Fast and Accurate Error Simulation for CNNs against Soft Errors

Cristiana Bolchini, *Senior Member, IEEE,* Luca Cassano, *Member, IEEE,*
Antonio Miele, *Senior Member, IEEE,* Alessandro Toschi

Abstract—The great quest for adopting AI-based computation for safety-/mission-critical applications motivates the interest towards methods for assessing the robustness of the application w.r.t. not only its training/tuning but also errors due to faults, in particular soft errors, affecting the underlying hardware. Two strategies exist: architecture-level fault injection and application-level functional error simulation. We present a framework for the reliability analysis of Convolutional Neural Networks (CNNs) via an error simulation engine that exploits a set of validated error models extracted from a detailed fault injection campaign. These error models are defined based on the corruption patterns of the output of the CNN operators induced by faults and bridge the gap between fault injection and error simulation, exploiting the advantages of both approaches. We compared our methodology against SASSIFI for the accuracy of functional error simulation w.r.t. fault injection, and against TensorFI in terms of speedup for the error simulation strategy. Experimental results show that our methodology achieves about 99% accuracy of the fault effects w.r.t. SASSIFI, and a speedup ranging from 44x up to 63x w.r.t. TensorFI, that only implements a limited set of error models.

Index Terms—Soft Errors, Convolutional Neural Networks, Cross-layer Reliability Analysis, Error Modeling and Simulation, Fault Injection

1 Introduction

There is a growing interest in employing Convolutional Neural Networks (CNNs) for perception functionalities in a wide range of application domains, including safety- and mission-critical ones (e.g., automotive, robots and avionics and aerospace). As a representative example, let us consider the Advanced Driver Assistance System (ADAS) in the automotive scenario [1]; CNNs are employed to detect the lanes of the track, to identify pedestrians and obstacles, to interpret road signs and traffic lights [2], [3]. Based on such observations, subsequent planning modules in the ADAS take trajectory and control decisions. In this context, CNNs are generally executed on Graphic Processing Units (GPUs) since the Single Instruction Multiple Data (SIMD) architecture is particularly well-suited to speed up the highly data-parallel elaborations that characterize these applications, allowing to meet the strict real-time requirements imposed by the ADAS [2].

The design of digital systems in the automotive domain is regulated by the ISO 26262 standard [4], and the functionalities offered by an ADAS are classified and regulated by the Society of Automotive Engineers (SAE). Both standards require the system to expose a very high degree of reliability and to provide fault detection/management mechanisms. Although radiation-induced faults have historically been considered a concern mainly in the aerospace domain, it has been demonstrated that soft errors, such as Single Event Upsets (SEUs) [5], may interfere with the functionality of electronic systems also at the ground-level [6], with an estimated ratio of two transient faults every thousand billion hours, on average. In 2019 the number of cars traveling in Europe has been 268 millions (see [7]), leading to an estimate of a fault per car every 3.7 hours, which may be a concern.

It is thus paramount to be able to evaluate the resiliency of CNN-based applications against soft errors within their application context, to determine how to harden the system to achieve the desired/required reliability level. CNNs, and in general image processing and Machine Learning (ML) applications, may expose an intrinsic degree of fault resilience due to several reasons: i) they may deal with noisy inputs (e.g., sensors) or data quantizations, ii) their outputs may be probabilistic estimates, or iii) produced data (such as image) may be used by a human, whose perceptual limitations provide resiliency to a certain level of inexactness [8]. Furthermore, studies have investigated how the internal redundancies of ML models offer a certain degree of fault resilience [9], [10], [11], [12].

To define novel hardening techniques specific for CNNs, it is necessary to be able to accurately identify the vulnerabilities against faults of the application and of the various parts of it; that is to analyze the effects of the faults occurring in the underlying hardware platform on the behavior of the application itself. Thus, for this class of applications, hardening techniques are moving from the classical bit-wise correct/corrupted checking of the outputs towards a *usability-based* classification, to answer the question "is the downstream system able to correctly carry out its task with the produced, possibly corrupted, output?" [13].

To support this strategy, the novel contribution we propose is an accurate and fast cross-layer framework for the reliability analysis of CNN-based applications against soft errors, that studies the effects of such faults not merely on the different outputs, but also on the final

- *Authors are with the Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Italy.*
  E-mail: {first_name.last_name}@polimi.it

Manuscript received April 19, 2005; revised August 26, 2015.

functionality within the entire system.

If we exclude radiation testing, which is very expensive and does not provide enough internal controllability and observability, most relevant methods for reliability analysis fall into two main families: architecture-level Fault Injection [14], [15], [16] and application-level Fuctional Error Simulation [17], [18], [19]. *Architecture-Level Fault Injection*, FI in short, provides controllability and observability, as well as high accuracy, because of the ability of emulating the physical effects of the faults in the architecture components. The main drawback of FI techniques is the long time and effort required to integrate the FI features within the application under analysis and then to deploy them on the target architecture. Moreover, since faults may either be not activated or produce no observable error, extensive FI campaigns may be required to collect a statistically relevant amount of corrupted data. *Application-Level Fuctional Error Simulation*, FES in short, provides shorter development and deployment times, it can be applied very early during the design process, and it ensures that every experiment produces a corrupted output. The main drawback resides in the accuracy; it is crucial that the adopted *error models* are representative of the effects that faults in the hardware may induce in the data processed by the application. Indeed, as discussed in [20], FES tools frequently used in past works adopt inaccurate error models; therefore, there is the necessity to fill the gap between the fault occurrence at the hardware layer and the error manifestation at software one.

This paper proposes CLASSES (**C**ross-**L**ayer **A**nalysi**S** framework for **S**oft-**E**rrors effect**S** in CNNs), a novel cross-layer framework for an early, accurate and fast reliability analysis of CNNs accelerated onto GPUs when affected by SEUs. The proposal bridges the gap between fault injection and error simulation, by leveraging both methods in different phases exploiting their advantages. CLASSES consists of two parts. The former is a systematic approach for accurate error modeling: i) an architecture-level FI tool is exploited to perform an extensive fault injection campaign on each one of the basic operators used in CNNs, collecting all corrupted data; ii) the outcomes are then automatically analyzed to derive the corruption patterns that faults may induce in the output of each considered operators; iii) corruption patterns are then used to define a set of error models integrated into the error simulation engine, constituting the second part of the framework. In particular, for the FES part we exploited TensorFlow [21], at present one of the most popular ML design frameworks. Error simulator is used to analyze in a faster and easier way the reliability of the entire application. Moreover, according to the peculiarities of the considered class of applications, the error simulator analyses CNN outputs by means of a usability-based classification strategy, specifically defined by the designer. As a result, our proposal enables an early but accurate evaluation of the reliability of any CNN. The contributions of our proposal can be summarized as follows:

- a cross-layer framework for evaluating CNNs robustness against fault affecting the underlying hardware platform;
- a designed and implemented semi-automated error modeling tool based on architecture-level FI in GPU;
- a set of validated functional error models for CNNs executed onto GPU; and
- a designed and implemented fully-automated error simulator integrated within TensorFlow.

The solution we designed is flexible, allowing the framework to integrate a different FI tool (for instance the recent [22], [23] and new ones that will emerge), based on the adopted development environment, to produce the error models required to support the FES appropriately.

We analyze the framework performance and outcomes by comparing it against two representative FI and FES engines for CNNs accelerated onto GPU, namely SASSIFI [15] and TensorFI [17]. Due to implementation limitations for existing tools, we used both the YOLO V3 CNN [24], and two smaller size benchmarks, LeNet-5 [25] and CIFAR10 [26].

With respect to the baselines, we compared accuracy and speedup of the proposed FES solution against the FI process carried out with SASSIFI; CLASSES achieves about 99% accuracy in terms of effect of the fault on the final YOLO V3 output with about a 6x speedup. A second evaluation refers to the quality of the FES approach with respect to the facilities offered by TensorFI, by comparing the available error models and the obtained speedup in executing the simulation on the same workload with the smaller CNNs, achieving a speedup from 44x to 63x. While the speedup is a secondary aspect considering the rapid evolution and developments in FI tools, it is the ease of use of the error models, the functional classification of the outputs and the final reliability analysis the elements we focus on. We adopted GPUs as a platform, because they are the most commonly selected devices for accelerating CNNs; nonetheless we claim that the tools can be re-targeted for different devices, while the methodology remains the same.

The remainder of the paper is organized as follows. Section 2 presents background on CNNs and on GPUs, while Section 3 reviews the related work on FI and FES for CNNs. Section 4 discusses CLASSES in its details and Section 6 presents the identified error models. Section 7 then discusses the results of the application of our framework to a set of real-world CNNs, and the comparison against SASSIFI and TensorFI. Section 8 summarizes the main advantages of the methodology, and finally, Section 9 concludes the paper.

2 Background

This section provides a brief background introduction of the two key elements of the proposed application context, namely CNN and GPU.

2.1 CNN

A Convolutional Neural Network [25] is a Deep Learning model generally employed in image processing and computer vision to derive a semantic representation from the input images to accomplish a high-end task, such as item classification, object detection and image segmentation. As shown in Figure 1, a CNN is internally organized in a sequence of *layers*, each one processing multidimensional data, known as *tensors*, by means of *operators*. A tensor consists of a multi-dimensional stack of bi-dimensional matrices of values, called feature maps, generating a multidimensional grid. As an example, an image can be seen as three

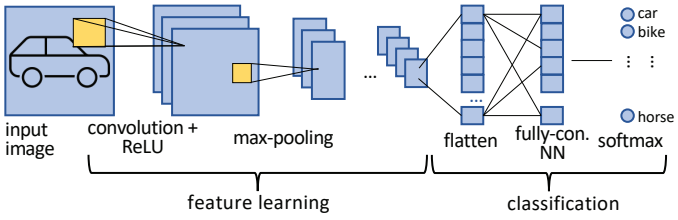

Figure 1. The typical topology of a CNN described in terms of a sequence of layers for features learning and a final classification layer, integrating a fully-connected Neural Network.

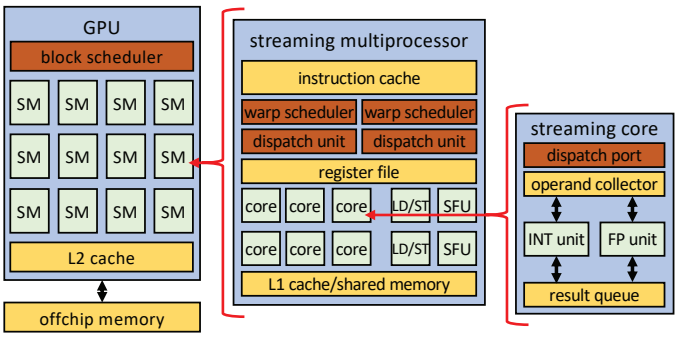

Figure 2. NVIDIA GPU architecture hierarchically organized in a grid of streaming multicores; each multicore is a SIMD unit, having centralized scheduling and dispatching units and a set of processing elements such as streaming cores, load/store units (LD/STs) and special functional units (SFUs).

stacked feature maps, each one for a color channel, thus producing a 3D tensor. There are several operators, that can be grouped into the following classes:

Convolution, used to "learn" and extract features from the input by inferring the appropriate weights;
Batch normalization, used to *fix* the data distribution, speeding up the learning process;
Activation function, a mathematical function applied element-wise to mimic the biological activation of a neuron. Common examples are the Sigmoid function, the softmax one and the Rectified Linear Unit (ReLU) one;
Max-pooling (and similar dimensionality reduction operators), used to reduce the size of the tensor for increasing the degree of generalization;
Element-wise operators for classical math operations or single-element manipulation, such as addition, multiplication, exponent and bias addition.

Since operators are general in the size of the input/output tensors, they are characterized by a set of hyper-parameters, specifying the actual size of the processed tensors. Operators belonging to these classes are usually organized in a sequence of layers devoted to the feature learning; each operator takes in input and produces in output tensors of specific sizes based on the structure of the CNN (as shown in the example in left-hand side of Figure 1). The result produced by the feature learning is a tensor as well, that is considered as final output of the CNN when the goal is for instance image segmentation or object identification. When the application goal is *classification*, the CNN contains a second part where the final tensor is first flattened and then is fed into a fully-connected Neural Network and a softmax function to produce the set of probabilities representing the likelihood of the identified object to be classified according to a set of classes.

Given the complexity of designing a CNN from scratch, a number of ML design frameworks, such as TensorFlow [21], Caffe [27], PyTorch [28] and Keras [29], has been developed. They provide general and extensible programming interfaces in high-level languages (e.g., Python and C++) to easily specify the structure of a CNN in terms of a dataflow model by instantiating operators available in the framework repository to create a graph. Moreover, these frameworks provide tools to automatically train the model to set the values for the parameters within each instantiated operator. Finally, they feature automated back-end support to target several processing devices such as CPUs or GPUs to optimize performance.

2.2 GPU

With reference to the NVIDIA architecture [30], a GPU is a many-core organized in an array of Streaming Multiprocessors (SMs) (left-hand side of Figure 2). The SM is in turn structured following the SIMD paradigm, thus with a single control unit scheduling and dispatching instructions, and a grid of simple arithmetic cores, memory load/store units and other special functional units for math operations, namely streaming cores, LD/STs and SFUs, respectively in the central part of Figure 2. Then, each single streaming core is internally implemented to read data from the register file (i.e. the operand collector), execute either integer or floating point operations and store results in the register file (result queue).

A *kernel* is a software function accelerated on the GPU by means of a large grid of threads grouped in a number of blocks. Threads compute on separate portions of data. Each threads' block is dispatched by the block scheduler to a single SM for its execution; the group is subdivided into *warps*, executed with the SIMD paradigm; indeed the SM contains multiple warp schedulers and dispatch units (2 in the figure) to execute concurrently several warps. Branch instructions, where divergences may occur among threads in the same warp, are executed by means of predicated approach; all alternative branches are executed in sequence for the entire warp and, each single thread is active only in the selected branch while it is disabled in the other ones. Finally, the dispatching unit schedules several interleaving warps to maximize the SM throughput.

The GPU has a memory hierarchy organized in a off-chip global memory and two levels of caches, a unified L2 and a per-SM L1. In addition to such a memory hierarchy, a SM contains a shared memory, that is a local scratchpad memory that can be used internally by the threads of a single block to cooperate and synchronize without requiring longer delays.

2.3 The fault model

We consider the Single Event Upset (SEU) fault model, whose effect is a bit-flip in a value stored in a register of the computing platform. Faults are rare events, therefore the classical assumption of a single fault per execution of

the application holds. The occurrence of faults may cause the application: i) to crash or the operating system to raise an exception blocking the execution, ii) to hang leading to a non-termination, or iii) to terminate by producing an erroneous result, without any alert, a.k.a. Silent Data Corruption (SDC) [31]. While the first two cases are usually managed at operating system level, SDCs represent the most critical situation, and here we target this class of effects.

When considering the GPU platform, as discussed in [15], an SEU occurring in the SM may corrupt the execution of a single thread causing an SDC, or a group of threads within the same warp belonging to a single kernel; moreover, when shared memory is used, erroneous data produced by a thread may propagate among several threads within the same block of the kernel. On the other hand, memories and caches are hardened by means of Error Correction Code (ECC); therefore, no fault in those locations will be here considered.

Within ML design frameworks, each single operator is accelerated onto GPU by means of one or multiple kernels. As a consequence, the final effect of a single SEU affecting the GPU running a CNN is the corruption of an intermediate tensor, that forwarded to the subsequent layers will potentially produce an erroneous final result. Our framework aims at provide insights on such fault effects on CNN execution.

3 Related Work

A large effort has been devoted to the reliability analysis of CNNs as surveyed in [10]. To mention the most relevant works, several papers (e.g. [19], [32]) analyzed the effects of faults corrupting the weights of the convolutional layers on the final output to measure the overall resiliency of different CNN models. [18], [33] presented approaches tracing the error propagation through the layers to analyze application-level error masking and final effects on the final output, and to identify the most critical parts of the CNN. In these works, faults corrupting both the memory storing the weights and the internal registers (causing effects on the final computation) have been considered. The conclusions are that different CNN models exhibit different vulnerabilities depending on the adopted data types and number of layers. In [34] several metrics are defined to measure the vulnerability of the layers and kernels composing a CNN model; then selective replication is applied to the most vulnerable parts to improve reliability while limiting overhead. A similar approach is proposed in [9] where an Algorithm-Based Fault Tolerance technique is used to reduce the hardening costs. In [35], selective Triple Modular Redundancy is used; an interesting aspect of this solution is the idea of considering as critical only those faults that cause the overall application to perform a misclassification. Finally, in [36] an application-aware classification is performed by considering precision and recall on the obtained object detection outputs; selective duplication is also used to harden the most critical portions of the code. In conclusion, due to the variety of possible internal structures and parameters of the CNNs and devices to accelerate such applications, the challenge is still open [35].

As far as FI facilities are concerned, radiation test offers the most adherent solution to cause soft errors, however, apart from the highly expensive equipment and the complex setup, there is little to no controllability on the injection process, and the observability is only on the primary outputs. As a result, it is suitable for final validation and black-box campaigns, not for in-depth analysis and investigations [9], [36]. To allow for controllability/observability in the campaigns, the most commonly adopted approach is emulating fault effects in the hardware platform running application, that is in the GPU executing the CNN in the present application scenario. GPU-Qin [14] exploits the CUDA-GDB debugger for NVIDIA devices to inject single bit-flips in the registers exposed by the Instruction Set Architecture (ISA); the solution is quite sophisticated and presents a 100x slowdown w.r.t. nominal execution. A similar debugging-based approach is used by CAROL-FI [22], which acts at the source code-level to inject both bit-flips and random values. The introduced performance degradation is below 5x, having both fault injection and error propagation analysis at source code-level. The limitation consists in the limited fault location sites, the used variables, introducing a significant gap between the *injected errors* and the reality of soft errors affecting the hardware platform. SASSIFI [15] and LLFI-GPU [16] adopt a different strategy, by instrumenting the source code to support error injection, before executing it on the GPU. SASSIFI, proposed by NVIDIA, can corrupt all ISA registers through several injection modes, with a reported 5x slowdown [22]. LLFI-GPU uses a similar approach, acting at the source code abstraction level, claiming a speedup of about 42x w.r.t. GPU-Qin, with similar benefits as CAROL-FI in terms of the analysis of the propagation of the errors.

All these tools rely on complex modification and re-compilation mechanisms to enable error emulation thus leading to a considerable performance degradation. Some of these tools, such as GPU-Qin [14], require a long profiling activity; moreover, as commented in [9], SASSIFI is the only tool working with the NVIDIA proprietary libraries generally employed when implementing CNNs. Indeed, library requirements of the employed FI tool, on the one hand, and of the considered application, on the other hand, may often conflict. As an example, CNNs implemented with the TensorFlow framework cannot be compiled for SASSIFI because they require different CUDA versions. Finally, the code instrumentation mechanism prevents the execution of complex applications; in our tests, a single run of YOLO V3 implemented in the Caffe framework and run in SASSIFI required more than 15 minutes, due to the large amount of data to be transmitted to the GPU. Based on these considerations faster and more accurate tools are required, as it has also been stated in [10]. The work in [23], contemporary to our work, proposes a new tool for NVIDIA GPUs called NVBitFI. The tool overcomes all limitations of previous solutions by performing a dynamic and selective code instrumentation to enable fault injection. NVBitFI does not require access to the source code, therefore improves both performance and usability. On the other hand, implementation issues still persist when working with external libraries.

Working at a higher abstraction level would be beneficial for two reasons: to dominate the complexity of CNN applications, and to accelerate and facilitate experiments set-up and execution. To this end, simulation approaches have

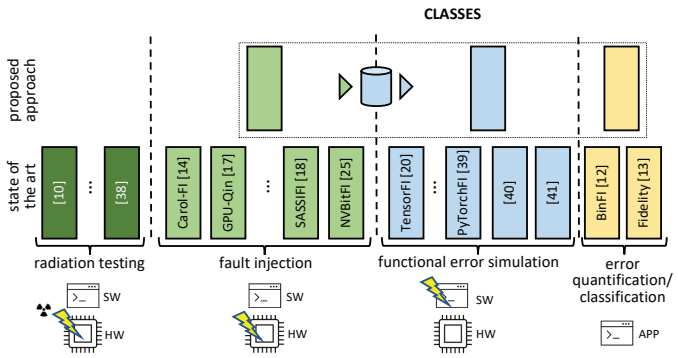

Figure 3. Comparison of the proposed approach w.r.t. the literature, highlighting the various contributions in radiation testing, architectural fault injection, functional error simulation, and error quantification/classification.

been proposed, injecting *errors* directly in the execution state of the running CNN application. The most representative example is TensorFI [17]; it is an error simulator specifically tailored for CNNs integrated in the TensorFlow framework. The tool acts at the abstraction level of the application dataflow and injects errors by means of saboteurs that manipulate the output tensor. The tool so emulates the effects of faults affecting the CNN operators execution. A similar strategy developed for other frameworks is the one presented in [33], [37], [38], integrated in the Keras, Pytorch and Caffe frameworks, respectively. [19], [32] introduce two tools developed in Keras and Darknet, supporting only the corruption of the operator weights in the CNN. Finally, the work in [39] is implemented in PyTorch and is based on a cross-layer approach executing the CNN model in software and switching down to the Hardware Description Language (HDL) description of the processing unit for the actual injection. This very last approach has not been applied to GPUs due to the high complexity of the corresponding HDL model, when available.

The considerable advantage of FES is that it can be easily integrated within the ML frameworks used in the design phase. Therefore, the CNN reliability analysis and hardening can be performed from the early phases and in the same environment of its design and training. Moreover, this high-level approach decouples the reliability analysis from the actual execution on the final device (CPU or GPU or custom hardware accelerator), thus sensibly easing the activity. On the other hand, the main criticality when working at such high abstraction level, is the need of sound and complete error models, able to reproduce all and only the effects of physical faults occurring in the underlying hardware. In [19], [32] only bit-flips in the weights are considered, neglecting the effects of faults affecting the processing unit, while in [33], [38] single or multiple bit-flips on the outputs of the CNN operators are also considered. PyTorchFI [37] allows to inject random values, single bit-flips or zero values both in the weights and in the operator outputs; indeed, the paper explicitly states the necessity of defining more advanced error models describing the effects of faults affecting the underlying hardware. Finally, TensorFI adopts more advanced multiple-value corruptions, which sometimes poses problems to correlate the corruption effect to a real physical fault able to induce such an error, as we will discuss later in the paper. Since the goal of the reliability analysis is to explore how a well designed and trained CNN behaves when an unexpected problem in the underlying hardware occurs, it is fundamental to use error models that correspond to the effects caused by faults. In particular two risks are associated with error simulation: considering errors that cannot be the result of a real fault or ignoring effects of statistically relevant faults. As a consequence, the overall analysis might lead to inaccurate conclusions and tailoring hardening solutions that do not fit real needs. Therefore, validated error models are vital to enable the adoption of error simulation.

Fidelity [11] and BinFi [12] are the most recent works similar to our proposal. The former defines suitable error models adherent to what is observed through FI and analyses the effects of the faults in terms of the Architectural Vulnerability Factor (AVF). The latter focuses on identifying the safety-critical bits in ML applications, relying on TensorFI for the injection of bit-flips in the operators. The effects of the faults on the outputs are analyzed with respect to the application, to determine the actual impact in the case of safety-critical contexts. Our framework integrates and merges both strategies, extracting different but comparable error models w.r.t. Fidelity, and performing a classification of the effects similar to the one in BinFI, yet based on a different error injection solution with the limitations of the existing FES based on TensorFI. In doing so, our proposal fills the gap between FI, FES and an application-related resiliency classification. As future work, we will investigate how Fidelity's error models can be used in CLASSES, and how CLASSES's final output can be re-formulated in terms of the metric adopted by BinFI.

Figure 3 shows a graphical summary of the reviewed literature and positions our cross-layer proposed approach. It exploits FES, integrating the facility in the most popular ML framework and adopting sound error models opportunely extracted by means of accurate low-level FI campaigns on the target hardware platform.

4 Methodological Framework

A high level representation of the proposed cross-layer reliability analysis framework, CLASSES, is depicted in Figure 4. The framework has been designed for CNNs, and more in general for Deep Learning applications, accelerated onto a target GPU-based platform. CNNs are generally exploited for perception tasks; therefore, we here consider a larger system where the CNN takes images from a source, such as a camera, and its output, being an enhanced image or a set of features, is transmitted to a downstream application using them for some decision making task.

The input of the framework is the CNN application under analysis implemented as a graph of operators op_1, op_1, ..., op_n in the adopted ML design framework, e.g., TensorFlow [21], and already properly trained, and the output is a detailed report of the performed reliability analysis highlighting the vulnerabilities and the weaknesses of the application under design. The framework has a cross-layer structure since it mixes architecture-level FI and application-level FES with to take advantage of the benefits of both.

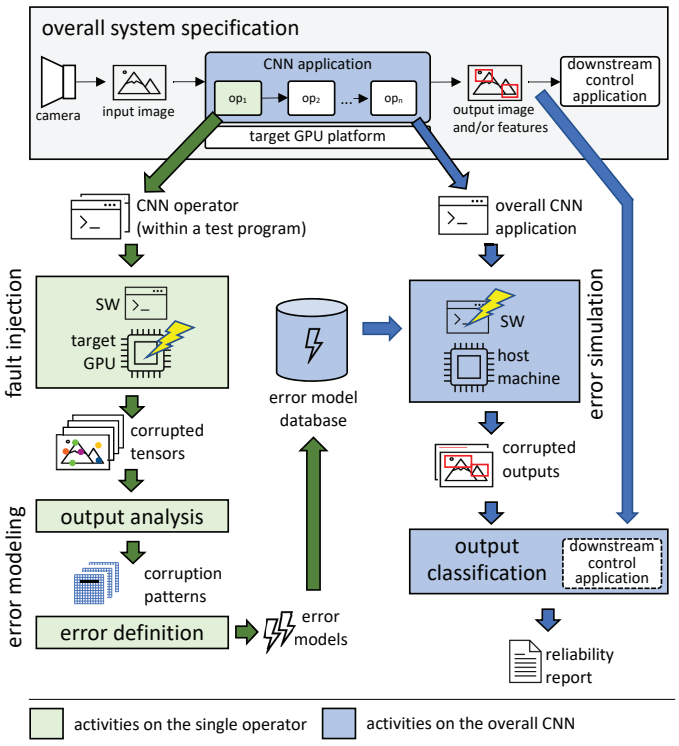

Figure 4. The proposed cross-layer reliability analysis framework: the system under analysis (top most part), the fault injection and error modeling part applied at the granularity of the single CNN operator (green left side) and the error simulation environment for the overall usability-based reliability analysis (blue right side).

In particular, FI offers high accuracy in emulating faults in the target GPU hardware while FES is flexible and fast in simulating fault effects directly in the application without requiring the specific target platform to be deployed and instrumented. To this end the framework is divided into two main parts as shown in Figure 4, discussed in details in the following subsections.

4.1 Fault Injection and Error Modeling

Fault injection is here used to analyze the output of the CNN basic operators when their execution on the target platform is corrupted by the occurrence of a fault in the underlying hardware. The final goal of this activity is to define a set of error models reproducing the effects of hardware faults on the output tensor of the executed operator.

The list of all operators used in the considered CNN is extracted and each operator is individually analyzed. The stand-alone operator is wrapped within a test program transmitting the input tensor and collecting the output one. For the sake of accuracy of the analysis, the values in the input tensor are defined according to the intermediate tensors extracted from a run of the overall CNN application. Moreover, several input tensors are considered to avoid any data-dependent bias in the obtained result. The test program is used for a large architecture-level fault injection campaign aimed at collecting a large set of corrupted output tensors.

The obtained corrupted tensors are then analyzed to derive a set of accurate high-level *functional error models* representative of all possible effects of hardware faults in the specific operator. In details, since a tensor is a multidimensional matrix of numeric values, each corrupted tensor is compared against the golden counterpart according to three different aspects:

error cardinality, i.e. the number of erroneous values in the tensor,
error value domain, i.e. the domain the erroneous values belong to, and
error spatial distribution, i.e. how erroneous values are distributed in the multidimensional matrix.

It is worth mentioning that these three aspects, and particularly the last one, are highly related to the peculiar characteristics of the GPU architecture based on the SIMD paradigm. Based on these aspects, the corrupted tensors are inspected to identify recurrent *corruption pattern*s and their occurrence frequency. The identified corruption patterns are also studied to assess if they are independent of the specific input sample and if they can be "reproduced" by an algorithm applied to the golden tensor. If all these conditions hold, the corruption pattern leads to the definition of an error model, that is implemented in the framework as a saboteur executed on the output of the corresponding operator. This activity, performed on all the operators, allows to build a database of error models to be then exploited to perform the reliability analysis of the overall CNN by means of an application-level error simulation.

CLASSES presents several advantages. As mentioned, the adoption of architecture level fault injection offers a high accuracy in the obtained results. At the same time, we are analyzing the stand-alone operators instead of the entire application simplifying the overall task. We operate on the single operators, that are the basic blocks of the CNN design, analyzing the input/output relation due to the presence of the fault. Moreover, many operators, such as the convolution, present hyper-parameters defining the size of the input and output tensors. For each of them, the hyper-parameters are tuned to execute fault injection and error modeling on the smallest version of the operator that allows for obtaining error models valid for any other version of the same operator. As we will see later, these choices considerably reduce the set up effort and the execution time of the fault injection campaign without compromising its soundness and completeness.

Finally, since CNNs are generally composed of a recurrent set of the same operators, when a new application is used, only the operators not already analyzed in previous campaigns need be taken into account, as their fault-error relation is application independent.

4.2 Error Simulation and Output Classification

The overall CNN application is analyzed w.r.t. fault effects by means of error simulation. According to the defined error models, the granularity of the corruption is at CNN dataflow graph level, by considering operators as elementary operations and corrupting the output tensors. The adopted error simulation strategy is thus based on saboteurs introduced between two nodes of the CNN dataflow graph corrupting the output tensor of the source operator. Thanks to the flexibility of the ML framework, CNN dataflow graph

structural analysis and instrumentation can be automatically performed. Similarly, probes can be inserted to trace error propagation. Finally, error simulation follows a pretty standard execution workflow: i) CNN structural analysis and instrumentation; ii) error list generation; iii) error simulation for each item in the list, collecting and classifying the produced outputs; and iv) final reliability report generation.

The error simulation approach presents higher flexibility, usability and performance than the classical fault injection counterpart. From the implementation point of view the structure of the software is less complex with less implementation and compilation issues, thus also obtaining better performance. Moreover, error simulation can be carried out on any host machine, non-necessarily featuring the target GPU; in fact, the adopted error models already represent the effects of the faults on such a platform. Thus, large workstations can be adopted to reduce the execution times of the error injection campaigns. From a methodological point of view, the approach allows for a greater controllability and observability of the fault effects also because error simulation does not suffer from fault activation issues, further reducing execution times.

Another relevant aspect of the proposed framework is related to how error simulation results are analyzed. As discussed in [13], the classical strategy based on a bit-wise comparison of the results against the golden counterpart to classify the experiment as *correct vs. error* is here not effective due to the approximate and inexact nature of CNN applications. Therefore, we adopted a *usability-based classification*; by considering the fact that the CNN is part of a larger system (refer to Figure 4). The *output classification* module is in charge of determining whether the corrupted output produced by the CNN still allows the downstream control application to perform its elaborations in an acceptable way or not. This is possible because the output classification module integrates the logic of the downstream control application itself and an application-specific policy that is actually meant to asses the usability of the produced output. As an example, we may consider a CNN performing an object detection task that supports an autonomous object grabbing robot: a slightly shifted bounding box in the produced output may be considered admissible since the robot may still be able to grab the object; therefore, in this case the corrupted output would be classified as *usable*. Conversely, a missing bounding box would cause the robot not to grab the object, and thus this second corrupted output would be classified as *unusable*. The advantage of this approach is therefore to focus on faults having a disruptive effect, identifying the main vulnerabilities of the CNN and its critical components, ignoring the ones that have a limited impact, inherently tolerated by the nature of the CNN computation.

5 Framework Implementation

CLASSES has been implemented as a semi-automated tool in Python and integrated within the TensorFlow framework, accessing the SASSIFI fault injector as an external module. The tool is semi-automated in the sense that almost all steps are automatically executed; the role of the designer is to supervise the overall workflow and critically check the correctness and the quality of the outputs of each step.

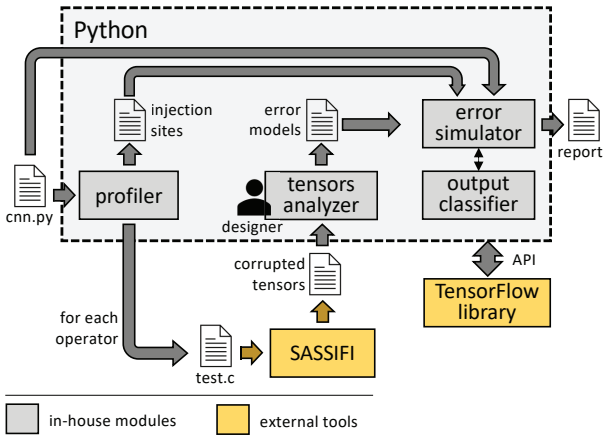

Figure 5. CLASSES internal organization.

The resulting structure is shown in Figure 5, where the *profiler* analyzes the considered CNN and identifies the fault injection sites for each operator. The *tensor analyzer* isolates the erroneous output tensors coming from the fault injection experiments and, supervised by the designer, it extracts recurrent corruption patterns within the analyzed tensors. The *error simulator* performs the functional error simulation of the whole CNN and, finally, the *output classifier* determines whether the outputs of the considered CNN after error simulation would be usable by the downstream application or not. Internals are discussed in the following subsections.

5.1 Fault Injection and Error Modeling

A first step (carried out by the *profiler* module in the software structure shown in Figure 5) is a pre-processing automatically performed by a set of Python functions using the TensorFlow Application Programming Interface (API). This step instantiates an instrumented copy of the graph used to perform a preliminary run of the CNN. This activity allows for extracting the list of the employed operators, their hyperparameters and some input/output tensors tuples from the CNN graph. The identified list of operators used in the CNN under analysis is filtered to keep only those operators for which error models are not available yet. If an operator is used in various versions, the one with smallest hyperparameter values is considered.

The fault injection activity uses SASSIFI. The choice of keeping this part separate from the rest of the framework is that FI is the only activity that has to be executed on the specific platform targeted by the application scenario. At the opposite, the main part of the framework is platform-independent, and it can be executed on any machine.

Test programs for the identified operators are implemented in C++ by using Caffe [27] since TensorFlow cannot be integrated in SASSIFI due to library conflicts. However, there is a 1:1 correspondence between SASSIFI and Caffe operators thus not leading to any methodological issue in our approach. Nonetheless, this implementation choice considerably simplifies the development and integration in SASSIFI; Caffe is a stand-alone library accelerated onto GPU. The test program of each operator is automatically generated by the tool by using a source code template.

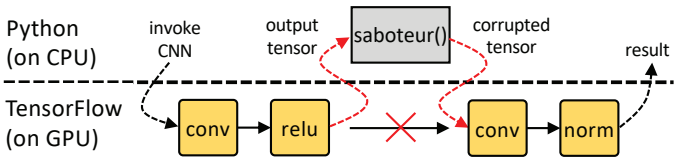

Figure 6. Error injection strategy. The sequence of operators in the CNN are accelerated onto the GPU until the operator to be corrupted is reached. The output tensor of this operator is fed into a saboteur (executed on a host CPU) that corrupts it by applying an error model. The corrupted tensor is then used as input of the remaining part of the operators (again, accelerated onto the GPU) to complete the execution of the CNN.

Basically, the test program loads the input tensor, executes the operator onto the GPU, and saves the output tensor. For each operator, the tool specializes the template with the hyper-parameters, weights and input tensor extracted by the profiler. Finally, SASSIFI is configured to inject single bit-flips (to represent SEUs) in all available sites and the execution is completely automated.

The output of the fault injection campaign executed on each operator is processed within the Python framework to perform the analysis based to the three aspects discussed in Section 4.1, thus classifying results in clusters (this activity is carried out by the *tensors analyzer* module in the structure shown in Figure 5). The most challenging aspect is the analysis of the error spatial distribution that would require the designer to manually inspect the produced output. However, based on the peculiarities of the GPU architecture and of the operators code, errors are mainly distributed in regular patterns (e.g., lines in the same feature map or lines crossing feature maps). Therefore, a manual inspection is here only required to classify corrupted tensors not handled by the automated script. It is worth noting that this manual inspection is the only step which requires the designer interaction (as shown in the figure); future work will be devoted in its automation by means of data mining techniques.

The output of this step is a tabular report describing all the identified clusters together with the corresponding occurrence frequencies. This information is saved in a JSON file constituting the error model database.

5.2 Error Simulation and Output Classification

CLASSES exploits the TensorFlow public API to load and execute the CNN model and to perform injection activities. It uses the information extracted during the pre-profiling activity to define the list of all injection sites; in particular, each output tensor is considered as a candidate.

The error injection strategy, carried out by the *error injection* module in the software structure shown in Figure 5, is implemented by virtually splitting the graph into two parts based on the injection site. As shown in Figure 6, the first part of the graph is executed to compute the tensor where the error has to be injected, the saboteur is executed and the corrupted tensor is re-introduced in the second part of the graph to compute the final CNN output. In other words, the insertion of a saboteur that modifies a tensor at the output of an operator allows to emulate the effect of the occurrence of a fault in the hardware platform when executing that operator while computing the modified tensor.

The saboteur is a Python function executed between two CNN operators, featuring the error models. As discussed in Section 6, many operators share a similar algorithmic implementation; therefore, several corruption patters are common among different operators, while their occurrence frequencies may vary. Therefore, the saboteur has been designed to implement all the identified corruption patters and the JSON file provides the fundamental information mapping patterns to operators and occurrence frequency.

The final step of the proposed error simulation environment is carried out by the *output classifier* module in the software structure shown in Figure 5. As introduced in Section 4.2, this activity consists in determining whether the produced corrupted output will be usable of not by the end-user application according to the working scenario.

The error simulator implements a standard workflow: i) error list generation according to the injection sites identified by the CNN pre-processing and designer parameters (e.g., the number of experiments); ii) experiment execution, i.e. run a number of error injection experiments, each one injecting in a randomly selected operator a randomly selected error while executing the application; iii) output classification by means of a user-defined classifier (a Python script) that assesses the usability of the output, according to the working scenario. Typically the number of injection sites is quite limited, therefore a caching strategy has been actuated to optimize the execution times of the tool; in particular, the tensor to be corrupted is cached so that for all experiments on the same injection site, the first part of the graph is not re-executed.

As a final note, the overall prototype consists of approximately 2,000 lines of codes.

6 Error Modeling Results

This section presents the results of the application of the methodological framework in the definition of the error models.

6.1 Experimental Setup

As a real-world case study for our error modeling activity we considered the YOLO V3 CNN, the state-of-the-art solution for object detection, also employed in commercial autonomous driving systems such as Apollo [40] and Autoware [41]. The considered implementation was trained upon the COCO dataset [42]. YOLO V3 internal structure counts about 6,000 operators instances belonging to 45 different operator types. The target GPU was the NVIDIA GeForce GT 750M implementing the Kepler architecture.

Table 1 summarizes the operator instances considered in the FI experiments and reports the operator name together with the size of the input and output tensors. For some operators additional parameters are reported, namely the adopted kernel size and stride for the convolutions and the negative slope value for the `Leaky ReLu`.

6.2 Results from Fault Injection Experiments

We selected the single bit-flip fault among the models supported by SASSIFI. Due to the complexity of modern GPUs and to the size of the their memory, an exhaustive FI

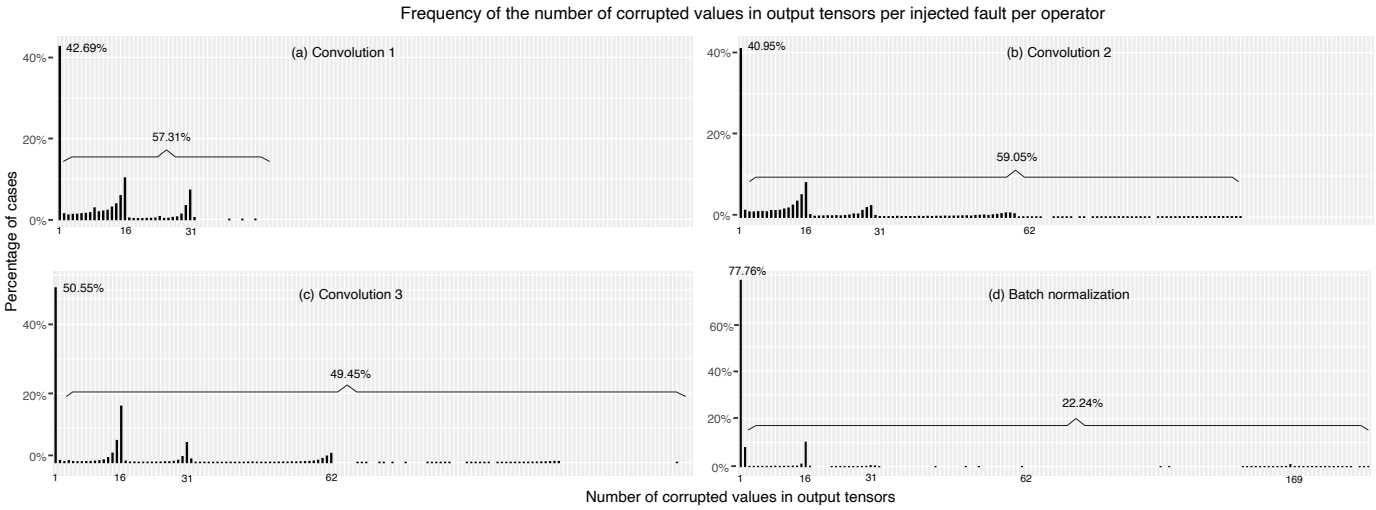

Figure 7. Frequency of the number of erroneous values per output tensor for different operators, highlighting the overall % of cases where multiple values are corrupted (e.g., 57.31% for `Convolution 1`).

Table 1
YOLO V3 operators considered for FI

Operator	Input size	Output size	Additional parameters
`Conv. 1`	$512 \times 13 \times 13$	$256 \times 13 \times 13$	K = 1, Str = 1
`Conv. 2`	$128 \times 52 \times 52$	$256 \times 52 \times 52$	K = 3, Strt = 1
`Conv. 3`	$256 \times 52 \times 52$	$512 \times 26 \times 26$	K = 3, Str = 2
`Add`	$1024 \times 13 \times 13$	$1024 \times 13 \times 13$	
`Batch Norm.`	$256 \times 13 \times 13$	$256 \times 13 \times 13$	
`Bias add`	$256 \times 13 \times 13$	$256 \times 13 \times 13$	
`Div`	1×10647	1×10647	
`Exp`	$1 \times 8112 \times 2$	$1 \times 8112 \times 2$	
`Leaky ReLU`	$256 \times 26 \times 26$	$256 \times 26 \times 26$	Slope = 0.1
`Mul`	$1 \times 8112 \times 2$	$1 \times 8112 \times 2$	
`Sigmoid`	$1 \times 2028 \times 80$	$1 \times 2028 \times 80$	

Table 2
Results from the fault injection experiments

Operator	Number of injected faults	Number of corrupted tensors
`Conv. 1`	56,000	24,273
`Conv. 2`	38,000	15,488
`Conv. 3`	66,000	31,245
`Add`	16,000	5,900
`Batch Norm.`	88,000	26,182
`Bias add`	16,000	7,400
`Div`	16,000	4,400
`Exp`	16,000	6,400
`Leaky ReLU`	16,000	5,100
`Mul`	16,000	5,700
`Sigmoid`	16,000	4,500

campaign is unfeasible. Nevertheless, it is possible to exploit the extreme regularity of the SIMD architecture to reduce the number of required experiments. Indeed, several threads simultaneously executing the same code, elaborate on different bunch of data of the same type and are run on different instances of the same hardware resources. Thus, the output corruption patterns observed when injecting faults during the execution of one of the threads will be representative of the effects of faults in all threads. The size of the fault list for each operator has been determined to inject in all memory and register bits accessed by a single thread for each instruction of the execution trace of the operator when executed on the GPU. When considering the high regularity of the SIMD architecture of a GPU, the setup allows one to obtain the same statistical results of an almost exhaustive injection campaign in the entire architecture. To make our FI experiments as general as possible, in each experiment we changed the input of the operator and we randomly selected the fault-injected thread. The outputs of the FI campaign are collected and compared against the expected output. Uncorrupted outputs are discarded while erroneous ones are further inspected to identify the recurrent error patterns. Overall we ran 360,000 FI experiments and collected 137,004 corrupted tensors; the number of injected faults and collected corrupted tensors for each operator are reported in the second and third columns of Table 2, respectively.

6.3 Error Modeling

We systematically analyzed the 137,004 collected corrupted tensors to identify the recurrent corruption patterns. In the following we report the results of such analysis based on the previously discussed aspects of interest (error cardinality, error value domain and error spatial distribution), followed by the definition of the error models.

6.3.1 Erroneous Values Cardinality

We first analyzed the number of erroneous values found in a corrupted output tensor; such a number is strongly dependent on the operator under analysis. For most considered operators the corrupted output tensor contains only one erroneous value in 85% to 92% of the experiments (depending on the operator) and two erroneous values in the remaining cases, while a higher cardinality is observed in less than 1% of the cases. However, when considering the `Convolution` (`1`, `2`, or `3`) and `Batch normalization` operators, the frequency of having multiple corrupted tensors is much higher, ranging from 22% to 57%, as shown in Figure 7.

This can be explained by considering that most operators are implemented as "linear kernels" where each thread

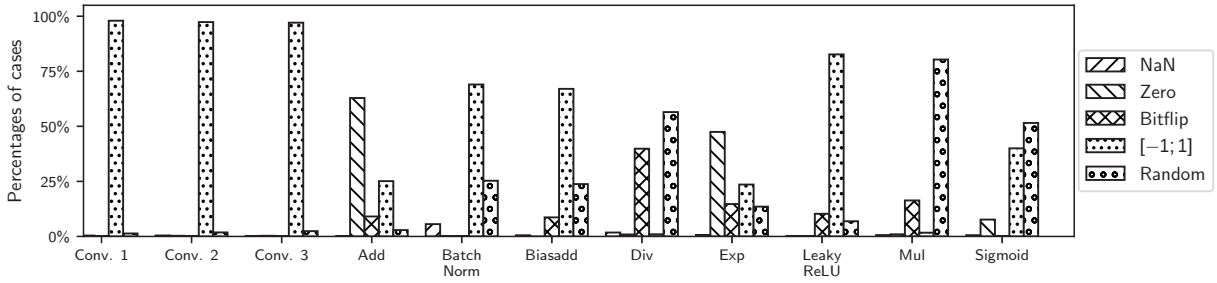

Figure 8. Distribution of erroneous values per each considered operator. For each operator, erroneous values in the corrupted output tensors have been classified as NaN, Zero, BitFlip, $[-1; 1]$, or Random.

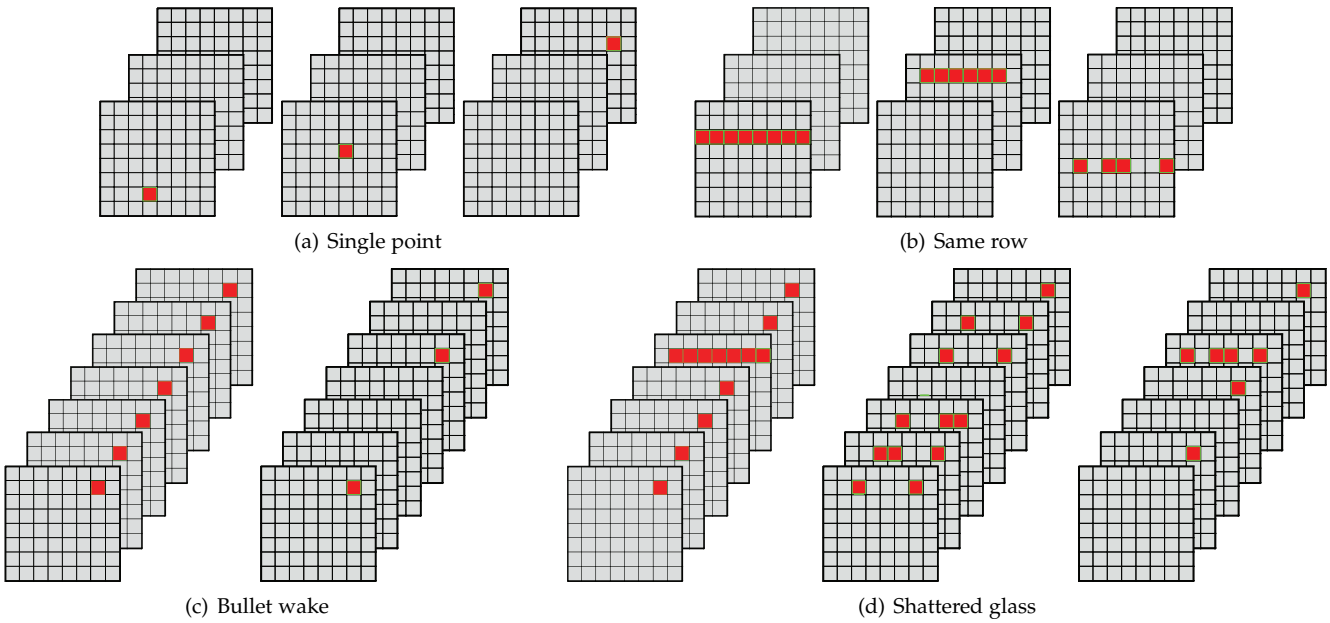

Figure 9. Spatial distribution patterns (erroneous values are colored in red). (a) *Single point*: the fault causes the corruption of a single value of a single feature map. (b) *Same row*: the fault causes the total or partial corruption of a row in a single feature map. (c) *Bullet wake*: the fault corrupts the same location in all or multiple feature maps. (d) *Shatter glass*: the fault causes the combination of the effects of *same row* and *bullet wake* patterns.

computes one value of the output tensor without exploiting GPU shared memory and with no cooperation among threads in the same block. The `Convolution` and `Batch normalization` operators are composed of several kernels based on General Matrix Multiplications (GEMM) algorithm; thus, their implementations heavily exploit GPU shared memory for threads cooperation. The corruption of one of those threads can propagate to other threads and result in many errors in the output.

6.3.2 Erroneous Values Domains

As a second analysis, we studied how each erroneous value in the corrupted tensor deviates from the golden counterpart. In particular, we defined the domains the erroneous values fall into as follows:

Not a number: the corrupted value is NaN,
Zero: the corrupted value is zero,
Bitflip: the corrupted value differs from the expected one by a single bit,
[-1,1]: the difference between the expected value and the actual one is between -1 and 1, and
Random: the value is corrupted in a completely random way, thus not falling into any of the above cases.

For most operators the great majority of faults cause an error in the *[-1,1]* class (70% up to 97% of the experiments, depending on the considered operator) with the remaining cases falling in the *Random* class. When dealing with the `Add` and `Exp` operators, we observe a great majority of cases falling in the *Zero* class. Finally, for the `Div` operator the *Random* and *Bitflip* classes are predominant, while for the `Sigmoid` operator most of the erroneous values belong to *Random* and *[-1,1]*. Figure 8 reports the observed distributions of the erroneous values for each considered operator.

6.3.3 Spatial Distribution

Finally, we analyzed the spatial distribution of the erroneous values within the corrupted output tensor. A first classification is between faults that produce one or more errors in a unique feature map and those whose effect spreads over multiple feature maps.

Single Feature Map: most of the faults that cause less than 16 erroneous values in the output tensor affect only a single feature map. This is due to the fact that the GPU kernel is

Table 3
Spatial patterns frequencies

Operator	Same Feature Map			Multiple Feature Maps		
	Single Point	Same Row	Random	Bullet Wake	Shatter Glass	Random
`Conv. 1`	42.7%	18.7%	0.0%	20.6%	16.2%	1.8%
`Conv. 2`	40.9%	13.1%	0.1%	32.9%	11.5%	1.3%
`Conv. 3`	50.5%	12.4%	0.1%	25.4%	11.5%	0.1%
`Add`	90.3%	1.8%	0.8%	0.0%	0.0%	7.1%
`Batch Norm.`	77.8%	2.5%	1.1%	12.7%	1.1%	3.0%
`Bias add`	90.1%	1.2%	1.0%	0.2%	0.0%	7.5%
`Div`	84.9%	6.7%	8.4%	0.0%	0.0%	0.0%
`Exp`	91.9%	0.5%	0.0%	0.0%	0.0%	7.5%
`ReLU`	84.5%	1.5%	1.1%	0.0%	0.0%	10.3%
`Mul`	88.4%	0.3%	0.0%	0.0%	0.0%	11.3%
`Sigmoid`	89.7%	1.4%	0.0%	0.0%	0.0%	9.0%

implemented to group threads working on the same feature map in a single block; thus such multiple errors may be due to the corruption of predicated instructions. In this subfamily of spatial distributions we identify:

Single point: a single value is corrupted (Figure 9(a)),
Same row: multiple corrupted values lie in the same row, as in Figure 9(b), and
Random: no regular pattern.

Multiple Feature Maps: most of the faults that cause more than 16 erroneous values in the output tensor spread the corrupted values among several feature maps; this is due to the fact that, as mentioned, such operators heavily exploit shared memories. In this subfamily of spatial distributions we identify:

Bullet wake: the same location is corrupted in all (or in multiple) feature maps, as in Figure 9(c),
Shattered glass: like one or more *Bullet wake* errors, but in one or multiple feature maps the corruption spreads over a row (or part of the row), as in Figure 9(d), and
Random: no regular pattern.

Table 3 reports the frequency of each spatial distribution pattern for each operator. It can be observed that, consistently with the considerations drawn in Subsection 6.3.1, spatial distributions involving multiple feature maps only appear for the `Convolution` (`1`, `2`, or `3`) and the `Batch normalization` operators (the only operators exposing a cardinality of erroneous values greater than 2). It is also worth mentioning that, on average, random errors have an incidence of 1.15% and 5.2% in single feature map and multiple features maps, respectively. To summarize, the great majority of the fault effects (93.65% on average) actually cause a well-identifiable effect, that can be modeled and reproduced through an algorithmic description.

6.4 Error Model Definition

The detailed analysis carried out on the outcomes of the extensive FI campaign is the solid basis on which we build the definition of functional error models for CNN operators. More precisely, two are the key elements we identified to generalize from a corruption pattern to an error model: i) a statistical relevance of the effects, and ii) the possibility to specify an algorithm that produces the desired pattern. Based on these premises, the only spatial pattern that cannot be systematically and accurately reproduced through an algorithm is the *Random* one, for which no error model has been defined. Since its statistical relevance is also modest (less than 6.35%), the set of defined error models can be considered sound and complete.

Table 4
Error model parameters

Error model	Parameters
Single point	x and y coords. of the point
Same row	x and y coords. of start and end point of the row
Bullet wake	x and y coords. of the enter point, first feat. map, last feat. map
Shattered glass	x and y coords. of the enter point, first feat. map, last feat. map, shattered feat. map

Furthermore, we identified a list of parameters for each model (reported in Table 4), to make it general and exploitable in an automatic simulation engine. In addition to the listed parameters, when generating a *Same row* error a number of points in the row is randomly left unaltered. Similarly for the *Bullet wake* and *Shattered glass* errors, a number of feature maps is randomly left unaltered.

The resulting set of error models well covers the effects caused by the faults and can be conveniently exploited in an error simulation campaign, for an early and accurate analysis of the reliability against SEU faults of a CNN-based application executed on a GPU.

The outcome of the adopted process is general with respect to the CNN operators and the target platform. However, the process is designed to fine tune the fault injection campaign and the error modeling activity, for an accurate outcome in relation to the case study under consideration. Therefore, when a new CNN is taken into account, it might require a re-execution of these activities, to guarantee an accurate and updated error model repository.

7 Error Simulation Results

The defined error models have been integrated in CLASSES error simulation engine. A final campaign has been prepared to analyze the ability of the proposed error simulator in managing a complete CNN. Moreover, we compared our proposal against the state-of-the-art fault injection and error simulation tools, namely SASSIFI and TensorFI, respectively.

All the experiments and comparisons have been performed on a machine equipped with an Intel Core® i7-4870HQ as CPU and a NVIDIA GeForce GT 750M as GPU, running on Ubuntu 18.04 LTS.

7.1 CLASSES vs. state-of-the-art FI

SASSIFI offers fault injection facilities, therefore the comparison between this state-of-the-art solution and our proposal considers both accuracy and performance measured in the execution time. For this experimental comparison we considered YOLO V3 CNN. We re-used the same fault list of the previous section, consisting of 360,000 random fault injections.

First of all we have to mention that we could not compare against a *pure* SASSIFI experiment where the entire YOLO CNN, implemented in Caffe, is loaded onto the FI environment due to scalability issues. Indeed, a single

YOLO run on SASSIFI took about 15 minutes, which would have led to about 10 years for running 360,000 experiments. Therefore we considered a *hybrid* SASSIFI configuration, coupled with TensorFlow; we limited the execution through SASSIFI only for the operator being corrupted by the fault, while the rest of the CNN is executed with TensorFlow at full speed. This workaround allows the extensive FI campaign to be executed using state-of-the-art tools. More in details, the 360,000 injected faults allowed to collect 137,004 corrupted faulty outputs, in about 92 hours. To perform the same 137,004 relevant error simulation experiments CLASSES took about 15 hours, corresponding to a 6x speedup with respect to the hybrid SASSIFI configuration. However, the speed up is beneficial but not paramount, considering the rapid evolution of new tools (in the future we will investigate the integration of NVBitFI), and it has been analyzed/compared to evaluate the efficiency of our solution.

The injection of the same error in the same operator for both environments allows the accuracy comparison. In particular, we considered the output of SASSIFI as ground truth (being it an accurate architecture-level fault injection tool) and we compared the effect of the fault (in SASSIFI) and of the error (in CLASSES) on the final classification produced by YOLO, to evaluate if both engines produce the same type of output or not (usable/unusable). YOLO performs an object detection task by computing bounding boxes around each identified object in the analyzed picture. In the discussed experiment, we were interested in studying whether the CNN was able to correctly detect the objects despite their actual position. For this reason, we defined an output classifier that tags the output of CNN as usable only if it contains the same list of detected objects as the golden counterpart. Collected data show an average 98.72% accuracy w.r.t. SASSIFI for the proposed CLASSES.

7.2 CLASSES vs. state-of-the-art FES

TensorFI offers FES facilities, based on a limited fault model with respect to the one we defined and use in the proposed FES activity. We did not perform any qualitative comparison between our results and the ones obtained by TensorFI because the error models implemented by TensorFI, basically bit-flips or single value corruptions, are a reduced subset of the ones we identified and implemented. The comparison against the state-of-the-art solution is therefore carried out considering only the performance of the approaches, given the previous discussion on the cardinality and adherence of the fault models.

Indeed, we were not able to execute YOLO with TensorFI, for two reasons mainly; TensorFI does not support all the operators, e.g., Leaky ReLu and Batch normalization, and the tensor data format (Batch, Height, Width, Channel – BHWC vs. Batch, Channel, Height, Width – BCHW) employed by YOLO and it would have required too long execution times, according to [17]. For this reason, we here introduce two simpler CNNs to carry out the performance comparison: LeNet-5 [25] and CIFAR10 [26] and gather a trend in the achievable speedup. LeNet-5 is used for hand-written digits classification for the MNIST dataset [43], able to achieve a 99.05% accuracy. It is a simple network composed of two convolutional layers and three dense layers. CIFAR10 performs object classification for the CIFAR10 dataset [26]. In particular, we employed the Keras implementation[1], that achieves an accuracy equal to the 78%.

Table 5
Execution times for FES (10,000 error simulation experiments)

	CNN		
	YOLO	LeNet-5	CIFAR10
TensorFI	-	18min	40min
CLASSES	25min	0.41min	0.63min
Plain execution (10,000 runs)	10min	0.27min	0.41min

For this test, we defined a campaign of 10,000 random error simulations. The execution times (in minutes) of TensorFI and CLASSES for the various campaigns are reported in Table 5. We also report the time for the nominal execution of the CNN to allow the reader to get an idea of the minimal overhead introduced by our error injection mechanism.

As it can be noticed, CLASSES is able to execute the experiment on YOLO in a reasonable time, i.e. with a 2.5x slow down. Moreover, CLASSES considerably outperforms the state-of-the-art solution on these small CNNs, with a speedup of about 44x and 63x. The motivation is that, as commented in [17], TensorFI re-defines CNN operators in Python to perform the injection and such implementation is not accelerated onto GPU as the original TensorFlow counterpart. At the opposite, CLASSES is exclusively based on TensorFlow public API, thus benefiting from the GPU acceleration of the operators. Moreover, the defined input caching strategy allows for an additional performance improvement. As a final remark, the time required for the instrumentation of the CNN to run the error simulations is about 80s, which is negligible for both approaches.

8 Advantages of the Proposed Approach

We here recap the several advantages offered by the proposed cross-layer framework with respect to the state-of-the-art FI and FES environments.

Accuracy. The exploitation of error models directly extracted from FI experiments provide an overall accuracy comparable with the one achieved by an FI tool.

Speed. The adoption of the FES paradigm not affected by fault activation issues allows for the framework to be much faster than the classical FI tools in achieving the desired amount of corrupted application outputs.

Increased productivity. The integration of the reliability analysis tool within the standard ML development framework used to design and train CNNs avoids the extra time and effort needed for porting the CNN to different tools, each one performing a portion of the overall analysis. The same unified environment is now used to analyze and design CNNs from all points of view.

Ease of use. The framework is almost fully automated, thus easing the designer's activity. Indeed, being the number of CNN operators limited, SASSIFI may be dismissed after analyzing a few CNN applications.

1. Keras: Deep Learning for humans, https://github.com/awslabs/keras-apache-mxnet, (Accessed on 12/01/2021).

Flexibility and customizability. The implementation in a scripting language gives a high flexibility, extensibility and customizability of the source code to meet the needs of each CNN being analyzed at a very low designer's effort, as, for instance, to specify the output classification function.

Portability and minimal intrusiveness. The instrumentation performed by CLASSES on the CNN graph to include the saboteurs is minimal and automated. No change has been introduced either into the TensorFlow library or in the operators used in the CNN graph. This allows the framework to be portable among different machines and to obtain a quite limited performance degradation w.r.t. nominal execution, as shown in the experimental results.

Finally, it is worth commenting that, even if the tool has been implemented in TensorFlow, it can be potentially ported to any other ML framework having a similar dataflow graph model and API (e.g. PyTorch or Keras).

9 Conclusions

We presented CLASSES, a cross-layer framework for the reliability analysis of CNNs that combines the accuracy of architecture-level fault injection with the ease of use, controllability, flexibility and speed of error simulation. We compared our methodology against the state-of-the-art SASSIFI fault injection environment and TensorFI functional error simulator, highlighting how our methodology achieves about 99% accuracy in terms of the ability of reproducing the effects of faults on the final CNN output with about 6x speedup w.r.t. SASSIFI, and a speedup ranging from 44x up to 63x w.r.t. TensorFI.

Future work is devoted to extend the framework to handle other platforms and ML algorithms, always keeping flexibility and extensibility in mind.

References

[1] M. Campbell, M. Egerstedt, J. How, and R. Murray, "Autonomous driving in urban environments: approaches, lessons and challenges," *Philosophical Trans. of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 368, no. 1928, pp. 4649–4672, 2010.

[2] R. Valiente, M. Zaman, S. Ozer, and Y. P. Fallah, "Controlling Steering Angle for Cooperative Self-driving Vehicles utilizing CNN and LSTM-based Deep Networks," in *Proc. Intelligent Vehicles Symp.*, 2019, pp. 2423–2428.

[3] Z. Ouyang, J. Niu, Y. Liu, and M. Guizani, "Deep CNN-Based Real-Time Traffic Light Detector for Self-Driving Vehicles," *IEEE Trans. Mobile Computing*, vol. 19, no. 2, pp. 300–313, 2020.

[4] International Organization for Standardization (ISO), "26262: Road vehicles - Functional safety," https://www.iso.org/standard/68383.html, 2011.

[5] T. Karnik and P. Hazucha, "Characterization of soft errors caused by single event upsets in CMOS processes," *IEEE Trans. Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, 2004.

[6] E. Normand, "Single event upset at ground level," *IEEE Trans. Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[7] ACEA - European Automobile Manufacturers' Association, "ACEA Report: Vehicles in use - Europe 2019," https://www.acea.be/publications/article/report-vehicles-in-use-europe-2019, 2019, (Accessed on 03/20/2020).

[8] S. Mittal, "A Survey of Techniques for Approximate Computing," *ACM Computing Surveys*, vol. 48, no. 4, pp. 62:1–62:33, 2016.

[9] F. dos Santos, P. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs," *IEEE Trans. Reliability*, vol. 68, no. 2, pp. 663–677, 2019.

[10] Y. Ibrahim, H. Wang, J. Liu, J. Wei, L. Chen, P. Rech, K. Adam, and G. Guo, "Soft errors in DNN accelerators: A comprehensive review," *Microelectronics Reliability*, vol. 115, p. 113969, 2020.

[11] Y. He, P. Balaprakash, and Y. Li, "FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators," in *Proc. Intl. Symp. on Microarchitecture*, 2020, pp. 270–281.

[12] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "BinFI: An Efficient Fault Injector for Safety-Critical Machine Learning Systems," in *Proc. Intl. Conf. High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–23.

[13] M. Biasielli, C. Bolchini, L. Cassano, E. Koyuncu, and A. Miele, "A Neural Network Based Fault Management Scheme for Reliable Image Processing," *IEEE Trans. Computers*, vol. 69, no. 5, pp. 764–776, 2020.

[14] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications," in *Proc. Intl. Symp. Performance Analysis of Systems and Software*, 2014, pp. 221–230.

[15] S. Hari, T. Tsai, M. Stephenson, S. Keckler, and J. Emer, "Sassifi: An architecture-level fault injection tool for GPU application resilience evaluation," in *Proc. Intl. Symp. Performance Analysis of Systems and Software*, 2017, pp. 249–258.

[16] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in GPGPU applications," in *Proc. Intl. Conf. High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 240–251.

[17] Z. Chen, N. Narayanan, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications," in *Proc. Intl. Symp. on Software Reliability Engineering*, 2020, pp. 426–435.

[18] G. Li, S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proc. Intl. Conf. High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 8:1–8:12.

[19] A. Bosio, P. Bernardi, A. Ruospo, and E. Sanchez, "A Reliability Analysis of a Deep Neural Network," in *Proc. Latin American Test Symp.*, 2019, pp. 1–6.

[20] G. Papadimitriou and D. Gizopoulos, "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers," in *Proc. Intl. Symp. Computer Architecture*, 2021, pp. 902–915.

[21] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," http://tensorflow.org/, 2015, (Accessed on 03/20/2020).

[22] D. Oliveira, P. Navaux, and P. Rech, "Increasing the Efficiency and Efficacy of Selective-Hardening for Parallel Applications," in *Proc. Intl. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, 2019, pp. 1–6.

[23] T. Tsai, S. K. S. H. M. Sullivan, O. Villa, and S. W. Keckler, "NVBitFI: Dynamic Fault Injection for GPUs," in *Proc. Intl. Conf. Dependable Systems and Networks*, 2021, pp. 284–291.

[24] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *CoRR*, vol. abs/1804.02767, pp. 1–6, 2018. [Online]. Available: http://arxiv.org/abs/1804.02767

[25] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[26] "Image classification with cifar-10 dataset," https://github.com/deep-diver/CIFAR10-img-classification-tensorflow, (Accessed on 12/01/2021).

[27] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proc. Intl. Conf. Multimedia*, 2014, p. 675–678.

[28] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8026–8037.

[29] F. Chollet *et al.*, "Keras," https://keras.io, 2015, (Accessed on 03/27/2020).

[30] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[31] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware