An Emulation Platform for Evaluating the Reliability of Deep Neural Networks

Corrado De Sio, Sarah Azimi, Luca Sterpone
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy

***Abstract*— In recent years, Deep Neural Networks have been increasingly adopted by a wide range of applications characterized by high-reliability requirements, such as aerospace and automotive. In this paper, we propose an FPGA-based platform for emulating faults in the architecture of DNNs. The approach exploits the reconfigurability of FPGAs to mimic faults affecting the hardware implementing DNNs. The platform allows the emulation of various kinds of fault models enabling the possibility to adapt to different types, devices, and architectures. In this work, a fault injection campaign has been performed on a convolutional layer of AlexNet, demonstrating the feasibility of the platform. Furthermore, the errors induced in the layer are analyzed with respect to the impact on the whole network inference classification.**

I. Introduction

In the last years, the use of Deep Neural Networks (DNNs) has grown dramatically. DNNs have been adopted in a wide range of applications within different industries, such as automotive, computer vision, and healthcare [1][2][3]. The adoption of DNNs in applications where high-reliability is demanded led to an increasing of interest in the study of their resilience and tolerance against faults. Due to their architecture, DNNs present a massive number of faulting points that can be a source of errors. The architecture of deep neural networks consists of tens of operational layers characterized by few to hundreds of millions of parameters implementing various computations such as convolution and pooling. The parameters, inputs, and outputs of each layer can all be a source of errors in the network. Additionally, the device and its specific fault models play a role that can hardly be emulated, affecting only parameters and nodes at application, topology, or algorithmic level. Nevertheless, the reliability analysis of deep neural networks usually abstracts from the real implementation on the hardware, and it is based only on the corruption of data and parameters of the layers, or topological modifications of the net (e.g., nodes removal). Even if the huge amount of memory needed by neural networks makes storage one of the main sources of error, soft errors in the storage elements are only a subset of the real faults that can affect a device. Therefore, reliability and resilience evaluation of deep neural networks should not ignore errors in data paths and hardware architecture.

In the last decades, hardware programmable devices (e.g., FPGAs and AP-SoCs) have seen wide employing for hardware acceleration, hardware emulation, and prototyping purposes. Thanks to their reconfigurability at the hardware-level, a new configuration of the available hardware resources can be achieved through the download of configuration data, also known as the bitstream, in the configuration memory of the device.

In this work, we propose FireNN, an emulation platform to evaluate the reliability of deep neural networks. The novelty of the approach is to involve programmable hardware for emulating faults affecting the hardware structure of the devices implementing the artificial neural networks. The platform enables a mixed reliability analysis of artificial neural networks combining software and hardware level through the extension of PyTorch and PyNQ frameworks [5] to support the integration of fault emulation both at the application and hardware level. The feasibility of our approach has been validated through a fault injection campaign emulating SEU effects in the configuration memory of the programmable logic of Zynq AP-SoC. Acting on the configuration memory, the modification induced on the implemented netlist faults not supported by the usual approach based on the corruption of weights, data, or topology of the network under test [6].

This paper is organized as follows: Section II provides an overview of the related works in the field of fault tolerance of deep neural networks. Section III describes the background of deep neural networks and AP-SoC. The developed evaluation platform is presented in section IV, while the experimental results of the reliability analysis on a convolutional layer of AlexNet neural network are reported in section V. Finally, conclusions and future works are drawn in section VI.

II. Related Works

Several approaches have been explored for quantifying fault tolerance of deep neural networks against soft errors. In [7], the authors analyze the effect and propagation of errors affecting the data and parameters of different layers, using different data types. In [8], a fault injection framework limited to the analysis of soft memory faults is presented. Differently, in [9], the authors characterize the impact of permanents faults using the darknet framework affecting layers for LeNet and Yolo convolutional neural networks. All the previous works perform analysis on the software layer, independently of the hardware. However, [8] reports a silicon validation, but it is limited to the faults affecting storage. On the hardware side, few works targeted the reliability of neural networks against faults affecting the hardware. In [10], the authors perform beam experiments and fault injection on different GPU architectures. In [6], a reliability analysis based on faults affecting the configuration memory of an FPGA device implementing a CNN has been executed.

III. Background

A. Deep Neural Networks

Deep neural networks (DNN) are artificial neural networks made by several computing layers that are characterized by the value of their tunable. Convolutional neural networks (CNN) are a type of DNN that proved to be very performing in several applications, especially visual classification. They are typically composed of convolutional, pooling, and fully connected layers. The increasing depth of DNN architectures brought to network with tens of millions of parameters involved [11]. Due to the necessity to manage the huge number of parameters, the complex structure of the networks, and the computational demands for the training

978-1-7281-9457-8/20/$31.00 ©2020 IEEE

phase, several frameworks have been developed. Between them, PyTorch is one of the most successful open-source frameworks [12].

B. AP-SoCs

The AP-SoCs are integrated circuits characterized by the combination of programmable hardware and a processor system. The configurable logic is characterized by programmable resources, such as LUTs, DSPs, and point-to-point interconnections which can be configured to implement a target netlist. The device configuration happens by means of a stream of configuration data downloaded in the configuration memory of the device. The reconfigurability is given by the possibility to change on demand the content of the configuration memory enabling the use of these devices for emulation and prototyping purposes. Moreover, it has been proved by several research works how fine-grained configuration memory manipulation is able to induce desired modifications in the netlist implemented on the device [13][14][15].

IV. The Developed Evaluation Platform

The proposed platform aims to exploit the reconfigurability feature of Zynq AP-SoC [16] to provide a first-order analysis of the faults affecting the hardware implementing neural networks through modification of the configuration memory content to emulate specific hardware faults (e.g. stuck-at faults, conflicting connections, delays, and others), also belonging to different hardware devices.

A. Background on FireNN Architecture

The developed platform is characterized by two environments, named *FireNN Machine* and *FireNN Platform*, typically running on the processor system of the Zynq AP-SoC and a host computer, respectively. Figure 1 shows an overall view of the architecture and modules involved. However, the *Machine* offers an interface to the *Platform* agnostic to the hardware device (i.e., usually the AP-SoC) on which it is running. Differently, the backend of the *Machine*, elaborated in the following subsections, is based on hardware-dependent elements named *Gears*. For this reason, the *FireNN Machine* is not constrained to the Zynq AP-SoC, and it can also run on other devices supporting Python and PyTorch. For the sake of clarity, the device on which the *Machine* runs will be identified with the Zynq AP-SoC, in the rest of the paper. However, we would like to emphasize that any device for which *Gears* are provided can be easily used.

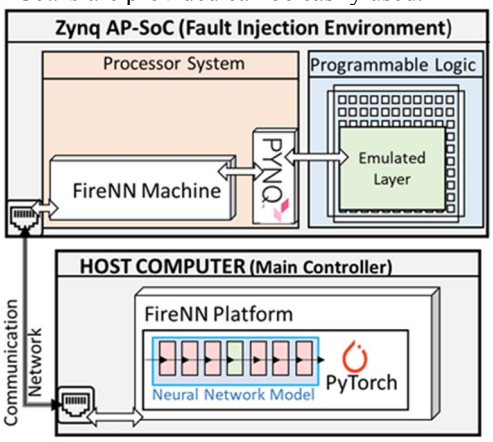

Fig. 1. Schema of architecture and modules.

The two FireNN environments are coded in Python, based on the PyTorch framework. *FireNN Platform* running on the host computer has features to wrap the PyTorch modules selected for the reliability evaluation. Additionally, it manages the communication with the *FireNN Machine* running on the processor system. The *FireNN Platform* provides APIs to virtually move the module to the device where the *Machine* is running similar to the APIs dedicated to moving the computation between GPUs and CPUs offered by PyTorch.

The main purpose of the *FireNN Platform* running on the host computer is to offload the computations demand of the whole network from the Zynq in favor of a higher-performance system. Indeed, Zynq devices can present some performance issues when running software neural network applications, especially related to the required amount of on-board RAM. Nevertheless, the *Machine* and the *Platform* can also run on the same emulation device if the performance is high enough.

B. FireNN Platform and Shells

The *FireNN Platform* provides the APIs for enabling relocation of a neural network submodule (e.g., a convolutional layer) to be transparently and remotely executed on another device with the aid of the *FireNN Machine*. The relocation APIs mimic the APIs to move the computation between GPUs and CPUs provided by PyTorch. In detail, the relocation is achieved through the encapsulation of neural network basic blocks in a container deriving from *module* class of PyTorch, named *Shell*. The *Shell,* which is encapsulating the original module, is created by the *to_device* relocation API. Additionally, the *to_device* API scans the structure of the neural network model to insert the *Shell* in the topology of the neural network in the place of the relocated module, as shown in Figure 2. The *Shell* is connected to the *Machine* running on the Zynq, where a *Gear* related to the *Shell* will be instantiated. In FireNN, a *Gear* is the hardware-dependent computational element emulating a layer of the PyTorch network. Their characteristics are explained in the next subsection. In addition to offering all the functions inherited from the encapsulated module, the *Shell* provides the frontend to the user to perform reliability experiments on the Zynq side. In particular, the *Shell* has methods to easily execute inference using the original data path, passing through the original operative layer, or the new one deployed on the Zynq device. It offers APIs to perform injections of different kinds of fault supported by the device.

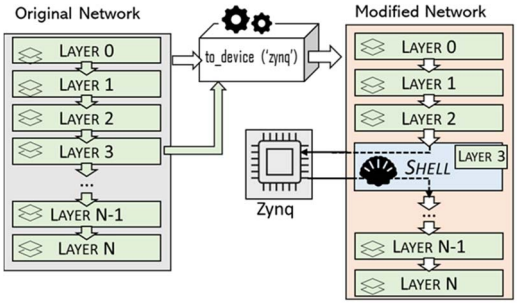

Fig. 2. Relocation of the layer under test to the Zynq through the proposed *Shell* encapsulation mechanism.

Moreover, the *Shell* is equipped with a tuning routine that can be used to infer the dimension of input and output data. This was necessary because some PyTorch modules are agnostic of I/O data dimensions, even if, within the neural network, their I/O data dimensions are defined univocally by the modules (e.g., fully connected layer) for which I/O data dimension need to be defined.

978-1-7281-9457-8/20/$31.00 ©2020 IEEE

C. FireNN Machine and Gears

The *Machine* is the module providing the mechanism to manage the computing layers to be deployed on the hardware. Layers are deployed on the AP-SoC with the purpose of manipulating the device-based implementation or execution of that module to perform reliability analyses.

The *Gears* are hardware-dependent implementations of a layer, provided with a python driver and descriptor. The *Gears* are the only hardware-dependent elements of the proposed evaluation platform. They offer a common interface to be easily managed by the *FireNN Machine,* independently of their backend. Differently, the backend is strongly dependent on the type of neural network layer that the user wants to test and the hardware on which it is implemented. Hence, the *Gears* consist of three parts: the interface; the implementation of the operational layer for the specific device; and the driver to allow the use of the hardware part through the interface. A conceptual schema summarizing the element composing the *Gear*s is reported in Figure 3.

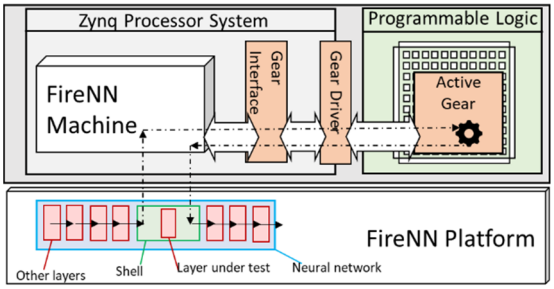

Fig. 3. Conceptual schema of the structure of *Gears*

The main operations supported by *Gears* are the *forward*, *deploy,* and the *injection* operations. The *forward* operation has the same role as the forward method in PyTorch. Given the input tensor, it performs the operation associated with that module and returns the output tensor. This requires that the characterizing parameters of the neural network layer (e.g. weights, bias, hyperparameters, etc.) are transmitted by the *Shell* during the relocation process. The *deploy* operation executes the step to deploy the *Gears* backend on the Zynq and performs the necessary setup steps for enabling the *forward* operation. Finally, through the *injection* operation, it is possible to insert faults in the layer accordingly with the fault models chosen by the user. The injection mechanism is able to select bits in the configuration memory related to specific resources in order to affect specific elements of the network architecture (e.g., connections, DSPs, etc.). The *Gears* injection mechanism for Zynq relies on the PyXEL framework [16]. Additionally, the *Gears* are provided with a tunable timeout mechanism to avoid fault-induced endless waits.

The *Machine* offers a frontend agnostic of the device on which it is running. Its main role is to handle the communication with the *Platform* module and manage and orchestrate the *Gears* instances and their usage. In detail, the *Machine* receives and processes commands from the *Shells* to instantiate and perform operations on the *Gears* associated with them. The creation of the *Gears* instances is managed on-demand by the *Machine*. Basing on the commands received by the *Shells,* the *Machine* instantiates, deploys, runs, and injects the *Gears* associated with the *Shell*.

V. Experimental Analysis and Results

To confirm the feasibility of the proposed approach, we carried out a reliability analysis of a convolutional layer of a modified version of the AlexNet neural network model [18].

A. Experimental Analysis

The architecture of the network used in the experimental analysis is presented in Figure 4. It is a modified pre-trained version, provided by *torchvision*, of the AlexNet network made of 13 layers for feature extraction and 2 fully connected layers for classification. The input is a tensor of dimensions 3×224×224 representing a cropped and normalized RGB picture. The network is pre-trained on the ImageNet dataset and is able to classify 1000 different categories [18]. The layer selected for the reliability analysis is the fifth (i.e. the last) convolutional layer of the network.

The layer is characterized by 590,080 parameters (i.e., 256 kernels with dimensions each of 256×3×3 and 256 bias). Using Vivado HLS, a custom IP Core performing the same operation of the convolutional layer under test has been developed. In detail, it computes 2-D multichannel convolution between input with dimension 13×13 and a 3×3 kernel with 256 input channels and 256 output channels. Data are represented using 32-bits floating-point representations, accordingly with the PyTorch model of the network. Data transfers between the processor system and programmable logic are supported by direct memory access to transfer streams of data from the DDR memory to the programmable logic and vice versa.

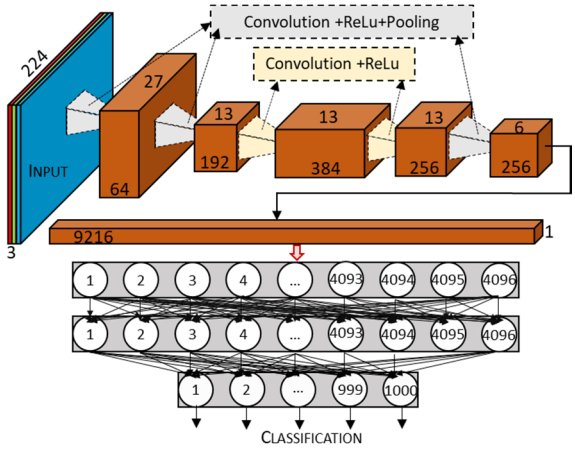

Fig. 4. DNN architecture adopted in the experimental analysis.

Typically, the results of the neural networks inference phase are reported as a numerical value for each category. Using a softmax function, these values are normalized and reduced to a probabilistic distribution over the labels. Hence, the output of the neural network is characterized by a ranking of labels with an associated percentage of confidence.

B. Experimental Analysis and Results

A set of images from the ImageNet dataset has been used as the test vector. As a first validation experiment, we compared the results obtained from the PyTorch and FireNN models of the network. The two networks led to the same classification results, both in terms of category and confidence. These outputs have been taken as golden results in order to detect deviations from nominal operations of the neural network when the convolutional layer is executed by the faulty netlist implemented on the Zynq. The random bitflip injections emulate the fault model typically observed in

978-1-7281-9457-8/20/$31.00 ©2020 IEEE

programmable devices affected by SEU. However, through an accurate bitstream manipulation, the platform also enables the injection of other fault models suited to the platform under emulation (e.g., stuck-at faults for ASICs). Please note that, due to the intrinsic characteristics of programmable devices, not all configuration data bits will generate errors when corrupted. To elaborate more, since only a subset of resources is used by design implemented on the programmable logic, random injections could target unused resources. These injections cause errors only if the modification to the unused resource will make it conflict with an active one. Moreover, we would like to emphasize that the injection of SEUs in the configuration memory of the programmable logic does not involve only faults related to the parameters and data of the network. Changing in the content of the configuration memory leads to undesired modification also in the structure of the hardware implementation of the layer, affecting data path, computational elements (e.g., DSP), interconnections, etc.

We performed 10,000 experiments, injecting a single bitflip in the configuration data of the convolutional IP core. The injected bits have been chosen randomly among the whole configuration memory bits. All the faults are injected before inference (i.e. not while the layer is running). A corrupted configuration bitstream is generated for each bitflip and tested singularly for multiple inferences on the test vector. Any deviation from the golden result has been considered as an error. The time required for each experiment, from the generation of the location to inject until the classification output is about 5 seconds. Errors have been classified into four groups. The proposed categories of errors are *misclassification*, *degradation, timeout*, and *representation*. The *misclassification* error category means that the category with the higher confidence obtained as output by the injected neural network does not match with the one reported by the golden output. The *degradation* error is a less significant error than misclassification. The *degradation* category gathers the outputs that differ from the golden one with respect to the confidence value but without changing the classification category. The injected faults caused an error that propagated until the output, affecting the obtained confidence values. Differently from previous categories, *timeout* and *representation* errors categories prevent the neural network from generating a classification output. *Timeout*s occur when injected faults prevent the on-Zynq layer from returning a result. In this case, it is not possible to forward the output of the layer to the following one and obtain a classification output. Finally, a *representation* error is detected when returning data from the *Gear* cannot be interpreted as a number.

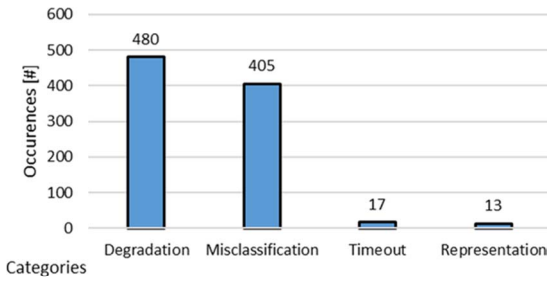

Fig. 5. Classification of the occurred errors. Total number of error is 915.

As a result of the reliability evaluation, 915 injections out of 10,000 caused errors at the neural network outputs. *Degradation* has been the category with more observed occurrences, followed by *misclassification* with 52.45% and 44.26% of the total detected errors, respectively. Only a few cases of *timeout* (1.85%) and *representation* (1.42%) errors occurred. Figure 7 reports a comparative graph of the detected errors.

VI. Conclusions and Future Works

In this paper, we presented an evaluation platform to analyze errors in a neural network not only at the application level but also at the hardware implementation level, fully integrated with the PyTorch library. Additionally, the platform enables the manipulation of the configuration data of AP-SoC to inject specific fault models. As future work, we plan to provide a ready set of Gears and fault models targeting most common faults, devices, and architecture to permit more wide analysis exploiting the proposed platform.

References

[1] A. Karpath *et al.*, "Large-Scale Video Classification with Convolutional Neural Networks," *2014 IEEE Conference on Computer Vision and Pattern Recognition*, Columbus, OH, 2014, pp. 1725-1732.

[2] F. Falcini, G. Lami and A. M. Costanza, "Deep Learning in Automotive Software," in *IEEE Software*, vol. 34, no. 3, pp. 56-63, May-Jun. 2017

[3] Y. LeCun, Y. Bengio and G. Hinton, "Deep Learning" in Nature, vol. 521, pp. 436, May 2015.

[4] W. G. Hatcher and W. Yu, "A Survey of Deep Learning: Platforms, Applications and Emerging Research Trends," in IEEE Access, vol. 6, pp. 24411-24432, 2018.

[5] "pynq.io", 2020, [online] Available: www.pynq.io.

[6] B. Du, *et al.*, "On the Reliability of Convolutional Neural Network Implementation on SRAM-based FPGA" *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, Noordwijk, Netherlands, 2019.

[7] G. Li *et al*, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications", *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*, New York, NY, USA, 2017, Article 8, 1–12

[8] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), San Francisco, CA, 2018, pp. 1-6.

[9] A. Bosio, P. Bernardi, A. Ruospo and E. Sanchez, "A Reliability Analysis of a Deep Neural Network," *2019 IEEE Latin American Test Symposium (LATS)*, Santiago, Chile, 2019, pp. 1-6.

[10] F. F. d. Santos *et al.*, "Analyzing and Increasing the Reliability of Convolutional Neural Networks on GPUs," in *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663-677, June 2019.

[11] Khan, Asifullah & Sohail, Anabia & Zahoora, Umme & Saeed, Aqsa. (2019). A Survey of the Recent Architectures of Deep Convolutional Neural Networks. Artificial Intelligence Review.

[12] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In *Advances in Neural Information Processing Systems 32*, Curran Associates Inc, pp. 8024–8035, 2019.

[13] A. Megacz, "A Library and Platform for FPGA Bitstream Manipulation" *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, 2007, pp. 45-54.

[14] S. A. Guccione and D. Levi, "JBits: A Java-Based Interface to FPGA", *Proc. 2nd Annual Military and Aerospace App. of Programmable Devices and Technology Conference*, 1999.

[15] K. Dang Pham, E. Horta and D. Koch, "BITMAN: A tool and API for FPGA bitstream manipulations," *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, Lausanne, 2017, pp. 894-897.

[16] Xilinx, Inc., "Zynq-7000 All Programmable SoC: Technical reference manual", San Jose, CA, USA, July 2018, User Guide, UG585.

[17] L. Bozzoli *et al.* "PyXEL: An Integrated Environment for the Analysis of Fault Effects in SRAM-Based FPGA Routing," *2018 International Symposium on Rapid System Prototyping (RSP),* Torino, Italy, 2018.

[18] Alex Krizhevsky, *et al*. ImageNet classification with deep convolutional neural networks. Commun. ACM 60, June 2017, 84–90.

978-1-7281-9457-8/20/$31.00 ©2020 IEEE