the *Score-Based Fault Detectors*.

3. *Test dataset*: consists of 10000 images. Used for the faulty run.

5.6.2 Training the Auxiliary Model

We trained the auxiliary model using knowledge distillation, minimizing the *k-d loss*. This kind of loss is computed by using a *teacher* model - VGG in our case - and a student model - the auxiliary network. By training the auxiliary network to minimize this loss with *high temperatures*, we obtain a vector score from the student that is similar in every component to the one returned by the teacher. We carried out this process in PyTorch, taking advantage of the *Distiller* library [30]. This framework allows us to perform various model compression tasks, including knowledge distillation. To train the auxiliary model, we need to compute the vector score of VGG in order to calculate the loss function. Distiller automatically does this, streamlining the training process.

Name	*Images*	*Purpose*
Training Dataset	30000	Train the auxiliary model.
Validation Dataset	10000	Model selection for the auxiliary model Computation of the thresholding metrics.
Test dataset	10000	Faulty run.

Table 5.1: Purpose of the dataset splits

We trained the model for 100 epochs, using *early stopping* to avoid overfitting. For the VGG-Like models, we also used *transfer learning*, as we it speeds up the training procedure.

5.6.3 Compute model prediction

The computation of the auxiliary model prediction over the test set needs to be done only once. As such, for each auxiliary model, we compute the vector score for each image of the test set before the execution of the faulty run and store the value in a *Pandas* dataframe.

We only simulate a clean run for the auxiliary model since we are supposing that the auxiliary network is significantly smaller than the main network. As such, the probability of a critical fault hitting the auxiliary network is negligible compared to the probability of it hitting VGG.

5.7 Fault Detector

Finally, we can talk about the implementation of the fault detector, talking about how we are going to measure the threshold and how we evaluate the performances of the different detectors.

The *Fault Detector* is a model that takes some vector scores as input and returns as output `True` if it detects a *critical fault* and `False` otherwise. This is illustrated in *Figure 5.2*. The *Auxiliary-Based Fault Detectors* take as input the vector scores of VGG-16 $D(x)$ and the vector score of the auxiliary network $A(x)$. The threshold examined by this model are Γ_{DA} and Γ_{KD}. The *Score-Based Fault Detectors* take as input only the VGG-16 vector score $D(x)$. In this case, the thresholds are Γ_D and $\Gamma_{\max}$.

The *Fault Detector* is very simple to implement. First off, it computes the metrics (either $||D(x) - A(x)||$, $E_{KD}(x)$, $||D(x)||$ or $\max D(x)$), then it compares this value with the threshold relative threshold. If the metric is larger than the threshold, then it outputs `True`, otherwise it outputs `False`.

5.7.1 Threshold computation

In order to compute our threshold, we need to create a *Pandas* dataset containing the vector score of VGG-16 and of the auxiliary networks computed over the validation dataset.

For the *Auxiliary-Based Fault Detectors*, the first threshold metric we need to computed is the average value of the norm of the difference Γ_{DA} (4.5). This is immediately done from the dataset we have built before.

Similarly, by using the two vector scores $D(x)$ and $A(x) \forall\, x \in X$ we can easily compute the *k-d loss threshold* Γ_{KD} as shown in (4.8).

For the *Score-Based Fault Detectors* we have to compute the average norm (4.11) and the max element (4.13). Both of this values can be easily computed from the same dataframe.

For each threshold we will try different value for the constant γ fine tune the fault detector.

5.7.2 Fault Detector Evaluation

To measure and compare the performances of the fault detectors, we will measure 3 values: the $accuracy^{FD}$, the $precision^{FD}$ and the $recall^{FD}$. These

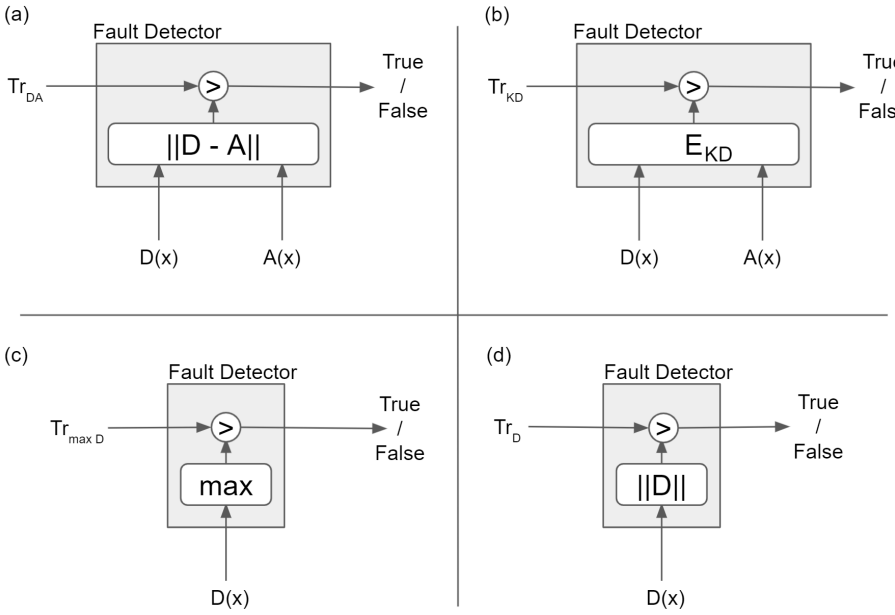

Figure 5.2: In this picture the detailed architecture of the fault injectors. In (a) the fault detector compares the norm of the difference of the two networks $||D(x) - A(x)||$ with the threshold Γ_{DA}. Similarly, in (b). the fault detector computes the k-d loss $E_{KD}(x)$ for input x and compares it with threshold Γ_{KD}. In (C) the fault injector compares the maximum element of the norm $max(D(x))$ with the threshold $\Gamma_{\max D}$. Finally, in (d) the fault detector compares the norm of the vector score $||D(x)||$ with the threshold Γ_D.

measurements can be computed as follow:

$$Accuracy^{FD} = \frac{TN + TP}{TN + TP + FP + FN} \tag{5.2}$$

$$Precision^{FD} = \frac{TP}{TP + FP} \tag{5.3}$$

$$Recall^{FD} = \frac{TP}{TP + FN} \tag{5.4}$$

Where TP (*True Positive*) are the critical data corruption that are correctly classified as such, FP (*False Positive*) the non critical data corruption that are mistakenly classified as critical, FN (*False Negative*) the undetected critical SDC and TN (*True Negative*) are the non critical data corruption that are correctly reported.

The $accuracy^{FD}$ measure how well the detector correctly classifies critical data corruption and non critical data corruption. With $precision^{FD}$ we can see if the network is detecting more outputs as critically corrupted than it should. Finally, $recall^{FD}$ shows how many critical data corruptions go undetected. These metrics are crucial in a *safety critical environment*, where even small mistakes can be quite costly. Therefore, all of them needs to be as high as possible.

First we are going to observe the average accuracy, precision and recall for different values of the constant γ . This gives us a rough idea on which method is the best.

However, computing the average accuracy, precision and recall is not the best way of measuring the fault injector performances. In fact, these values can be heavily influenced by outlier runs. For example, a very small number of false negative and no false positive result in a precision of 0 that negatively affects the average. To adjust for this, we are not going to compute them for every run and for every fault number. Instead, we will *sum up* all the TP, FP, FN and TN for the same number of faults - over different runs - and evaluate the accuracy, precision and recall over the aggregated results. From this we can gather the average metrics for a given fault number. This allows us to plot 3 graphs for every fault detector. On the x-axis we have the number of faults, while on the y-axis the accuracy, precision and recall of the model.

Chapter 6

Results

In this chapter we are going to illustrate the *results* of our experiments. First off, we are going to show what is the *effect of faults* on VGG-16, in terms of loss of accuracy and impact rate. After that, we are going to discuss how well auxiliary models approximate the main network. Finally, we are going to discuss the result obtained by the *Auxiliary-Based* and the *Score-Based* fault detectors.

6.1 Fault Injections Result

We execute 6 different runs generating a *fault list* for each one of them. From each list, we select different *incremental subsets* of faults and we inject them into the weights of VGG-16. This procedure is called *fault injection campaign*. After each injection, we evaluate the *faulty network* over the whole *test dataset* to compute the network *accuracy* and the *impact rate*.

The first 3 runs (*Figure 6.1*, left column) span over a large interval of faults. This was done in order to find out how many faults to inject in the network to have interesting results. *Run #1* and *Run #3* show that, for a small number of faults, the accuracy of faulty VGG-16 is similar to the one of clean VGG-16. However, by increasing the number of faults, the accuracy slowly drops to the one of a random classifier. In *Run #2* we have that 10 faults are enough to significantly decrease VGG-16 accuracy.

Since we are interested in studying critical faults, it makes sense to focus only on a number of fault that is high enough to significantly reduce the network accuracy. Based on the result of the first 3 runs, alongside with our hypothesis that one in 128 faults is potentially critical, we focused our run on a smaller interval. Particularly, in *Run #4*, *#5* and *#6* we injected

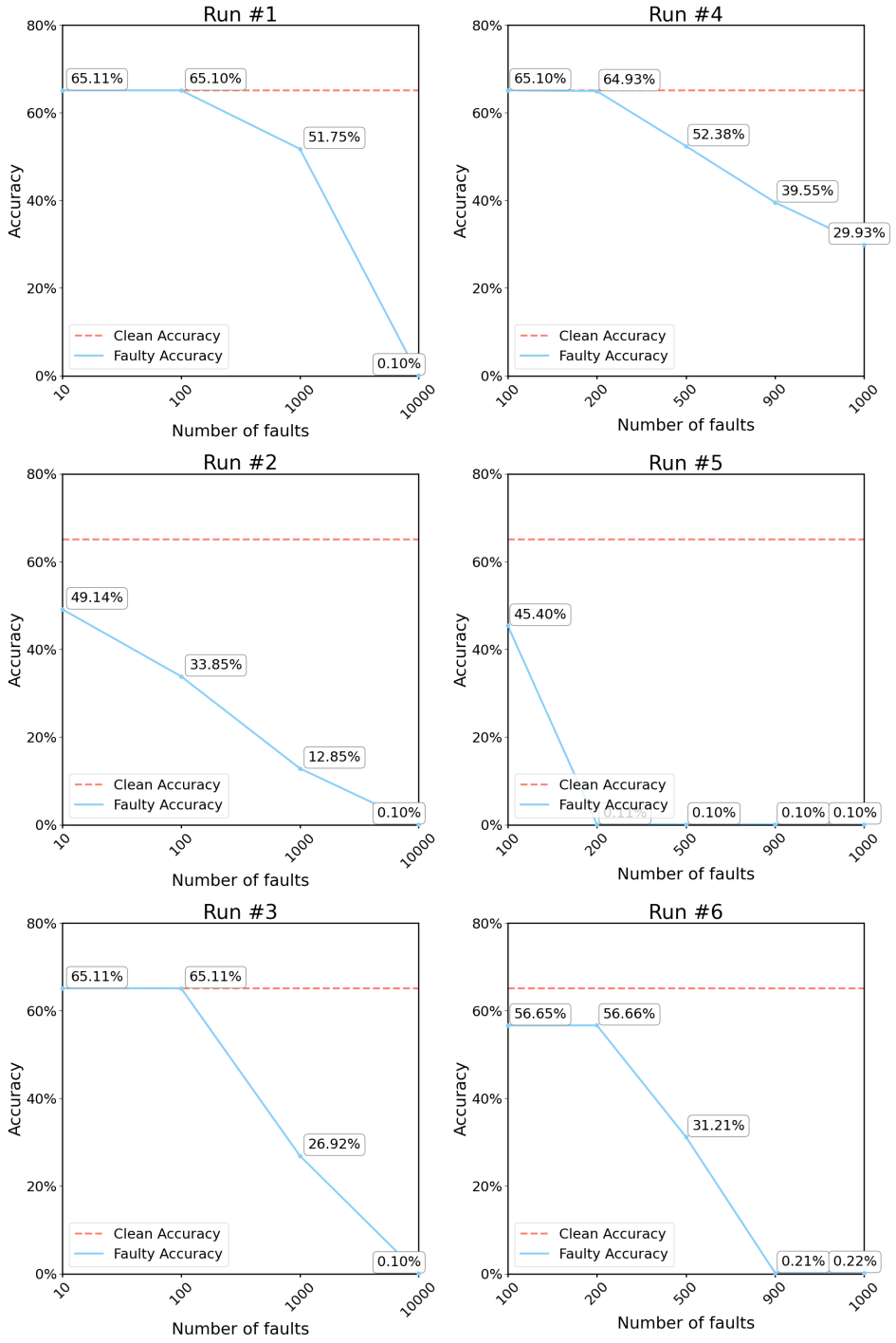

Figure 6.1: The *accuracy* of a faulty VGG-16 over different run. We can see how the accuracy worsen as more fault are injected into the network.

between 100 and 1000 faults. It is pointless to study what happens with an higher number of faults, as the performances of faulty VGG-16 are the same of a random classifier. Therefore, for the last runs (*Figure 6.1*, right column), we injected up to 1000 faults. These 3 simulations have different effects on how quickly VGG-16 accuracy worsen. However, in all the runs, the accuracy at the end is significantly worse than it was at the beginning. To have a better overview on why this happens, let's examine the *impact rate* r_i (equation 5.1)

The impact rate (*Figure 6.2*) confirms what we have seen before. Crucially, we can notice that the number of fault is not indicative of the number of critically corrupted outputs. In fact, *Run #5* shows that 200 faults are enough to completely change the network predictions. Contrarily, 1000 faults injected in *Run #1* change only 20% of the predictions.

6.2 Auxiliary Models Training

The first step into the realization of the *Auxiliary-Based Fault Detector* is the training of the auxiliary model. In the previous chapter, we have presented 4 different networks. Two of them are *VGG-like*, that is, they are built with a topology similar to the one of VGG-16. The other two are *MobileNets* with $\alpha = 0.1$ and $\alpha = 0.25$. We have trained these models over the *training set* composed of 30000 images and we evaluated them over the *validation set* of 10000 images. To perform model selection, we used *early stopping* - with a *patience* of 5 epochs. The training was executed over 100 *epochs*. The metrics we are going to present are obtained by evaluating the networks over the *test dataset*.

First, we discuss the results of the training of the VGG-like models. For these models we have used *transfer learning* to speed up the process. *VGG Slim*, the most parameter-heavy model we have presented, reached a validation *accuracy* of 22.05%, and a *relative accuracy* of 23.30% (*Table 6.1*). This last parameter is the percentage of input for which the auxiliary network predicts the same output as the auxiliary model. These are good results, since wanted to build a fairly inaccurate model, as we do not want to create a replacement of the network, but rather an *approximation*. While the relative accuracy seems low, it is not worrisome. In fact, the training for *distillation learning* is done with an high temperature. Therefore, the student model does not need to have the same prediction: what is important is that the vector scores of the two networks are similar. In this case, we

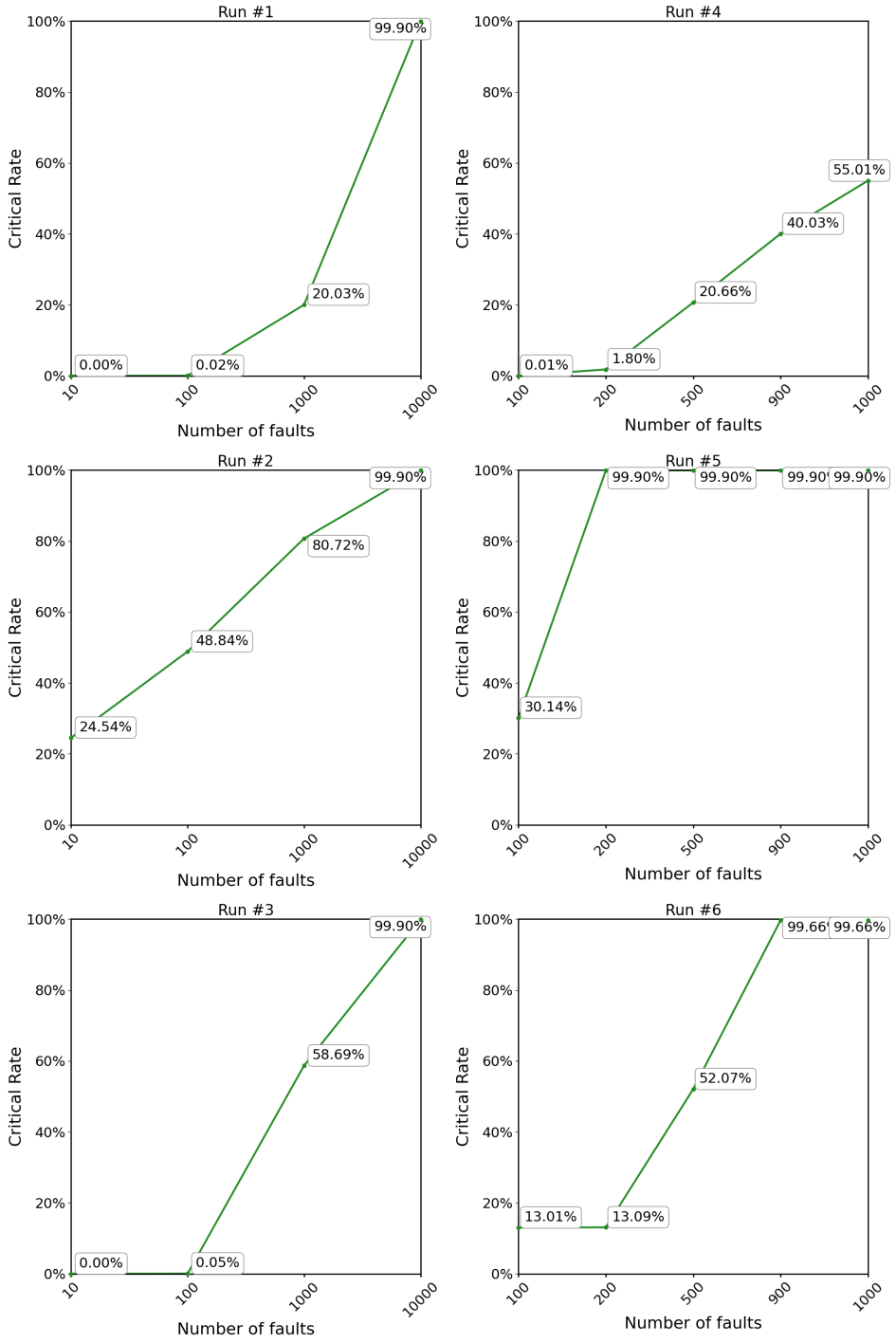

Figure 6.2: The *impact rate* of a faulty VGG-16 over different run.

have achieved that, as the *k-d loss* is only of 2.29.

For what concerns *VGG GAP*, we see that it has a worse accuracy than its Slim counterpart. Clearly, this is due to the smaller number of parameters present in this network, as the two topologies are quite similar. Notice that, despite having less than one third of VGG Slim parameters (as shown in *Figure 4.9*) the accuracy is only slightly worse. What is more concerning is the higher *k-d loss*: an higher value of this parameters suggests that VGG GAP is worse at approximating VGG-16 vector score than VGG Slim.

Moving on the the *MobileNet* architectures, we see an improvement of the accuracy of the model. In particular, for $\alpha = 0.1$, we have an accuracy of 20.86% and a relative accuracy of 22.04%. This result are in line with those achieved by VGG Slim, despite having less than one tenth of its parameters and operations. However, the k-d loss is still in favour of the VGG-like model. In fact, with a loss of 3.35, the results are even worst than the ones of VGG GAP. This shows that MobileNet is clearly better than the other models in predicting the teacher network output. However, using an architecture similar to the one of the teacher, improves the student ability of learning its vector score. Finally, MobileNet with $\alpha = 0.25$ improves its smaller counterpart accuracy, moving to an accuracy of 32.90% and a relative accuracy of 35.05%. By slightly increasing the number of parameters, we have significantly improved the network performances. Even the k-d loss jumps down to 2.35, reaching a value similar to the one obtained by VGG Slim. However, the number of operations and parameters of this network are just a fraction of the VGG-like model.

The result of this training clearly shows that having a small, well documented architecture provides a more accurate approximation of the main model. Furthermore, despite having less than one tenth of VGG Slim pa-

Name	*Accuracy*	*Relative Accuracy*	*K-D Loss*
VGG Slim	22.05%	23.30%	2.29
VGG GAP	18.84%	20.41%	3.07
MobileNet ($\alpha = 0.1$)	20.86%	22.04%	3.35
MobileNet ($\alpha = 0.25$)	32.92%	35.05%	2.35

Table 6.1: Accuracy, Relative Accuracy and K-D Loss of the auxiliary models over the *test* dataset.

rameters, MobileNet with $\alpha = 0.25$ provides similar k-d loss. From this preliminary results, this latter network could be the better solution for a fault detector architecture with an auxiliary network.

There is a possibility that, using *larger* values for the the *width multiplier*, the loss of the auxiliary model will be even smaller. However, the number of parameters of the network is *not linearly dependent* on the value of α. As such, parameter acts as a *trade-off* between the complexity of the auxiliary model and its accuracy.

6.3 Auxiliary-Based Fault Detector Results

Once we have trained the auxiliary model, we can use it in our Fault Detector. As such, in this section, we will discuss what happens when we use the *Auxiliary-Based Fault Detector*. First we are going to talk about the model that uses the norm of the difference to make a prediction, and then we are going to discuss what happens when we use the k-d loss based fault detector.

6.3.1 Norm of the Difference

First, we will talk about the norm-based fault detector, that uses as threshold Γ_{DA}. It compares this value to the norm of the difference between the main and the auxiliary vector score $||D(x) - A(x)||$. Since values in the vector score reach very high values, the norm sometimes goes to *infinite*. In this case, we assume that the fault are easily identifiable and we do not report these occurrences in our results.

In *Table 6.2* we can see the performances of the *fault detector* for multiple values of γ. We can observe how all the models perform roughly the same in terms of accuracy. This means that *distillation learning* is effective in finding a vector score similar to the one of the teacher model. However, there are big differences when it comes to the precision and recall. We can see that the *recall*, no matter the value of the multiplier, stays around 80% with a very high variance. This tells us that, in some runs, the number of *false negative* is very high. Conversely, the *precision* depends on the model and on γ. With γ close to one, MobileNets have an higher precision than the VGG-like models, with a very smaller variance. This means that the number of false positive is very small when using a MobileNet architecture. The precision of the VGG-like models improves when we increase the value of γ, but it never reaches comparable values.

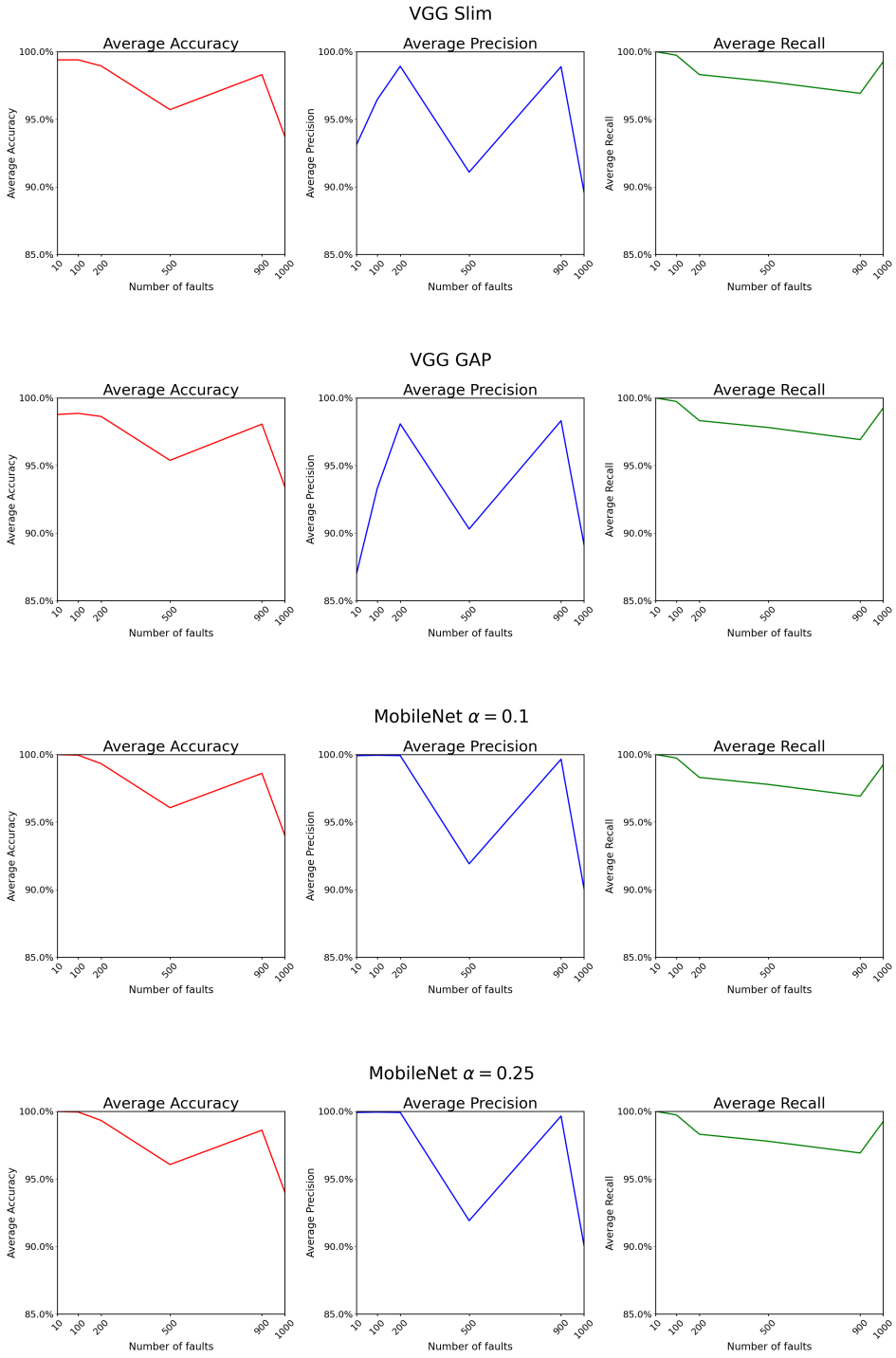

Figure 6.3: Accuracy, Precision and Recall of the *Auxiliary-Based Fault Detector* using the norm of the difference $||D(x) - A(x)||$ for different number of faults.

Auxiliary-Based with Γ_{DA}	*Accuracy*^{FD}		*Precision*^{FD}		*Recall*^{FD}	
($\gamma = 1.5$)	mean	variance	mean	variance	mean	variance
VGG Slim	95.62%	14.25	68.01%	1576.89	82.00%	1403.76
VGG GAP	96.18%	13.44	69.04%	1605.69	82.03%	1399.98
MobileNet ($\alpha = 0.1$)	97.93%	12.79	72.20%	1741.03	81.92%	1415.23
MobileNet ($\alpha = 0.25$)	97.66%	12.47	71.67%	1703.99	81.95%	1411.40
($\gamma = 1.9$)						
VGG Slim	97.56%	12.64	71.47%	1707.66	81.92%	1415.37
VGG GAP	97.29%	12.73	71.06%	1679.56	81.98%	1407.59
MobileNet ($\alpha = 0.1$)	98.02%	12.88	95.28%	125.96	81.92%	1415.23
MobileNet ($\alpha = 0.25$)	98.02%	12.88	95.28%	125.96	81.92%	1415.23
($\gamma = 2$)						
VGG Slim	97.69%	12.74	71.73%	1719.65	81.92%	1415.37
VGG GAP	97.36%	12.71	71.20%	1684.91	81.98%	1407.59
MobileNet ($\alpha = 0.1$)	98.02%	12.88	95.28%	125.96	81.92%	1415.23
MobileNet ($\alpha = 0.25$)	98.02%	12.88	95.28%	125.96	81.92%	1415.23

Table 6.2: Accuracy^{FD}, Precision^{FD} and Recall^{FD} of the Auxiliary-Based Fault Detector with threshold Γ_{DA}, mean and variance. $\gamma = 1.5, 1.9, 2$

As we discussed at the end of the previous chapter, these metrics do not paint a complete picture of the fault detector performances. We are now going to present the metrics computed over the data *grouped by* the number of faults. In *Figure 6.3*, we can see how the accuracy, precision and recall of the model varies for different number of faults.

The first thing we can notice is how bad the accuracy and precision of the VGG-like models is compared with the MobileNets. This is particularly true for smaller number of faults. Furthermore, we can see that both the MobileNet architectures have *exactly* the same performances. Since $\alpha = 0.1$ means having a smaller network than using $\alpha = 0.25$, the latter is a worse candidate for our *Auxiliary-Based Fault Detector*.

Moreover, we can see that, for all the models, the recall is quite stable, never dropping under 95%. As we supposed before, the average recall is not

indicative of what is actually happening. in fact, runs with a small critical rate - such as *Run #4* with 200 faults - may have a small recall to due the absence of true positives and the presence of very few undetected critically corrupted data. By aggregating the data for different runs, we can provide a more accurate evaluation of the recall.

6.3.2 K-D Loss

We will now talk about the fault injector that uses the *k-d loss* to decide whether an input is critically corrupted or not. In this case, the threshold used is Γ_{KD}. As before, values that go to infinite are not considered in this presentation. In *Table 6.3* we can see how the fault injector performs for different values of the multiplier γ. It is quite clear that this method provides worst results than the norm-based fault detector. Increasing the

Auxiliary-Based with Γ_{KD}	$Accuracy^{FD}$		$Precision^{FD}$		$Recall^{FD}$	
($\gamma = 1.5$)	mean	variance	mean	variance	mean	variance
VGG Slim	62.18%	1394.48	62.06%	1492.62	57.80%	2057.54
VGG GAP	63.13%	1436.11	63.18%	1498.07	54.28%	2040.27
MobileNet ($\alpha = 0.1$)	62.59%	1409.64	62.48%	1492.55	53.15%	2200.91
MobileNet ($\alpha = 0.25$)	61.01%	1346.69	61.01%	1489.74	60.45%	1949.51
($\gamma = 2$)						
VGG Slim	66.70%	1605.31	68.13%	1563.15	54.37%	1997.81
VGG GAP	66.74%	1613.69	69.28%	1584.05	48.76%	1903.109
MobileNet ($\alpha = 0.1$)	67.99%	1672.19	70.24%	1612.58	50.95%	2090.49
MobileNet ($\alpha = 0.25$)	66.94%	1617.79	67.61%	1554.30	56.47%	2146.93
($\gamma = 3$)						
VGG Slim	65.51%	1605.20	72.69%	1764.91	39.24%	1245.23
VGG GAP	63.36%	1552.32	72.49%	1750.46	34.26%	980.23
MobileNet ($\alpha = 0.1$)	64.52%	1598.93	74.16%	1834.73	36.33%	1062.02
MobileNet ($\alpha = 0.25$)	69.02%	1739.49	74.43%	1747.69	47.79%	1866.69

Table 6.3: Accuracy^{FD}, Precision^{FD} and Recall^{FD} of the Auxiliary-Based Fault Detector with threshold Γ_{KD}, mean and variance. $\gamma = 1, 2, 3$

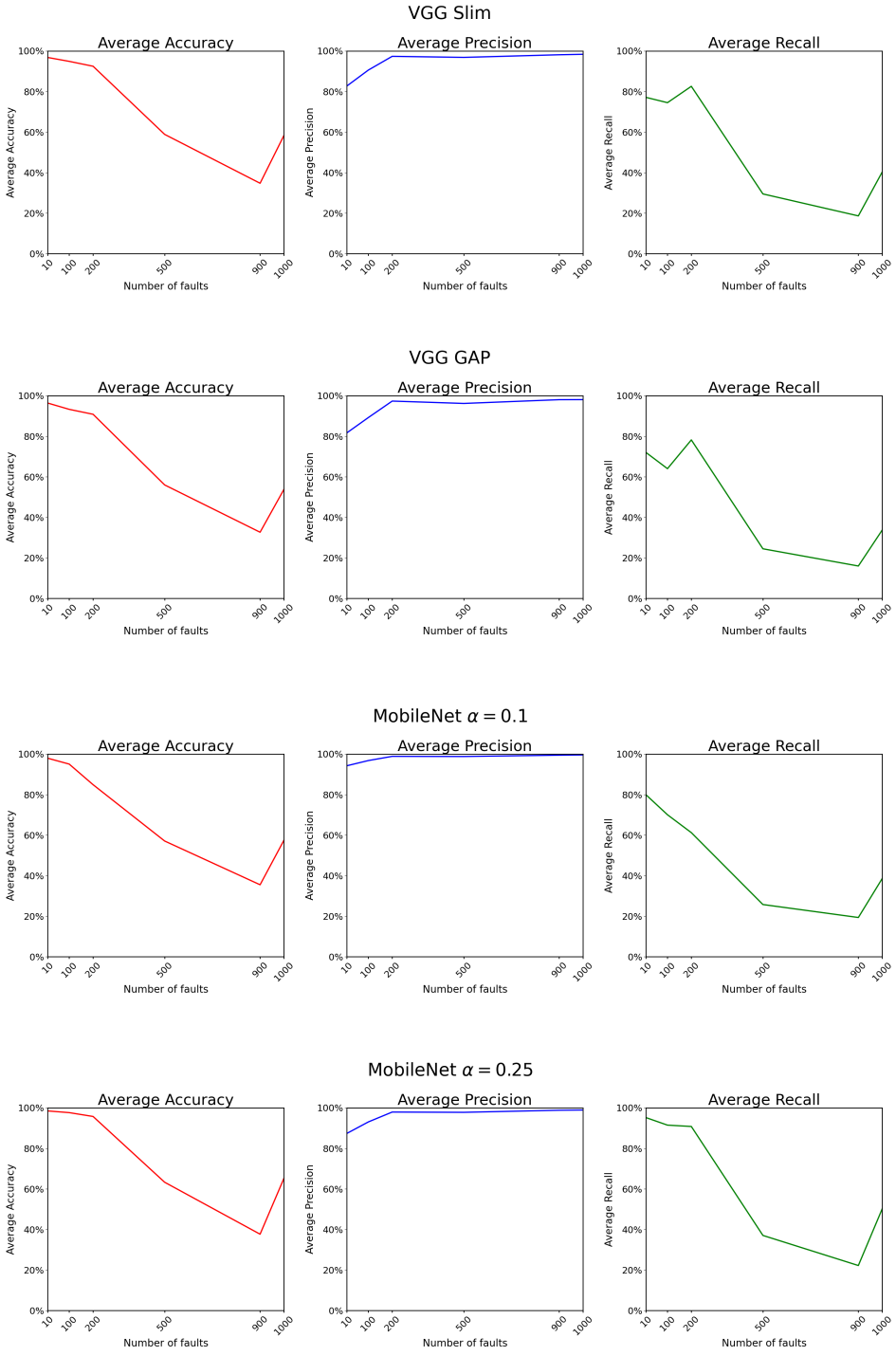

Figure 6.4: Accuracy, Precision and Recall of the *Auxiliary-Based Fault Detector* using the *k-d loss* $E_{KD}(x)$ for different number of faults.

value of the multiplier slightly improves the accuracy and the precision of the detector. However, we can notice how, even if with $\gamma = 2$ we have a better average *accuracy* than with $\gamma = 1.5$, we have worse *recall*. In fact, recall worsen when we increase γ: this parameter acts as a trade-off between accuracy/precision and recall. Furthermore, we can notice a very high variance for all the metrics. This means that this kind of fault injector is very *unreliable*, and, while it may work perfectly in certain scenarios, is completely useless in others.

Moving on to the aggregate analysis shown in *Figure 6.4* we can see yet again the poor performances of this fault detector. In this case, the metrics are computed for $\gamma = 3$. We can immediately notice three thing about these results: first off, only the precision is high and constant. Secondly, accuracy and recall have a high variance with regard to the number of faults. Even if the fault detector works sufficiently well for very low number of faults, its performances worsen significantly when we increase the number of faults. Finally, all the models have roughly the same scores, This means that - since the k-d loss is very similar for all the models - to have a more performing loss-based fault detector we would need an auxiliary network that decreases even further the k-d loss.

As a final consideration, we observe that, for all the models, the norm-based approach works better than the loss-based one. Particularly, while results may be comparable for a *small number* of faults, it is clear that looking at the norm is preferable when there are *more faults*.

6.4 Score-Based Fault Detectors Results

After having analyzed the performances of the *Auxiliary-Based Fault Detectors*, we will now focus on the *Score-Based Fault Detectors*. We recall that the main difference between the two models is that the former take advantage of an auxiliary network that mimics VGG, while the latter use only the output of the main network.

For this kid of fault detectors, we have presented two possibilities. In the first solution, the norm of the vector score is compared with threshold Γ_A (*norm thresholding*). In the second solution, the component of maximum value is compared with $\Gamma_{\max A}$ (*max thresholding*). As for the Auxiliary-Based Fault Detectors, if the norm of the vector score $||D(x)||$ or the maximum value of the component $\max(D(x))$ is infinite, it is not included in the computation of the metrics.

First off, we observe what happens for different values of the multiplier

Auxiliary-Based with Γ_{KD}	*Accuracy*^{FD}		*Precision*^{FD}		*Recall*^{FD}	
($\gamma = 1.5$)	mean	variance	mean	variance	mean	variance
$\|\|D(x)\|\|$	98.02%	12.88	95.28%	125.96	81.92%	1415.23
$\max(D(x))$	99.70%	0.25	99.94%	0.00	81.93%	1415.61
($\gamma = 1.9$)						
$\|\|D(x)\|\|$	98.02%	12.88	95.28%	125.96	81.92%	1415.23
$\max(D(x))$	99.70%	0.25	99.94%	0.00	81.93%	1415.61
($\gamma = 3$)						
$\|\|D(x)\|\|$	98.02%	12.88	95.28%	125.96	81.92%	1415.23
$\max(D(x))$	99.70%	0.25	99.94%	0.00	81.93%	1415.61

Table 6.4: Accuracy^{FD}, Precision^{FD} and Recall^{FD} of the Score-Based Fault Detector, mean and variance. $\gamma = 1, 1.9, 2$

γ. This is shown in *Table 6.4*. At a first glance, we can notice how the performances are unaffected by the value of the multiplier. Furthermore, the performances of the norm-based fault detector are exactly the same as the performances obtained by the best *Auxiliary-Based Fault Detector*. This suggests that the usage of an auxiliary network does not help the fault detector in performing his task. On the other hand, by comparing the result of the two Score-Based methods, we can notice how the fault detector using the max of the vector score clearly outperforms the one using the norm. The *accuracy* and *precision* are over 99% with a very small variance. This means that the max-based fault detector has close to none False Positive. However, the *recall* of the model is still quite low, reaching only 80% with a very high variance. When we will discuss the aggregate metrics, we will see that this is due to noise.

As we said, the norm-based fault detector has the same metrics of the *Auxiliary-Based Fault Detectors*. The reason for this can be found in the value of the vector score for the *critically impacted outputs*. Often, one or more components of the faulty vector score are in the order of 10^{38}, while the average value of the *clean vector score* is usually lower than 10^3. Since the value of the vector score of the auxiliary network A is typically in the same range, it is quite clear why it does not provided added value. By subtracting to a number in the order of 10^{38} a number in the order of 10^3 , we still obtain a number in the former range. Since the threshold is comparable - in value

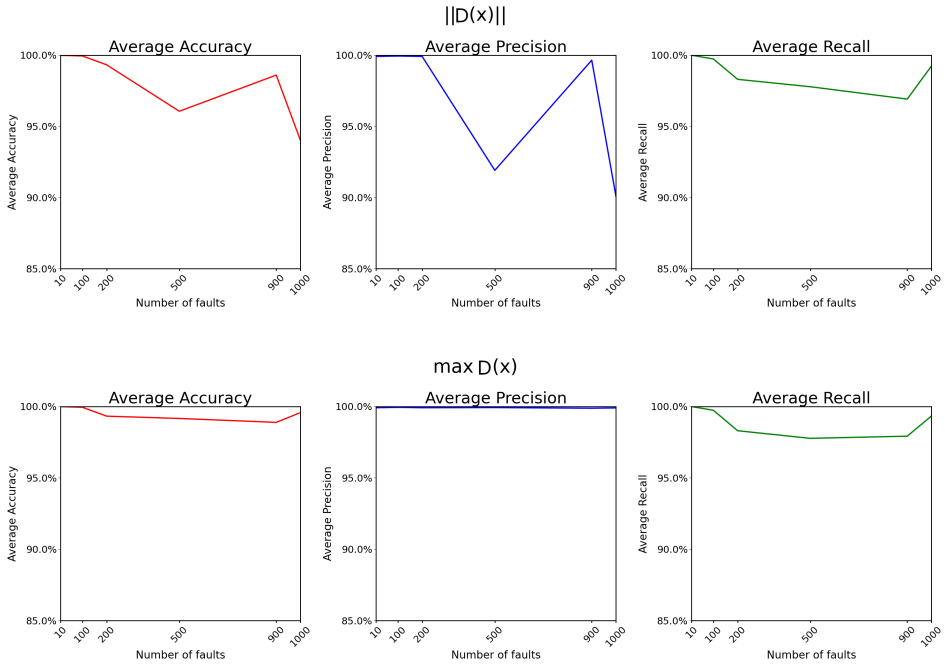

Figure 6.5: Accuracy, Precision and Recall of the *Score-Based Fault Detectors* for different number of faults.

- to a clean vector score, the fault detector is unable to distinguish between such big number that have a very small difference.

We conclude the analysis of the *Score-Based Models* by discussing the aggregate metrics we have already presented for the *Auxiliary-Based Fault Detectors*. The accuracy, precision and recall of these method are visualized in *Figure 6.5*. Even in this analysis, it appears clear that the norm-based solution has the same performances of the best auxiliary-based model, confirming what we have said while talking about the average values. However, the most interesting observation we can make on these results is how the max-based approach outperforms the norm-based method *for every number of faults*. Moreover, for this fault detector, the accuracy, precision and recall are always over 98% and are minimally affected by the number of faults.

6.5 Final Considerations

To conclude this chapter, we will briefly summarize the results of our experiments, showing what we have achieved. Initially, we have seen how the

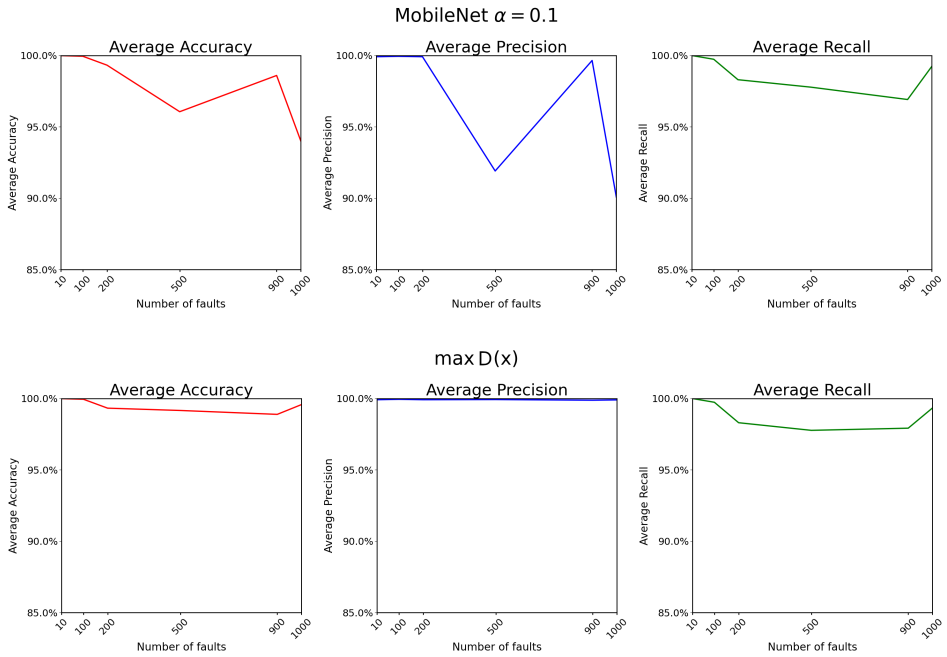

Figure 6.6: Accuracy, Precision and Recall of the best *Score-Based Fault Detectors* and of the best *Auxiliary-Based Fault Detectors* for different number of faults.

number of fault injected in a network is not proportional to the impact rate. This means that, in different situations, the same number of faults may lead to different numbers of critically corrupted predictions. Consequently, we are interested in a fault detector that works well with any number of faults, rather than one that perform well for few faults and worse when there are more faults in the network. Consequently, we have analyzed the performances of all the solution we have presented in Chapter 4. In particular, we have seen how the norm-based *Score-Based Fault Detector* has the same performances of the best *Auxiliary-Based Fault Detector*, the norm-based solution that employs MobileNet (with $\alpha = 0.1$). Furthermore, as show in *Figure 6.6*, the max-based *Score-Based Fault Detector* clearly outperforms all the other solutions, for what concerns accuracy, precision and recall. This fault detector can be considered very *effective*. Furthermore, it is inexpensive, as it simply requires the computation of a max to decide whether VGG is affected by *critical faults*. Therefore, we have proved that the max-based *Score-Based Fault Detector* is the optimal solution to our problem.

Chapter 7

Conclusion and Future Works

In this thesis, we have proposed a new way of dealing with fault tolerance in convolutional neural network. By examining different architectures, we have proposed a *fault detector* that is *inexpensive* and *effective*. In this chapter we are going to provide a summary of our *contributions* and discuss what we expect from *future works* on this subject.

7.1 Contributions

Our contributions include the development of a *fault injector* capable of dealing with large networks such as *VGG-16*. In particular, we started from a tool that only worked with *LeNet-5* and we provided a pipeline to extend it to other topologies of neural networks. We have realized a version of it for VGG-16, optimizing the execution of the program.

Secondarily, we have carried out an analysis of the resilience of VGG-16 to *multi-fault* injections. Even tough we have provided satisfactory explanations, more data is needed to properly test our hypothesis.

Finally, we have analyzed different *architectures* for a fault injector. First off, we have tested different *Auxiliary-Based Fault Detectors* that use a smaller replica of VGG-16 and compares the results of the two networks. Although these solutions seemed promising, the results show that an auxiliary model doesn't provide enough information to justify the cost of an additional network. As such, we proposed a *Score-Based Fault Injector* that compares the component of *maximum* value of the vector score with the

maximum value ever achieved by a component of VGG-16 vector score over an external dataset. As we discussed extensively in *Chapter 6*, this version of the fault detector is very cheap and offers an aggregate precision, recall and accuracy of over 98%.

7.2 Future Works

While the results of this work laid out a solid *exploratory* work on the subject of fault detection, the data gathered by our experiment is not enough to provide statistically significant proves of our hypothesis.

In future works we are going to increase the number of runs. The main constraint in our thesis was that the fault injector did not take advantage of the GPU, as it was run entirely on the CPU. As such, we will expand the fault injector, integrating it directly with *PyTorch* or *Tensorflow*, taking advantage of the ability of these two libraries to work on the GPU. This allows us to run thousands of simulations way quicker than we have done up until now.

Another possible expansion, is to *extend* this work to *other network topologies*. Even tough we expect that the solution proposed to work independently from the network, we need to provide data to back this claim. Furthermore, some architectures, such image segmentation networks, don't return a one-dimensional array but a larger matrix. In this case, we would need to test whether our solutions provide the same results.

Finally, we have seen that working only with the output of VGG-16 is enough to understand if there is a critical fault or not. However, while thresholding on the maximum value of the vector score is quite effective, there may be better options.

Bibliography

[1] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[2] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[3] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[5] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

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

[7] TS Nidhin, Anindya Bhattacharyya, RP Behera, T Jayanthi, and K Velusamy. Understanding radiation effects in sram-based field programmable gate arrays for implementing instrumentation and control systems of nuclear power plants. *Nuclear Engineering and Technology*, 49(8):1589–1599, 2017.

[8] James Reason. *Managing the risks of organizational accidents*. Routledge, 2016.

[9] Anand Dixit and Alan Wood. The impact of new technology on soft error rates. In *2011 International Reliability Physics Symposium*, pages 5B–4. IEEE, 2011.

[10] Cesar Torres-Huitzil and Bernard Girau. Fault and error tolerance in neural networks: A review. *IEEE Access*, 5:17322–17341, 2017.

[11] Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1677–1689, 2021.

[12] C. Khunasaraphan, T. Tanprasert, and C. Lursinsap. Recovering faulty self-organizing neural networks: by weight shifting technique. In *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, volume 3, pages 1513–1518 vol.3, 1994.

[13] C. Khunasaraphan, K. Vanapipat, and C. Lursinsap. Weight shifting techniques for self-recovery neural networks. *IEEE Transactions on Neural Networks*, 5(4):651–658, 1994.

[14] Andrea Diecidue. Detection of critical faults in deep neural networks via auxiliary models. 2021.

[15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[16] Balázs Csanád Csáji et al. Approximation with artificial neural networks. *Faculty of Sciences, Etvs Lornd University, Hungary*, 24(48):7, 2001.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[18] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[19] T. Tanprasert, C. Tanprasert, and C. Lursinsap. Probing technique for neural net fault detection. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 2, pages 1001–1005 vol.2, 1996.

[20] *IEEE standard for binary floating-point arithmetic*. Institute of Electrical and Electronics Engineers, New York, 1985. Note: Standard 754–1985.

[21] Alberto Bosio, Paolo Bernardi, Annachiara Ruospo, and Ernesto Sanchez. A reliability analysis of a deep neural network. In *2019 IEEE Latin American Test Symposium (LATS)*, pages 1–6, 2019.

[22] 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, 2018.

[23] Nvbitffi. `https://github.com/NVlabs/nvbitfi`.

[24] Sassifi. `https://github.com/NVlabs/sassifi`.

[25] Llfi. `https://github.com/DependableSystemsLab/LLFI`.

[26] Zitao Chen, Niranjhana Narayanan, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. Tensorfi: A flexible fault injection framework for tensorflow applications. *CoRR*, abs/2004.01743, 2020.

[27] N2d2. `https://github.com/CEA-LIST/N2D2`.

[28] Onnx. `https://github.com/onnx/onnx`.

[29] Ppm format specification. `http://netpbm.sourceforge.net/doc/ppm.html`.

[30] Distiller. `https://github.com/IntelLabs/distillerl`.