HAL
open science

Investigating data representation for efficient and reliable Convolutional Neural Networks

Annachiara Ruospo, Ernesto Sanchez, Marcello Traiola, Ian O'Connor, Alberto Bosio

To cite this version:

Annachiara Ruospo, Ernesto Sanchez, Marcello Traiola, Ian O'Connor, Alberto Bosio. Investigating data representation for efficient and reliable Convolutional Neural Networks. Microprocessors and Microsystems: Embedded Hardware Design , 2021, pp.104318. 10.1016/j.micpro.2021.104318 . hal-03320356

HAL Id: hal-03320356

https://hal.science/hal-03320356v1

Submitted on 4 Feb 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Investigating Data Representation for Efficient and Reliable Convolutional Neural Networks

Annachiara Ruospo, Ernesto Sanchez

DAUIN - Politecnico di Torino, Italy

Marcello Traiola, Ian O'Connor, Alberto Bosio

Univ Lyon, ECL, INSA Lyon, CNRS, UCBL, CPE Lyon, INL, UMR5270, France

Abstract

Nowadays, Convolutional Neural Networks (CNNs) are widely used as prediction models in different fields, with intensive use in real-time safety-critical systems. Recent studies have demonstrated that hardware faults induced by an external perturbation or aging effects, may significantly impact the CNN inference, leading to prediction failures. Therefore, ensuring the reliability of CNN platforms is crucial, especially when deployed in critical applications. A lot of effort has been made to reduce the memory and energy footprint of CNNs, paving the way to the adoption of approximate computing techniques such as quantization, reduced precision, weight sharing, and pruning. Unfortunately, approximate computing reduces the intrinsic redundancy of CNNs making them more efficient but less resilient to hardware faults. The goal of this work is twofold. First, we assess the reliability of a CNN when reduced bit widths and two different data types (floating- and fixed-point) are used to represent the network parameters (i.e., synaptic weights). Second, we intend to investigate the best compromise between data type, bit-widths reduction, and reliability. The characterization is performed through a fault injection environment built on the *darknet* open-source framework and targets two CNNs: LeNet-5 and

Email addresses: `{firstname.lastnale}@polito.it` (Annachiara Ruospo, Ernesto Sanchez), `{firstname.lastnale}@ec-lyon.fr` (Marcello Traiola, Ian O'Connor, Alberto Bosio)

YOLO. Experimental results show that fixed-point data provide the best trade-off between memory footprint reduction and CNN resilience. In particular, for LeNet-5, we achieved a 4X memory footprint reduction at the cost of a slightly reduced reliability (0.45% of critical faults) without retraining the CNN.

Keywords: Approximate Computing, Convolutional Neural Networks, Reliability, Fault Injections

2010 MSC: 00-01, 99-00

Journal of Microprocessors and Microsystems

DOI: 10.1016/j.micpro.2021.104318

© 2021. This manuscript version is made available under the CC-BY-NC-ND 4.0 license https://creativecommons.org/licenses/by-nc-nd/4.0/

---

1. Introduction

In the last decades, Convolutional Neural Networks (CNNs) have gained popularity due to their excellent performance in solving complex learning problems [1]. Indeed, they provide very good results for many tasks such as object recognition in image/videos, drug discovery, natural language processing up to playing games [2, 3, 4]. They are categorized as a class of Deep Neural Networks (DNNs) [5] and their name origins from the mathematical linear operation between matrixes called convolution.

Being brain-inspired models, convolutional neural networks have an inherent resilience attributed to their distributed and parallel structure and the space redundancy introduced because of over-provisioning [6]. Several studies have investigated the fault tolerance of DNNs, for example [7], and have demonstrated that NNs are highly robust against computation errors [8, 9]. As a consequence, hardware-specific NNs, ranging from circuits for embedded machine learning applications to custom Very Large Scale Integration (VLSI) of neural networks in silicon, are traditionally considered tightly robust to hardware faults (HW faults).

However, several recent studies have demonstrated that HW faults induced

by an external perturbation (i.e., in a harsh environment) or due to silicon wearout and aging effects can significantly impact the inference leading to CNN prediction failures [10, 11, 12]. Therefore, ensuring the reliability of CNNs is crucial, especially when they are deployed in safety-critical and mission-critical applications, such as robotics, aeronautics, smart healthcare, and autonomous driving [13]. Reliability can be defined as the probability that a HW fault causes a failure [14]. The reliability analysis and its evaluation are regulated by standards depending on the application domain (e.g., IEC 61508 for industrial systems, DO-254 for avionics, ISO 26262 for automotive), and it is usually assessed through fault injection (FI) campaigns.

Nowadays, to address data confidentiality issues and bandwidth limitations, the trend is to push deep learning based systems from the cloud to edge devices [15, 16, 17], such as Internet-of-Things (IoTs) devices, given the ever-increasing internet-connected IoTs. One of the principal advantage is that it alleviates the communication latency which is unacceptable for real-time safety-critical decisions, e.g., in autonomous driving. For this reason, the design of custom Application Specific Integrated Circuit (ASIC) hardware accelerators to support the energy-hungry data movement, speed of computation, and memory resources that CNNs require to realize their full potential at the edge is crucial [18].

In parallel with reliability assessments [19], a significant effort has been done to reduce the memory and energy footprint of CNNs leveraging on reduced bit-width data type in either training or inference phase. Indeed, one important limitation about the adoption of the newer version of CNNs is the memory required for storing the network parameters (e.g., synaptic weights). For example, VGG-Net [20] requires 500 MB of memory, a complexity that simply cannot fit the constrained hardware of many embedded systems. Recently, Approximate Computing (AxC) has become a major field of research to improve both speed and energy consumption in embedded and high-performance systems [21]. By relaxing the need for fully precise or completely deterministic operations, AxC substantially improves energy efficiency and reduces the memory requirement. Various techniques for AxC augment the design space by providing another set

of design knobs for performance-accuracy trade-offs. As an example, the gain in energy between a low-precision 8-bit operation suitable for vision and a 64-bit double-precision floating-point operation necessary for high-precision scientific computation can reach up to 50× by considering storage, transport, and computation. The gain in energy efficiency (the number of computations per Joule) is even larger since the delay of basic operations is greatly reduced. Having simpler operators also reduces the implementation cost, which allows the network to use more resources in parallel.

Therefore, the major challenge is to find an adequate data representation for CNNs that fits well with the application and hardware constraints. CNNs lend themselves well to AxC techniques, especially with fixed-point arithmetic or low-precision floating-point implementations, which expose large fine-grain parallelism. For instance, in [22] the authors describe a *binary network* which exploits only two values $\{-1, 1\}$ for the weights representation. Another proposed solution is the *ternary network* [23]; it quantizes weights into 3 different values $\{-1, 0, 1\}$. Finally, XNOR-Net[24] uses a slightly different methodology: all computations are performed through XNOR and bit counting operations, at the same time reducing the precision of the operands involved during the computation. Consequently, it is necessary to understand if those optimized models are reliable enough to tolerate failures that propagate throughout the system. It starts to be crucial to evaluate CNNs behaviour in a faulty scenario to determine if they can still be safely deployed in a safety-critical system. These doubts are justified if considering the growing technology scaling in chip manufacturing. Due to the transistors shrinking, newer hardware platforms are more complex and, at the same time, more susceptible to faults, albeit faster.

The end-goal of this paper is to characterize the impact of permanent faults affecting a CNN by means of fault injection campaigns, when custom data types are used for representing the network parameters. We analysed different implementations of the same CNN architecture, when different data types are exploited, and we identified the best representation leading to achieve a 4x reduction of memory footprint with the highest resilience to faults. Two case

studies are presented: the former targets LeNet-5, a popular CNN, the latter focuses on a deeper CNN (YOLO) used for the task of detecting objects in real time.

The rest of the paper is structured as follows: after summarizing the related works in the field, our main contribution is highlighted (Section 1.1). Section 2 presents the methodology, focusing on the custom data type description and the data conversion technique. In the same section, both the fault injection scenario and environment are described. Next, in Section 3, we present the two case studies (i.e., LeNet-5 and YOLO), along with the experimental results coming from the FI campaigns. Finally, Section 4 concludes the article and outlines some of the possible future research directions.

1.1. ***Related Works***

In the literature, increasing attention is paid to the Neural Network Reliability. Depending on multiple factors, such as FI typology, level of abstraction, and fault models, it is possible to identify different sets of interesting research activities.

A significant set focuses on analysing a specific fault model: the *soft* error (i.e., bit flip). In [25, 26] authors performed a deep analysis of CNN reliability when 32-bit floating-point values are used as data type for weights representation, it turned out that bit 30th is the most critical among the overall 32. Interestingly, we found out the same results with our methodology as shown in next sections. In [27, 28], the authors evaluate the reliability of one CNN executed on three different GPU architectures (Kepler, Maxwell, and Pascal). The soft errors injection has been done by exposing the GPUs running the CNN under controlled neutron beams. A similar but wider approach is detailed in [29], where the authors assess the reliability of a 54-layers DNN (NVIDIA DriveWorks) through fault injection experiments and accelerated neutron beam testing for permanent and transient faults, respectively. Faults are injected on the DNN's weights and on the input images. All inferences are executed on Volta GPU only targeting 32-bit floating-point values.

Moving forward, Li *et al.* present in [30] a different analysis. They characterize the propagation of soft errors from the hardware to the application software of different CNNs. The injections are performed by using a DNN simulator based on open-source simulator framework, Tiny-CNN [31]. Thanks to the flexibility of the simulator, it is possible to characterize each layer for a more precise analysis. In this article, CNNs with six different data types are considered: 64-bit double precision floating-point, 32-bit single precision floating-point, 16-bit half precision floating-point, 32-bit fixed-point (with two different radix points), and 16-bit fixed-point. Overall, they conclude that, the larger the dynamic value range of the network's data type, the higher the likelihood of having large deviations in values in the event of faults leading to wrong predictions.

Furthermore, a different framework is shown in [32]: Ares, a light-weight DNN fault injection framework. The authors present an empirical study on the resilience of three prominent types of DNNs (fully connected, CNNs and Gated Recurrent Unit). In particular, they focus on two fixed-point data types for each network: $Q_{3,13}$ i.e, 3 integer and 13 fractional bits, and $Q_{2,6}$. Their experiments demonstrate that the optimized $Q_{2,6}$ data type is 10x more fault tolerant. The reason lies in the fact that the unnecessary larger range of integer values increases the chance of failure happening. It is worth noting that this result is in line with our gathered results, presented in Section 4.2. It is a common trend to explore fixed-point computations for ultra-low power embedded systems with a limited power budget [33]. Finally, in [34], the authors analyse the reliability of a DNN accelerator by following a High Level Synthesis (HLS) approach. They characterize the effects of both permanent and transient faults by exploiting a fault injector framework embedded into the RTL design of the accelerator. Faults are injected during the inference cycle only on a subset of registers: those that are in charge to store weights, input values, and intermediate ones used throughout the inference job, without considering the effects of faults in the other data-path units. As for the used data representation, they perform the experiments by only adopting a 16-bits fixed-point low precision model, claiming a negligible accuracy loss with respect to a full-precision data

model.

Additionally, it is worth also mentioning the research contribution of [35] and [36], where the authors investigate the reliability of CNNs exploiting both the 16-bit and 8-bit integer data representation. Then, moving towards an even smaller data dimension, related works in [37] and [38] exploit reduced bit-widths. The former uses 5-bit and 3-bit fixed-point data types and a binary representation. The latter performs reliability assessment analysis on a binary neural network, where only 1 bit is used to represent the parameters (weights and biases).

In this light, the principal contribution of this paper is a comprehensive analysis on the behavior of CNNs depending on their data representation. In a previous work [11, 39], we evaluated the impact of permanent faults affecting CNNs through software FI campaigns. Compared to the state-of-the-art analyses [32, 34, 11, 39], a wider spectrum of floating- and fixed-point representations is given (five typologies ranging from 32 bits up to 8 bits). As a result, we identified the best data representation leading to the highest memory footprint reduction and the highest resilience to faults.

2. Methodology

This section presents the adopted methodology. We exploit the *darknet* open source DNN framework [40]. Implemented in C and CUDA language, it is suitable to perform end-to-end deployment of neural network architectures in a very simple way. It further supplies a very simple environment where several configurations of DNNs, including CNNs, can be executed either to perform training or inference jobs. In our work, we modified *darknet* framework to (i) approximate the DNN and (ii) inject faults at the inference time.

2.1. DNN Data Type Approximation

In the neural networks field, a common approximation approach is to reduce the precision and data type of weights and activation's values. More in detail,

we intend to use custom floating-point and fixed-point representations with different precision (i.e., bit-width) at the inference time.

The *darknet* framework leverages on 32-bit floating-point data types only. Therefore, we modified the *darknet* source code to allow data type conversions. All the conversions between the standard 32-bit floating-point and **custom data type** have been carried out by integrating two open source libraries into the *darknet* framework: the *libfixmath* library [41] for managing fixed-point and the *FloatX* library.

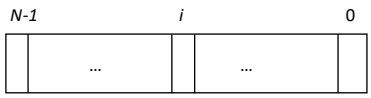


Figure 1: Custom Data Type.

Figure 1 illustrates our custom data type. It is defined as following:

- N: it determines the data bit-width;
- i: it determines the dynamics and the precision of the data type depending on the data representation:
  - Floating Point: i is the mantissa width, $N-1-i$ is the exponent width;
  - Fixed Point: i is the fractional width, $N-1-i$ is the integer width.

Since the end-goal is to characterize fault effect propagation through the network (speeding up computations and compacting the model size are out of the scope of this work), we performed on-line conversions while maintaining all internal operations in floating-point (Figure 2). The benefits coming from this approach are two-fold: first, it is not necessary to change the framework structure every time new experiments with a different data type have to be performed; second, it allows to change the representation without retraining the DNN model for each data type, exploiting the same set of trained parameters. In this way, the assessment of the CNN reliability is quicker; it is possible to

switch between experiments with different numerical formats in a reasonable amount of time. To confirm the choice suitability, we performed a preliminary experiment to confirm that the use of the online conversion does not introduce important accuracy differences while providing execution time benefits. To this end, in the experiments, we converted the data representation of the darknet framework from 32-bit floating point to 32-bit fixed point. In this way, all the operations were performed in the fixed-point arithmetic domain by using the *libfixmath* library [41]. We executed the inference of 70,000 images from the MNIST database with the LeNet-5 CNN (more details in Section 3.1) with both versions, i.e., the original floating-point one and the modified fixed-point one. Specifically, we did not retrain the CNN. The results of the experiments showed that the average accuracy error of the fixed-point version with respect to the floating-point one was of -0.01%, i.e., a slight accuracy increase. Moreover, while the execution time of the fixed-point version experiment was 8,566 seconds, i.e., $\approx$ 0.122 seconds per image, the execution time of the original floating-point version was 3,280 seconds, i.e., $\approx$ 0.047 seconds per image. Therefore, the use of the *darknet* 32-bit floating-point framework with on-line conversion allows us to run 2.6x faster experiments with respect to converting the darknet datatype, while not incurring in significant inference accuracy differences.

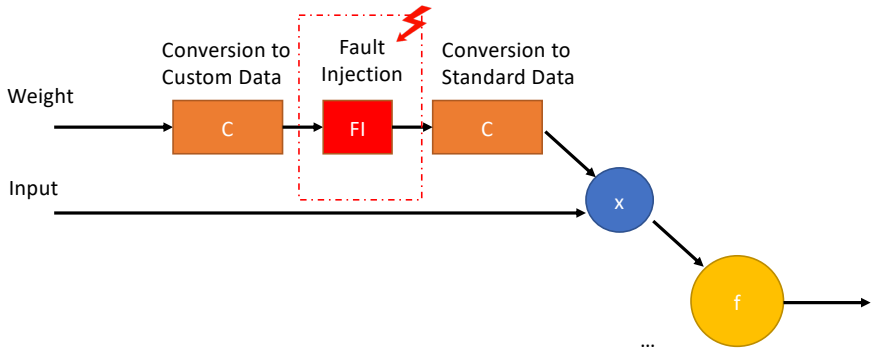


Figure 2: On-line Weights Conversions.

For the sake of completeness, we describe how the on-line conversion of a

32-bit floating-point weight is applied before reaching a single neuron (Figure 2). The applied scheme works in the following way:

1. The weight is converted from standard 32-bit floating-point to a given custom data type representation.
2. The custom data type weight is corrupted according to a chosen fault list and a fault location, i.e., the fault is injected.
3. The custom data type weight is converted back to the standard 32-bit floating-point representation in order to preserve the native implementation of the framework. In such a way, the gained value reflects the same fixed-point corrupted value, while still remaining a floating-point data.
4. The weight is multiplied by the input value.
5. The neuron performs the arithmetic computations.

Although all network operations are executed between 32-bit floating-point variables, it should be outlined that the loss of precision caused by the first conversion is preserved. Indeed, when moving from the standard 32-bit floating-point representation to a low-precision one (e.g., 16-bit fixed-point), we are witnessing a truncation error effect. Then, converting back from a narrow range of value to a wider one, the truncation error still remains.

2.2. Fault Injector

The intent of the section is to describe the Fault Injection (FI) scenario and the environment built on the *darknet* framework.

FI Scenario: Figure 3 illustrates the scenario in which the fault injection campaign is executed. First of all, we work with the trained CNN with 32-bit floating-point data types, called **Standard**. The trained network is approximated using custom data type representation, called **Custom**. The inference

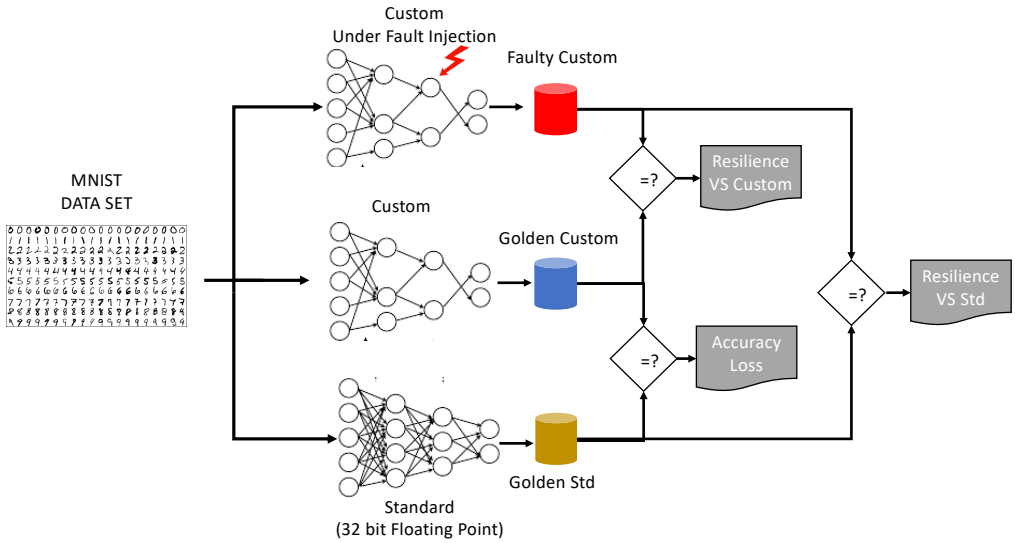


Figure 3: Fault Injection Scenario.

outputs are stored (i.e., the "Golden Std" and the "Golden Custom") and compared to determine the **Accuracy Loss** due to the approximation. The fault injection campaign is carried out on the Custom CNN and the faulty inference outputs are stored in the "Faulty Custom" log. The latter is then compared with the "Golden Custom" and the "Golden Std" to assess the resilience.

The Custom CNN (i.e., approximate) is intended to replace the Standard CNN in edge/resource-limited devices. Thus, we have to assess the reliability of the Custom CNN with respect to the standard CNN. On the other hand, it is also possible to train directly the custom-data-type CNN. In this case, the reference with respect to the reliability has to be assessed is the Custom CNN itself.

FI Environment: The hardware system can be affected by faults caused by physical manufacturing defects. As Figure 4 highlights, faults could propagate through the different hardware structures composing the full system. However, it could happen that they are masked during the propagation either at the technological or at the architectural level [42]. When a fault reaches the soft-

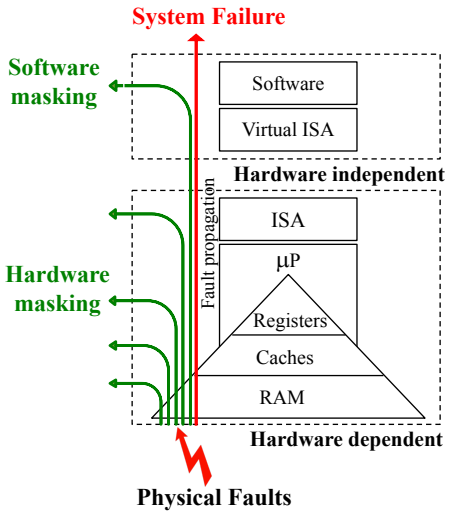


Figure 4: System Layers and Fault Propagation.

ware layer of the system, it can corrupt data, instructions or the control flow. These errors may impact the correct software execution by producing erroneous results or prevent the execution of the application leading to abnormal termination or application hangs. The software stack can play an important role in masking errors; at the same time, this phenomenon is implicitly important for the system reliability but a hard challenge for engineers that have to ensure the safeness of their systems.

As stated in the Introduction, HW faults can be transient or permanent induced by external perturbations (i.e., in a harsh environment) or due to silicon wearout and aging effects. It is important to stress once more, that we perform fault injection into synaptic weights that are **constant values** (i.e., never rewritten in the memory). It means that, even for transient faults, once the fault is triggered, it behaves exactly like a permanent one since the flipped memory cell will not be rewritten. For this reason, as target fault model, we consider the Stuck-at Fault (SaF) model at 0/1 ($SaF0$ and $SaF1$). The Fault Location (FLo) is defined by (1).

```
1 run_CNN (CNN, standard, golden_prediction_std);
2 run_CNN (CNN, custom, golden_prediction_cst);
3 for (i=0; i < FLo.size(); i++) {
4   inject_fault (Flo[i], CNN);
5   run_CNN (CNN, custom, faulty_prediction);
6   compare (faulty_prediction, golden_prediction_cst);
7   compare (faulty_prediction, golden_prediction_std);
8   release_fault (Flo[i], CNN);
9 }
```

Listing 1: Fault Injection Pseudo-Code

$$FLo =< Layer, Connection, Bit, Polarity > \tag{1}$$

where *Layer* corresponds to the CNN layer, *Connection* is the edge connecting one node of the *Layer*, and *Bit* is one of the bits of the weight associated with the *Connection*. Finally, the *Polarity* can be '0' or '1' depending on the SaF. The Fault Injector actually works at the software layer, and its pseudo-code is provided in the $Pseudo - Code$ (Listing 1). It corresponds to a simple serial fault injector that modifies the CNN topology as described by (1).

The FI process follows the scenario depicted in Figure 3 and consists of the following: once the CNN is fully trained, a golden run is performed collecting the golden results (i.e., standard and custom golden prediction), i.e., lines 1 and 2 in Listing 1. Then, the actual fault injection process is performed. The initial step requires generating the list of faults to be injected. This fault list should be seen as a list of places to inject the faults, as previously described. Then, for any fault in the fault list (line 3), a prediction run is performed and the results collected and named as faulty_prediction. It is necessary to underline that faults are injected regardless of their polarity (stuck-at-0 or stuck-at-1). Once the fault location is fixed, the target bit is inverted (if '0' it becomes a '1' and *vice-versa*). In this way, we do not distinguish between the singular effect of the two fault models while obtaining great flexibility for the considerable

amount of performed simulations. At this point (lines 6 and 7), the obtained results are compared with the expected ones (again for both the standard and custom CNN), and the results are logged for further analysis.

In detail, the function *compare* of the Pseudo-code (Listing 1) classifies the prediction/classification of the faulty CNN with respect to the golden one. The classification depends on the CNN type. In our paper, we consider two types of CNNs: (i) a classifier and (ii) an object detector. Concerning the classifier, outputs of the faulty CNN are labelled as follows:

- **Masked:** No difference is observed between the faulty CNN and the golden one.
- **Observed:** A difference is observed between the faulty CNN and the golden one. Depending on how much the results diverge, we further classify these as:
  - **Good:** The confidence score of the top-ranked is higher with respect to the golden CNN. In other words, the faulty CNN provides a better inference than the golden one;
  - **Accept:** The confidence score of the top-ranked element is reduced by less than 5% with respect to the golden CNN;
  - **Warning:** The confidence score of the top-ranked element is reduced by more than 5% with respect to the golden CNN;
  - **Critical:** The top-1 prediction is different. In other words, the faulty CNN makes a wrong inference.

From a safety assessment perspective, we consider three classes of faults *Critical*, *Warning*, and *Accept* as events **reducing the CNN safety**. Indeed, whenever one of those fault classes occurs, either the top-1 prediction is different (*Critical*) or the top-1 prediction confidence level decreases (*Warning*, *Accept*). On the other hand, the two fault classes *Masked* and *Good* either **leave the safety of the CNN unaltered** (Masked) **or even improve it** (Good).

On the other hand, the effects of the injected faults on the object detector CNN are classified differently. The CNN output is an image having bounding boxes indicating the detection of the objects. To assess whether any two bounding boxes overlap, we use the intersection over union (IoU) metric as defined in [12]. IoU is the ratio of the intersection area over the union area of two bounding boxes. The closer IoU is to 1, the higher overlap the two bounding boxes have. Thanks to the IoU metric, we can redefine the fault outcome as follows:

- **Masked:** No difference is observed between the faulty CNN and the golden one.
- **Observed:** A difference is observed between the faulty CNN and the golden one. By using the IoU calculated between the boxes of the golden CNN and those of the faulty one, we further classify the observed outcomes as follows:
  - **Accept:** The IoU is lower than 1 and higher than 0.95;
  - **Warning:** The IoU is lower than 0.95 and higher than 0.9;
  - **Critical:** The number of bounding boxes is different, or the label associated with the boxes does not match the good ones. In other words, the faulty CNN identified wrong objects. Moreover, if the IoU is lower than 0.9, the fault is classified as critical, meaning that the faulty CNN correctly identifies the objects, but it is not able to locate them precisely enough.

We did not consider the "Good" outcome since it does not make sense for object detection. It is worth noting that the fault classification used for the object detector is more stringent than the one used for the classifier. Indeed, we consider the object detection task more critical than the classification task. From a safety assessment perspective, we consider faults falling in the *Critical*, *Warning*, and *Accept* classes as events reducing the CNN safety. Conversely, *Masked* faults leave the safety of the CNN unaltered.

3. Experimental Results

This section first details the two case studies and the related fault injection campaigns, then it discusses the experimental results.

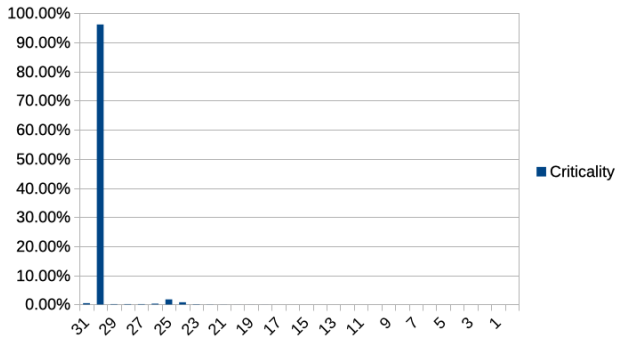
3.1. First Case Study

Among the existing CNNs, our interest falls on LeNet-5 [43], a well-known classifier for handwritten digit recognition tasks introduced by Y. Lecun et al. in 1998. The network architecture is composed of 1 input layer, 5 hidden layers, and 1 output layer, whose typology ranges from Convolutional, Fully-Connected, and Max-Pool layers. For running the experiments, the MNIST database [44], a well-known dataset used to evaluate the accuracy of new emerging models, has been selected. It is composed of 60,000 images for training and 10,000 for test/validation of the model, encoded in 28×28 pixels in grayscale. However, to lower the computational cost and time, a workload of 2,023 images was randomly selected from the MNIST test/validation dataset for the experiments. Moreover, since we are focusing on the inference phase and on the response of the network in a faulty scenario, a set of pretrained weights has been adopted. It is available from the *darknet* website and includes all weights in a 32-bit floating-point representation.

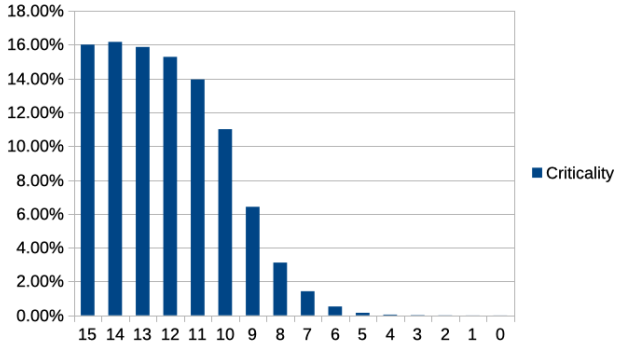
3.2. Custom Data Type

The selection of the custom data type is not trivial. Let us resort to an example to illustrate that point. In our preliminary work [11], we analyzed different fixed-point data types in terms of resilience to faults. One of those was configured with $N = 16$ and $i = 8$, meaning that 8 bits were devoted to represent the fractional part and 8 bits the integer one. Figures 5a and 5b show the "criticality" of each bit of the data type. It can be seen that in the 32-bit floating-point data type, only one bit (i.e., the bit 30th) is the main responsible for the Critical Observed faulty behaviors: up to 95% of critical observed faulty behaviors is due to a fault at bit 30th. On the other hand, in the 16-bit fixed-point representation (with 8 bits for integers and 8 bits for the fractional part),

the number of bits responsible for the Critical Observed faulty behaviors is six. This means that the custom CNN has a lower memory footprint of 50%, but it also shows a lower resilience with respect to the standard CNN, since a higher number of faulty bits may seriously affect the inference results.



(a) Standard 32-bit floating-point resilience



(b) Custom 16-bit fixed-point resilience

Figure 5: Critical Bits

To carefully select the custom data representation, we first analyze the weight distribution of LeNet-5. It is illustrated in Figure 6 and evidences that all values are in the range -0.6 to 0.6 with most of them around zero. From this analysis, we simply deduced that the data type does not need higher dynamics while a high precision is preferred. Hence, we selected the custom data types reported in Table 1. Two data types are used, the fixed and floating point with different bit width. Moreover, we computed the accuracy loss of the CNN

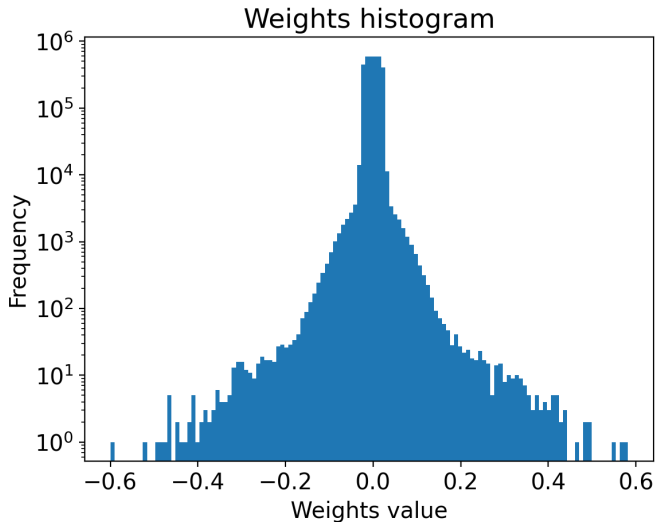


Figure 6: Distribution of pretrained Weights Values for *LeNet-5* network.

Table 1: LeNet-5 Data Type Accuracy Loss [%]

Scenario	**Data type**	**Bit-width**	**Bit encoding**	**[%] Accuracy Loss**
FP32	floating-point	32	1 sign, 8 exponent, 23 fractional	Ref.
FP16	floating-point	16	1 sign, 5 exponent, 10 fractional	0%
FP8	floating-point	8	1 sign, 4 exponent, 3 fractional	0.02%
FxP32	fixed-point	32	1 integer, 31 fractional	0%
FxP16	fixed-point	16	1 integer, 15 fractional	0%
FxP8	fixed-point	8	1 integer, 7 fractional	0.04%

resulting from the adoption of custom data type weights. As highlighted in Table 1, five different scenarios have been analyzed. The second column of the table reports the data type used in each FI campaign, while the third column reports the bit-width of the weights. The fourth column shows the amount of bits allocated to encode the different parts of the number, i.e., sign, exponent, and fractional parts in the case of floating-point representations, and the integer and fractional part in the case of fixed-point data. To compute the accuracy of the CNN in the different scenarios, the inference of the images belonging to the validation set of the MNIST database (10,000 images) has been run on LeNet-

5, clearly without injecting any faults, i.e., in a golden scenario. The results showed that only when reducing the bit width to 8 bits, the network exhibited some accuracy loss. In detail, for the network with weights encoded by using 8-bit floating-point variables (scenario FP8), the accuracy loss was 0.02%, while it was 0.04% when the weights were encoded by using 8-bit fixed-point variables (scenario FxP8).

3.3. Fault List

In this section, we discuss the complexity of the considered CNN in terms of fault injection. Furthermore, we illustrate the approach that we used to manage this complexity by performing a statistically meaningful subset of fault injections. For this complex evaluation, we consider only four LeNet-5 layers that perform arithmetic computations involving trainable weights, i.e., two convolutionals and two Fully Connected. Indeed, we consider the resilience of the CNN against faults striking the memory, where the weights are stored. Table 2 provides details about the configuration as well as the fault list of each layer. The first two rows (labeled *"Layer"* and *"Detail"*) of the table present the target layers; the third one (*"Connections"*) specifies the amount of their connection weights. The number of possible faults is computed as the multiplication between the connections number (*"Connections"*) and the weight size (*"Bit-width"*).

As the rows *"#Faults"* point out, the overall number of possible faults is very high and this reflects in a non-manageable FI campaign execution time. Thus, to reduce the fault injection execution time, we can randomly select a subset of faults. To obtain statistically significant results with an error margin of 1% and a confidence level of 99%, an average of 15.6k fault injections have to be considered for 32-bit scenarios (FP32 and FxP32), 15k for 16-bit scenarios (FP16 and FxP16), and 13.8k for 8-bit scenarios (FP8 and FxP8). The precise numbers are given in the rows of Table 2 labeled *#Injections"* and they have been computed by using the approach presented in [45]. In details, we resorted

Table 2: LeNet-5 Fault List for Fault Injection Campaigns

	Layer	**L0**	**L2**	**L4**	**L6**
	Detail	Convolutional	Convolutional	Fully Connected	Fully Connected
	Connections	2,400	51,200	3,211,264	10,240
Scenarios FP32, FxP32	Bit-width	32	32	32	32
	#Faults	76,800	1,638,400	102,760,448	327,680
	#Injections	13,678	16,474	16,638	15,837
Scenarios FP16, FxP16	Bit-width	16	16	16	16
	#Faults	38,400	819,200	51,380,224	163,840
	#Injections	11,610	16,310	16,636	15,107
Scenarios FP8, FxP8	Bit-width	8	8	8	8
	#Faults	19,200	409,600	25,690,112	8,1920
	#Injections	8,915	15,991	16,630	13,831

to the following formula:

$$fault_injections = \frac{N}{1 + e^2 \cdot \frac{N-1}{t^2 \cdot 0.25}} \tag{2}$$

where N is the total number of fault locations (i.e., row *#Faults* of Table 2), e is the desired error margin (1%), and t depends on the desired confidence level (t=2.58 corresponds to 99% confidence level [45]). Equation 2 has an horizontal asymptotic value ($N \to \infty$) equal to 16,641, thus limiting the number of fault injections necessary to achieve an evaluation with an error margin of 1% and a confidence level of 99%. Moreover, it is worth underlining that the injections are performed by randomly selecting the faulty bit among all bits of the connection weights.

To perform the FI experiments, the weight conversions are performed as described in Section 2.1, i.e., the weights are mapped to the custom type and then reconverted back to the original format. In particular, every time a fault is placed, the randomly selected weight value must be converted to the custom representation, injected with the fault, and reconverted back to the original format to perform the inference. On the one hand, this approach requires an overhead during the fault injection phase; however, on the other hand, this technique allows performing the experiments without introducing overhead due

to external computationally intensive custom floating- and fixed-point libraries for the arithmetic operations.

3.4. Fault Injection Outcomes

We conducted two sets of experiments (see Figure 3). In the first one, we evaluated the reliability by using as reference the **Standard** 32-bit floating-point CNN. This is useful in the case where a designer wants to approximate the CNN (i.e., change its data type and/or bit-width) after that it has been trained. In the second one, we assess the CNN reliability by using as reference the **Custom** fault-free CNN. This is useful in the case where a designer wants to train directly the custom data-type/bit-width CNN. To discuss the results, we refer to the classification presented in Section 2.2. In particular, we want to evaluate the safety of the different CNN versions, when subject to faults. Therefore, we consider faults in the classes *Critical*, *Warning*, and *Accept* as events reducing the CNN safety. The sum of these contributions is represented by symbol '$<$' in tables. Conversely, we consider the faults in the classes *Masked* and *Good* as events either leaving the safety of the CNN unaltered or even improving it. The sum of these contributions is represented by symbol '$\geq$' in tables.

The results of the first set of experiments (i.e., having the Standard 32-bit floating-point CNN as reference) are shown in Table 3 where each row corresponds to one of the CNN variants (FP32-FP16-FP8-FxP32-FxP16-FxP8 defined in Table 1). Each column corresponds to a faulty behavior class as described in Section 2.2.

First, we can note a different resilience to faults depending on the data type: floating *versus* fixed. More in detail, **the safety decreasing effect is lower for the fixed-point than for the floating-points**, for a given bit-width. As an example, we may resort to scenarios FP32 and FxP32 (32-bit CNNs): the *safety increasing (decreasing)* effect varies from 69% (31%) of the floating-point version (scenario FP32) to 74% (26%) of the fixed-point version (scenario FxP8). This corresponds to a difference of 5%. The average difference between

Table 3: LeNet-5 Fault Injection outcomes w.r.t. Golden Std.

Layer	Data	Observed				Masked	Safety		Gain w.r.t. FP32	
		Critical	Warning	Accept	Good		<	≥	Safety*	Memory
L0	FP32	1.32%	0.06%	29.96%	28.98%	39.68%	31.34%	68.66%	-	-
	FP16	2.61%	0.12%	53.28%	41.27%	2.71%	56.01%	43.98%	-24.67%	2X
	FP8	3.41%	0.91%	64.13%	31.53%	0.02%	68.45%	31.55%	-37.11%	4X
	FxP32	0.03%	0.04%	25.93%	25.54%	48.47%	26.00%	74.01%	+5.34%	0
	FxP16	0.05%	0.08%	49.98%	46.94%	2.96%	50.11%	49.90%	-18.77%	2X
	FxP8	0.45%	0.60%	46.74%	52.18%	0.03%	47.79%	52.21%	-16.45%	4X
L2	FP32	1.37%	0.02%	23.88%	23.39%	51.34%	25.27%	74.73%	-	-
	FP16	2.59%	0.08%	55.00%	38.57%	3.76%	57.67%	42.33%	-32.40%	2X
	FP8	1.10%	0.91%	65.86%	32.10%	0.03%	67.87%	32.13%	-42.60%	4X
	FxP32	0.01%	0.01%	23.82%	23.62%	52.54%	23.84%	76.16%	+1.43%	0
	FxP16	0.03%	0.02%	49.87%	46.61%	3.47%	49.92%	50.08%	-24.65%	2X
	FxP8	0.42%	0.45%	46.57%	52.53%	0.03%	47.44%	52.56%	-22.17%	4X
L4	FP32	0.71%	0.00%	3.85%	3.86%	91.57%	4.56%	95.44%	-	-
	FP16	0.84%	0.13%	57.79%	36.12%	5.13%	58.75%	41.25%	-54.19%	2X
	FP8	0.49%	0.06%	66.69%	32.72%	0.04%	67.24%	32.76%	-62.67%	4X
	FxP32	0.00%	0.00%	10.99%	11.49%	77.52%	10.99%	89.01%	-6.43%	0
	FxP16	0.00%	0.00%	49.40%	45.59%	5.01%	49.40%	50.60%	-44.84%	2X
	FxP8	0.40%	0.36%	46.38%	52.82%	0.04%	47.14%	52.86%	-42.58%	4X
L6	FP32	0.61%	0.01%	7.69%	8.14%	83.54%	8.32%	91.68%	-	-
	FP16	1.19%	0.03%	56.34%	37.65%	4.78%	57.57%	42.43%	-49.25%	2X
	FP8	1.76%	0.40%	65.07%	32.74%	0.04%	67.22%	32.78%	-58.91%	4X
	FxP32	0.01%	0.04%	13.50%	14.11%	72.33%	13.55%	86.45%	-5.24%	0
	FxP16	0.03%	0.08%	49.15%	46.11%	4.63%	49.26%	50.74%	-40.94%	2X
	FxP8	0.44%	0.53%	46.28%	52.72%	0.04%	47.25%	52.75%	-38.93%	4X

* Safety increasing effect difference between a given scenario and FP32

floating- and fixed-point versions with respect to safety increasing/decreasing effect over the three variants (32, 16, and 8 bits) is 8.96% over all the layers. This can be seen by comparing the scenarios FP32 with FxP32, FP16 with FxP16, and FP8 with FxP8, in terms of the average safety increase/decrease effect variation (columns 8 and 9). Is it worth highlighting that, in general terms, the safety decreasing effect is critical only in a few cases. The percentage

of critical faults is always lower than 3.42% for all the variants. In particular, fixed-point variants have a very small percentage of critical faults, always lower than 0.46%. Moreover, the contribution of *Good* faults to the safety increasing effect turns out to be significant, especially for 16- and 8-bit versions. As an example, in the scenario FxP8 for the layer L0, we observed a safety increasing effect in 52.21% of the cases, with a 52.18% of *Good* faults.

Furthermore, the bit-width plays an important role for the reliability: **the lower the bit-width the lower the resilience**. Therefore, a designer who wanted to use a more efficient version of the CNN (reduced memory footprint) has to be aware that it would be also less resilient with respect to the original CNN (FP32). However, it is worth also remarking that using fixed-point data representation, instead of the floating-point counterpart, provides the better results in terms of trade-off between resilience and efficiency. This is reported in the last two columns of Table 3. For instance, we may compare scenarios FP8 and FxP8 (8-bit CNNs) for layer L0: we observed a safety loss with respect to FP32 of 37% in the floating-point version (scenario FP8) and only of 16% in the fixed-point version (scenario FxP8). Therefore, choosing the CNN in the scenario FxP8, namely 8-bit fixed-point (1 bit for integer and 7 bits for fractional), allows the designer to compact the memory footprint by a 4x factor while reducing the safety only by 16%. Moreover, by looking more closely, the occurrence of critical faults in scenario FxP8 even decreases from 1.32% of FP32 to 0.45%, while in scenario B it increases to 3.41%. Additionally, for scenario FxP32 (32-bit fixed-point CNN), it has been observed that the CNN achieves improved safety with respect to FP32 scenario for the same memory footprint for layers L0 and L2 (+5.34% and +1.43%, respectively). Thus, simply changing the CNN data type to a fixed-point representation may improve its resilience for some layers.

Table 4 reports the results of the second set of experiments (i.e., having the Custom CNN as a reference). While in the first set we compared the FI results of each scenario to the ones obtained with the fault-free 32-bit floating-point CNN (FP32), in this set of experiments *we compare the results of each*

scenario to the results obtained with the corresponding fault-free custom CNN. This scenario corresponds to directly training the custom-data-type CNN, so the reliability has to be assessed with respect to the Custom CNN itself. For details, see Figure 3. Note that the row related to the FP32 version is the same one in both experiment sets.

Table 4: LeNet-5 Fault Injection outcomes w.r.t. Golden Custom

Layer	Data	Observed				Masked	Safety		Gain w.r.t. FP32	
		Critical	Warning	Accept	Good		$<$	$\geq$	Safety*	Memory
L0	FP32	1.32%	0.06%	29.96%	28.98%	39.68%	31.34%	68.66%		-
	FP16	2.61%	0.12%	42.10%	40.86%	14.30%	44.84%	55.16%	-13.50%	2X
	FP8	3.30%	0.88%	47.08%	44.67%	4.07%	51.26%	48.74%	-19.91%	4X
	FxP32	0.03%	0.04%	24.45%	23.81%	51.67%	24.51%	75.49%	+6.83%	0
	FxP16	0.05%	0.08%	41.96%	40.81%	17.10%	42.09%	57.91%	-10.75%	2X
	FxP8	0.11%	0.15%	49.03%	46.94%	3.76%	49.29%	50.71%	-17.95%	4X
L2	FP32	1.37%	0.02%	23.88%	23.39%	51.34%	25.27%	74.73%		-
	FP16	2.59%	0.08%	35.66%	34.59%	27.07%	38.34%	61.66%	-13.07%	2X
	FP8	0.96%	0.89%	46.21%	43.17%	8.77%	48.06%	51.94%	-22.79%	4X
	FxP32	0.01%	0.01%	21.41%	20.93%	57.64%	21.43%	78.57%	+3.84%	0
	FxP16	0.03%	0.02%	38.70%	37.74%	23.52%	38.75%	61.25%	-13.47%	2X
	FxP8	0.06%	0.05%	47.94%	45.79%	6.17%	48.05%	51.95%	-22.77%	4X
L4	FP32	0.71%	0.00%	3.85%	3.86%	91.57%	4.56%	95.44%		-
	FP16	0.84%	0.13%	6.21%	6.23%	86.59%	7.17%	92.83%	-2.61%	2X
	FP8	0.32%	0.06%	10.45%	10.11%	79.06%	10.83%	89.17%	-6.26%	4X
	FxP32	0.00%	0.00%	4.06%	4.02%	91.92%	4.06%	95.94%	+0.50%	0
	FxP16	0.00%	0.00%	7.83%	7.75%	84.42%	7.83%	92.17%	-3.27%	2X
	FxP8	0.01%	0.00%	10.67%	10.33%	78.99%	10.68%	89.32%	-6.12%	4X
L6	FP32	0.61%	0.01%	7.69%	8.14%	83.54%	8.32%	91.68%		-
	FP16	1.19%	0.03%	11.47%	12.63%	74.67%	12.70%	87.30%	-4.39%	2X
	FP8	1.59%	0.39%	15.87%	17.10%	65.05%	17.85%	82.15%	-9.53%	4X
	FxP32	0.01%	0.04%	7.15%	7.27%	85.52%	7.21%	92.79%	+1.11%	0
	FxP16	0.03%	0.08%	13.48%	13.72%	72.69%	13.59%	86.41%	-5.27%	2X
	FxP8	0.05%	0.16%	17.25%	18.27%	64.27%	17.47%	82.53%	-9.15%	4X

* Safety increasing effect difference between a given scenario and FP32

In general, the trends highlighted in the first set of experiments are observed

also in this scenario: (i) the safety is impacted by the bit-width reduction; (ii) for fixed-point CNN versions, we observed a more graceful safety decrease compared to floating-point versions. Besides that, an interesting effect is that for floating-point custom variants (FP16 and FP8) the difference between the two sets of experiments (Tables 3 and 4) in terms of Safety gain with respect to FP32 is higher than for fixed-point ones (FxP32, FxP16, and FxP8). For example, in layer L0, for variant FP8 (8-bit floating-point) the Safety gain with respect to FP32 is -37.1% for the first set of experiments (Table 3) and -19.9% for the second set of experiments (Table 4), that is a 17.2% difference. Conversely, for variant FxP8 (8-bit fixed-point) the Safety gain with respect to FP32 is -16.4% for the first set of experiments (Table 3) and -17.9% for the second set of experiments (Table 4), which is a difference of 1.5%. On average, the difference between the two sets of experiments for floating-point variants (FP16 and FP8) over all the layers is 33.72%, while for fixed-point custom variants (FxP32, FxP16, and FxP8) is 14.81%. Practically, this means that a designer who chose to approximate the original Standard FP32 CNN version by using a custom floating-point variant without retraining it, would be exposed to higher safety degradation than by using the fixed-point alternative with the same bit-width. When a training is performed directly on the custom-data-type CNN, the safety degradation difference between the floating-point variants and the fixed-point ones is smaller. However, fixed-point ones still guarantee less critical fault occurrence, i.e., less than 0.12%.

Finally, as a result of the analysis, we think that in general, the CNN in FxP8 scenario provides the best results in terms of memory footprint reduction, i.e., 4X, with a significant resilience, i.e. less than 0.45% critical faults.

3.5. Second Case Study

The second case study targets YOLO [46], a CNN capable of detecting objects in real time, analyzing up to 45 frames per second. YOLO is a predictor CNN used to detect a certain set of objects (i.e., the list of recognizable objects came from [40]). In Figure 7, the prediction results are shown highlighting the

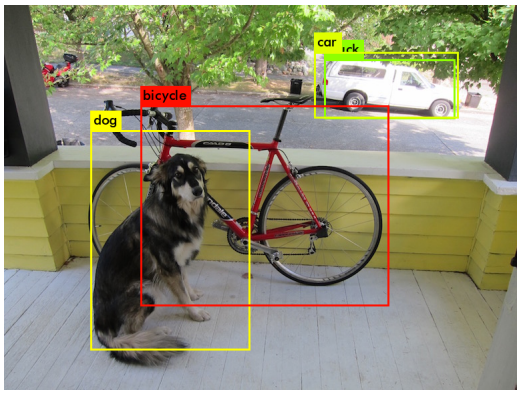


Figure 7: YOLO CNN prediction result

identified objects including the object name and area occupation. This example presents an image containing three relevant objects to be detected: one dog, one bike, and one car. Moreover, the car is further recognized as a truck for a total of four objects detected.

3.6. Custom Data Type

To carefully select the custom data representation, we first analyzed the the weight distribution of YOLO. It is shown in Figure 8. As evidenced, all

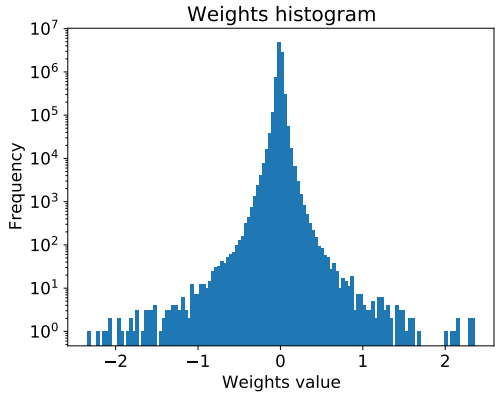


Figure 8: Distribution of pretrained Weights Values for *YOLO* network

values are in the range -2.35 to 2.36 with most of them around zero. Thus,

Table 5: YOLO Data Type Accuracy Loss [%]

Scenario	Data type	Bit-width	Bit encoding	[%] Accuracy Loss
FP32	floating-point	32	1 sign, 8 exponent, 23 fractional	Ref.
FP16	floating-point	16	1 sign, 5 exponent, 10 fractional	42% masked, 58% acceptable
FP8	floating-point	8	1 sign, 4 exponent, 3 fractional	28% warning, 72% critical
FxP32	fixed-point	32	3 integer, 29 fractional	100% masked
FxP16	fixed-point	16	3 integer, 13 fractional	42% masked, 58% acceptable
FxP8	fixed-point	8	3 integer, 5 fractional	100% critical

Masked: IoU=1; acceptable: 0.95<IoU<1; warning: 0.9≤IoU≤0.95; critical: IoU<0.9 or different objects recognized

also for this CNN, the data type does not need higher dynamics while a high precision is preferred. Therefore, we selected the custom data type reported in Table 5. Moreover, we computed the accuracy loss of the CNN resulting from the adoption of custom data types. As highlighted in Table 5, five different scenarios have been analyzed. The second column of the table reports the data type used in each scenario, while the third column reports the bit-width of the weights. The fourth column shows the amount of bits allocated to encode the different parts of the number, i.e., sign, exponent, and fractional parts in the case of floating-point representations, and integer and fractional part in the case of fixed-point ones. To compute the accuracy loss of the network in the different scenarios, the inference for 7 images has been run on YOLO, clearly without injecting any faults, i.e., in a golden scenario. The results are reported according to the classification reported in Section 2.2 for the object detector. The results show that for the FxP32 scenario there is no degradation, for the 16-bit data types (both FP16 and FxP16) in 42% of the cases (3 images out of 7) there is no degradation and for 58% of the cases (4 images out of 7) there is an acceptable degradation (i.e., all objects are correctly recognized and the IoU metric is between 0.95 and 1). Finally, when using the 8-bit floating-point data type (FP8), the CNN is able to deliver usable results (i.e., classified as *warning*, 0.9≤IoU≤0.95) for 28% of the inputs (2 images out of 7) and cannot produce correct outputs (*critical*) for the rest (5 images out of 7). On the contrary, with the 8-bit fixed-point data type (FxP8), the CNN is unable to provide the correct

results at all, i.e., for all input images the output was classified as *critical*.

3.7. Fault List

In this section, we present the complexity of YOLO in terms of fault injection. As for LeNet-5, also for YOLO we consider only the layers performing arithmetic computations involving trainable weights, i.e., the thirteen convolutional layers. Table 6 provides details about the configuration as well as the fault list of each layer. The first column (labeled *"Layer"*) reports the target layers; the second one (*"Connections"*) specifies the number of connection weights. The number of possible faults is computed as the multiplication between the connections number (*"Connections"*) and the weight size (*"Bit-width"*).

Table 6: YOLO Fault List for Fault Injection Campaigns

		FP32 and FxP32 Bit-width=32		FP16 and FxP16 Bit-width=16		FP8 and FxP8 Bit-width=8	
Layer	Connections	#Faults	#Injections	#Faults	#Injections	#Faults	#Injections
L0	432	13,824	7,551	6,912	4,884	3,456	2,862
L2	4,608	147,456	14,954	73,728	13,577	36,864	11,466
L4	18,432	589,824	16,184	294,912	15,752	147,456	14,954
L6	73,728	2,359,296	16,524	1,179,648	16,410	589,824	16,184
L8	294,912	9,437,184	16,612	4,718,592	16,583	2,359,296	16,524
L10	1,179,648	37,748,736	16,634	18,874,368	16,626	9,437,184	16,612
L12	4,718,592	150,994,944	16,639	75,497,472	16,637	37,748,736	16,634
L13	262,144	8,388,608	16,608	4,194,304	16,575	2,097,152	16,510
L14	1,179,648	37,748,736	16,634	18,874,368	16,626	9,437,184	16,612
L15	130,560	4,177,920	16,575	2,088,960	16,509	1,044,480	16,380
L18	32,768	1,048,576	16,381	524,288	16,129	262,144	15,648
L21	884,736	28,311,552	16,631	14,155,776	16,621	7,077,888	16,602
L22	65,280	2,088,960	16,509	1,044,480	16,380	522,240	16,127

As the columns *"#Faults"* point out, the overall number of possible faults is very high and this reflects in a non-manageable fault injection campaign execution time. As done for LeNet-5, we selected a subset of faults in order to reduce the fault injection execution time. For each layer, we injected the number of faults reported in the columns *"#Injections"*. The number of faults to inject was obtained by using the approach presented in [45] (see Section 3.3) and it

is statistically representative with an error margin of 1% and a confidence level of 99%. On average, 15.7k faults are injected in each layer for 32-bit scenarios (FP32 and FxP32), 15.3k for 16-bit scenarios (FP16 and FxP16), and 14.8k for 8-bit scenarios (FP8 and FxP8). The injections are performed by randomly selecting the faulty bit among all bits of the connection weights.

3.8. Fault Injection Outcomes

In this section, we report the results of the FI campaign on the YOLO CNN. In line with the LeNet-5 FI campaign, two sets of experiments are carried out. Firstly, we evaluated the reliability by using as reference the **Standard** 32-bit floating-point CNN and then by using the **Custom** fault-free one. The results are reported in terms of the classification presented in Section 2.2. In the following, we define faults belonging to the classes *Critical*, *Warning*, and *Accept* as events reducing the CNN safety. The sum of these contributions is represented by the symbol '<' in the tables. On the other hand, we consider the masked faults as events leaving the safety of the CNN unaltered. Thus, their contribution is represented by the symbol '=' in Tables.

Table 7 reports the results of the first set of FI campaigns.

Table 7: YOLO Fault Injection outcomes w.r.t. Golden Std.

Level	Data	Observed			Masked	Safety		Gain w.r.t. FP32	
		Critical	Warning	Accept		<	=	Safety*	Memory
L0	FP32	20.17%	0.66%	16.25%	62.92%	37.08%	62.92%	-	-
	FP16	30.70%	1.04%	41.32%	26.95%	73.05%	26.95%	-35.97%	2x
	FP8	48.46%	49.88%	1.66%	0.00%	100.00%	0.00%	-62.92%	4x
	FxP32	13.35%	0.54%	12.43%	73.68%	26.32%	73.68%	10.76%	0
	FxP16	27.23%	1.11%	44.52%	27.14%	72.86%	27.14%	-35.78%	2x
L2	FP32	6.81%	0.10%	14.46%	78.63%	21.37%	78.63%	-	-
	FP16	12.07%	0.17%	49.85%	37.91%	62.09%	37.91%	-40.72%	2x
	FP8	16.67%	78.65%	4.68%	0.00%	100.00%	0.00%	-78.63%	4x
	FxP32	7.40%	0.43%	12.22%	79.95%	20.05%	79.95%	1.33%	0
	FxP16	14.38%	0.84%	51.95%	32.83%	67.17%	32.83%	-45.80%	2x
L4	FP32	5.05%	0.05%	11.24%	83.66%	16.34%	83.66%	-	-
	FP16	9.88%	0.10%	49.32%	40.70%	59.30%	40.70%	-42.97%	2x

	FP8	13.30%	82.95%	3.75%	0.00%	100.00%	0.00%	-83.66%	4x
	FxP32	6.32%	0.35%	11.86%	81.47%	18.53%	81.47%	-2.19%	0
	FxP16	12.45%	0.70%	52.81%	34.04%	65.96%	34.04%	-49.62%	2x
	FP32	4.02%	0.01%	7.59%	88.37%	11.63%	88.37%	-	-
	FP16	8.29%	0.04%	49.55%	42.13%	57.87%	42.13%	-46.25%	2x
L6	FP8	9.48%	88.49%	2.02%	0.00%	100.00%	0.00%	-88.37%	4x
	FxP32	4.49%	0.26%	11.04%	84.21%	15.79%	84.21%	-4.17%	0
	FxP16	8.82%	0.52%	53.75%	36.91%	63.09%	36.91%	-51.47%	2x
	FP32	3.41%	0.01%	3.94%	92.65%	7.35%	92.65%	-	-
	FP16	7.15%	0.02%	50.53%	42.30%	57.70%	42.30%	-50.35%	2x
L8	FP8	7.06%	91.51%	1.44%	0.00%	100.00%	0.00%	-92.65%	4x
	FxP32	2.84%	0.19%	9.22%	87.75%	12.25%	87.75%	-4.90%	0
	FxP16	5.49%	0.38%	54.32%	39.81%	60.19%	39.81%	-52.84%	2x
	FP32	3.26%	0.00%	1.70%	95.04%	4.96%	95.04%	-	-
	FP16	6.42%	0.02%	52.08%	41.49%	58.51%	41.49%	-53.55%	2x
L10	FP8	4.93%	94.16%	0.91%	0.00%	100.00%	0.00%	-95.04%	4x
	FxP32	1.67%	0.11%	6.85%	91.37%	8.63%	91.37%	-3.67%	0
	FxP16	3.14%	0.21%	54.33%	42.32%	57.68%	42.32%	-52.72%	2x
	FP32	3.17%	0.00%	0.84%	95.98%	4.02%	95.98%	-	-
	FP16	6.18%	0.07%	52.75%	41.00%	59.00%	41.00%	-54.98%	2x
L12	FP8	3.69%	95.54%	0.77%	0.00%	100.00%	0.00%	-95.98%	4x
	FxP32	0.76%	0.06%	5.34%	93.85%	6.15%	93.85%	-2.13%	0
	FxP16	1.39%	0.08%	54.87%	43.66%	56.34%	43.66%	-52.32%	2x
	FP32	3.27%	0.01%	3.37%	93.36%	6.64%	93.36%	-	-
	FP16	6.76%	0.04%	51.46%	41.74%	58.26%	41.74%	-51.62%	2x
L13	FP8	6.09%	92.80%	1.10%	0.00%	100.00%	0.00%	-93.36%	4x
	FxP32	1.86%	0.18%	9.08%	88.87%	11.13%	88.87%	-4.49%	0
	FxP16	3.61%	0.37%	55.13%	40.88%	59.12%	40.88%	-52.47%	2x
	FP32	3.13%	0.01%	1.25%	95.61%	4.39%	95.61%	-	-
	FP16	5.83%	0.07%	52.96%	41.13%	58.87%	41.13%	-54.47%	2x
L14	FP8	3.92%	96.08%	0.00%	0.00%	100.00%	0.00%	-95.61%	4x
	FxP32	1.03%	0.13%	5.96%	92.89%	7.11%	92.89%	-2.72%	0
	FxP16	2.05%	0.30%	54.55%	43.10%	56.90%	43.10%	-52.50%	2x
	FP32	0.82%	0.01%	0.19%	98.97%	1.03%	98.97%	-	-
	FP16	1.75%	0.04%	65.39%	32.82%	67.18%	32.82%	-66.15%	2x
L15	FP8	1.05%	98.95%	0.00%	0.00%	100.00%	0.00%	-98.97%	4x
	FxP32	0.34%	0.06%	0.36%	99.24%	0.76%	99.24%	0.26%	0
	FxP16	0.57%	0.11%	56.69%	42.63%	57.37%	42.63%	-56.35%	2x

	FP32	3.26%	0.02%	0.19%	96.53%	3.47%	96.53%	-	-
	FP16	6.10%	0.04%	53.82%	40.04%	59.96%	40.04%	-56.49%	2x
L18	FP8	5.55%	92.84%	1.61%	0.00%	100.00%	0.00%	-96.53%	4x
	FxP32	0.98%	0.14%	0.89%	98.00%	2.00%	98.00%	1.48%	0
	FxP16	2.04%	0.26%	57.61%	40.10%	59.90%	40.10%	-56.43%	2x
	FP32	3.22%	0.00%	0.06%	96.72%	3.28%	96.72%	-	-
	FP16	5.61%	0.02%	53.96%	40.41%	59.59%	40.41%	-56.31%	2x
L21	FP8	3.83%	94.51%	1.67%	0.00%	100.00%	0.00%	-96.72%	4x
	FxP32	0.37%	0.06%	0.56%	99.02%	0.98%	99.02%	2.29%	0
	FxP16	0.63%	0.17%	57.73%	41.47%	58.53%	41.47%	-55.25%	2x
	FP32	0.44%	0.00%	0.03%	99.52%	0.48%	99.52%	-	-
	FP16	0.89%	0.00%	56.66%	42.45%	57.55%	42.45%	-57.07%	2x
L22	FP8	1.14%	98.83%	0.03%	0.00%	100.00%	0.00%	-99.52%	4x
	FxP32	0.14%	0.02%	0.04%	99.79%	0.21%	99.79%	0.27%	0
	FxP16	0.29%	0.02%	57.05%	42.63%	57.37%	42.63%	-56.89%	2x

* Safety increasing effect difference between a given scenario and FP32

While for LeNet-5 we figured out that the fixed-point versions of the CNN had a higher average safety level (8.96%) with respect to the floating-point counterparts, for YOLO the difference is not as significant. Indeed, for YOLO, the fixed-point versions of the CNN have a slightly lower average safety level, i.e., -0.44%, with respect to the floating-point versions. This is calculated for the 32- and 16-bit versions, as the 8-bit fixed-point one always yields critical errors (see Table 5). If we include also the 8-bit versions, the fixed-point versions have a lower average safety level of -3.42% compared to the floating-point ones. This is probably due to the different distribution of the pretrained weight values for the YOLO network compared to LeNet-5 (compare Figure 8 with Figure 6). Indeed, for YOLO, the need of more bits for the integer part of the weights reduced the representation precision. Furthermore, it is worth highlighting that, in general terms, YOLO turns out to be much less resilient than LeNet-5. Indeed, while for LeNet-5 the percentage of critical faults is always lower than 3.42%, the YOLO network reaches up to 48.46% of critical faults (layer L0, FP8 scenario)

and, as already mentioned, the FxP8 version produces critical faults even in a fault-free scenario. This is probably due to the much more stringent definition that we used for critical faults. Indeed, for LeNet-5 we classified a fault as critical only when the top-1 prediction was wrong; conversely, for YOLO a fault is classified as critical also when an object is correctly classified but it is not located perfectly (IoU<0.9). Finally, as in the LeNet-5 case, also for YOLO reducing the bit-width implies reducing the CNN resilience.

In Table 8, we report the results of the second set of FI campaigns, where we compare the results of each scenario with the results obtained with the corresponding fault-free custom CNN.

Table 8: Yolo Fault Injection outcomes w.r.t. Golden Custom

Level	Data	Observed			Masked	Safety		Gain w.r.t. FP32	
		Critical	Warning	Accept		<	=	Safety*	Memory
L0	FP32	20.17%	0.66%	16.25%	62.92%	37.08%	62.92%	-	-
	FP16	30.70%	1.03%	28.10%	40.16%	59.84%	40.16%	-22.76%	2x
	FP8	42.08%	5.63%	37.02%	15.27%	84.73%	15.27%	-47.64%	4x
	FxP32	13.35%	0.54%	12.43%	73.68%	26.32%	73.68%	10.76%	0
	FxP16	27.23%	1.13%	23.50%	48.14%	51.86%	48.14%	-14.78%	2x
L2	FP32	6.81%	0.10%	14.46%	78.63%	21.37%	78.63%	-	-
	FP16	12.07%	0.18%	23.04%	64.71%	35.29%	64.71%	-13.92%	2x
	FP8	14.19%	2.64%	38.61%	44.55%	55.45%	44.55%	-34.07%	4x
	FxP32	7.40%	0.43%	12.22%	79.95%	20.05%	79.95%	1.33%	0
	FxP16	14.39%	0.84%	23.60%	61.16%	38.84%	61.16%	-17.46%	2x
L4	FP32	5.05%	0.05%	11.24%	83.66%	16.34%	83.66%	-	-
	FP16	9.88%	0.10%	18.29%	71.73%	28.27%	71.73%	-11.94%	2x
	FP8	11.84%	1.70%	30.70%	55.76%	44.24%	55.76%	-27.90%	4x
	FxP32	6.32%	0.35%	11.86%	81.47%	18.53%	81.47%	-2.19%	0
	FxP16	12.45%	0.70%	22.51%	64.34%	35.66%	64.34%	-19.33%	2x
L6	FP32	4.02%	0.01%	7.59%	88.37%	11.63%	88.37%	-	-
	FP16	8.29%	0.04%	13.34%	78.33%	21.67%	78.33%	-10.04%	2x
	FP8	8.37%	1.25%	22.66%	67.72%	32.28%	67.72%	-20.65%	4x
	FxP32	4.49%	0.26%	11.04%	84.21%	15.79%	84.21%	-4.17%	0
	FxP16	8.82%	0.53%	20.99%	69.66%	30.34%	69.66%	-18.72%	2x
L8	FP32	3.41%	0.01%	3.94%	92.65%	7.35%	92.65%	-	-

	FP16	7.15%	0.02%	7.80%	85.03%	14.97%	85.03%	-7.61%	2x
	FP8	6.22%	1.04%	16.41%	76.33%	23.67%	76.33%	-16.31%	4x
	FxP32	2.84%	0.19%	9.22%	87.75%	12.25%	87.75%	-4.90%	0
	FxP16	5.49%	0.38%	17.48%	76.65%	23.35%	76.65%	-16.00%	2x
	FP32	3.26%	0.00%	1.70%	95.04%	4.96%	95.04%	-	-
	FP16	6.42%	0.02%	3.89%	89.67%	10.33%	89.67%	-5.36%	2x
L10	FP8	4.42%	0.52%	10.04%	85.01%	14.99%	85.01%	-10.03%	4x
	FxP32	1.67%	0.11%	6.85%	91.37%	8.63%	91.37%	-3.67%	0
	FxP16	3.14%	0.21%	12.41%	84.23%	15.77%	84.23%	-10.81%	2x
	FP32	3.17%	0.00%	0.84%	95.98%	4.02%	95.98%	-	-
	FP16	6.18%	0.07%	2.20%	91.55%	8.45%	91.55%	-4.44%	2x
L12	FP8	3.44%	0.23%	7.94%	88.39%	11.61%	88.39%	-7.59%	4x
	FxP32	0.76%	0.06%	5.34%	93.85%	6.15%	93.85%	-2.13%	0
	FxP16	1.39%	0.08%	9.90%	88.63%	11.37%	88.63%	-7.35%	2x
	FP32	3.27%	0.01%	3.37%	93.36%	6.64%	93.36%	-	-
	FP16	6.76%	0.04%	6.42%	86.78%	13.22%	86.78%	-6.58%	2x
L13	FP8	5.20%	1.18%	14.59%	79.03%	20.97%	79.03%	-14.33%	4x
	FxP32	1.86%	0.18%	9.08%	88.87%	11.13%	88.87%	-4.49%	0
	FxP16	3.61%	0.37%	16.92%	79.09%	20.91%	79.09%	-14.27%	2x
	FP32	3.13%	0.01%	1.25%	95.61%	4.39%	95.61%	-	-
	FP16	5.83%	0.07%	2.66%	91.43%	8.57%	91.43%	-4.18%	2x
L14	FP8	3.60%	0.25%	4.60%	91.54%	8.46%	91.54%	-4.06%	4x
	FxP32	1.03%	0.13%	5.96%	92.89%	7.11%	92.89%	-2.72%	0
	FxP16	2.05%	0.30%	11.29%	86.35%	13.65%	86.35%	-9.25%	2x
	FP32	0.82%	0.01%	0.19%	98.97%	1.03%	98.97%	-	-
	FP16	1.75%	0.04%	0.34%	97.87%	2.13%	97.87%	-1.10%	2x
L15	FP8	0.92%	0.17%	0.46%	98.45%	1.55%	98.45%	-0.53%	4x
	FxP32	0.34%	0.06%	0.36%	99.24%	0.76%	99.24%	0.26%	0
	FxP16	0.57%	0.12%	0.57%	98.74%	1.26%	98.74%	-0.23%	2x
	FP32	3.26%	0.02%	0.19%	96.53%	3.47%	96.53%	-	-
	FP16	6.10%	0.04%	0.38%	93.48%	6.52%	93.48%	-3.05%	2x
L18	FP8	5.22%	0.86%	10.56%	83.35%	16.65%	83.35%	-13.18%	4x
	FxP32	0.98%	0.14%	0.89%	98.00%	2.00%	98.00%	1.48%	0
	FxP16	2.04%	0.26%	2.01%	95.70%	4.30%	95.70%	-0.83%	2x
	FP32	3.22%	0.00%	0.06%	96.72%	3.28%	96.72%	-	-
	FP16	5.61%	0.02%	0.14%	94.23%	5.77%	94.23%	-2.50%	2x
L21	FP8	3.62%	0.48%	9.10%	86.80%	13.20%	86.80%	-9.92%	4x

	FxP32	0.37%	0.06%	0.56%	99.02%	0.98%	99.02%	2.29%	0
	FxP16	0.63%	0.17%	1.19%	98.00%	2.00%	98.00%	1.28%	2x
	FP32	0.44%	0.00%	0.03%	99.52%	0.48%	99.52%	-	-
	FP16	0.89%	0.00%	0.04%	99.07%	0.93%	99.07%	-0.45%	2x
L22	FP8	1.13%	0.06%	0.21%	98.61%	1.39%	98.61%	-0.91%	4x
	FxP32	0.14%	0.02%	0.04%	99.79%	0.21%	99.79%	0.27%	0
	FxP16	0.29%	0.02%	0.10%	99.59%	0.41%	99.59%	0.07%	2x

* Safety increasing effect difference between a given scenario and FP32

First, it is immediately clear that, also for this FI campaign, the DNN safety is impacted by the bit-width reduction. Second, also in this case, YOLO exhibits high critical fault occurrence, i.e., up to 42.08% in the layer L0 in FP8 scenario. Finally, for YOLO we notice the same phenomenon that we observed for LeNet-5: for floating-point custom variants (FP16 and FP8), the difference between the two sets of experiments (Tables 7 and 8) in terms of Safety gain with respect to FP32 is higher than for fixed-point ones (FxP32, FxP16, and FxP8). Indeed, on average, the difference for floating-point variants over all the layers is 59.38%, while for fixed-point custom variants is 13.92%. As already mentioned, this means that approximating the FP32 CNN version by using a custom floating-point variant without retraining exposes to higher safety degradation than by using the same bit-width fixed-point alternative.

4. Conclusions

This paper presents a characterization framework for analyzing the impact of permanent faults affecting a Convolutional Neural Network intended to be deployed in safety-critical and resource-constrained systems. The characterization is done by means of fault injection campaigns on the *darknet* open source framework. The experiments are performed at the software level with the aim of being independent from the hardware architecture and, on the whole, to

derive a common characterization of the behavior of CNNs affected by permanent faults. This could be considered an interesting outcome since the designer, starting from these results, could be able to select the most convenient data type for his or her CNN application. Compared to previous works, we show that depending on the data type definition, we can deeply impact the reliability of the CNN. In particular, the results of the fault injection campaign showed that YOLO is less resilient than LeNet-5, according to the fault classifications used. Furthermore, for LeNet-5 CNN, fixed-point data provide a better trade-off between memory footprint reduction and network resilience compared to floating-point data. In conclusion, depending on the width of the distribution of the CNN weight values, a designer can decide to use either the fixed-point or the floating-point data type, with different bit-widths, to obtain different trade-offs between resilience and resource gain. For example, the use of 8-bit fixed-point data type for LeNet-5 provides 4X memory footprint reduction at the cost of less than 0.45% critical faults. In the future, we intend to investigate the reliability of DNNs by exploiting other data representations, such as the integer data type.

References

[1] S. Albawi, T. A. Mohammed, S. Al-Zawi, Understanding of a convolutional neural network, in: 2017 International Conference on Engineering and Technology (ICET), 2017, pp. 1–6.

[2] Recent Advances in Deep Learning for Speech Research at Microsoft, IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP).

[3] A. Krizhevsky, I. Sutskever, G. E. Hinton, Imagenet classification with deep convolutional neural networks, in: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1, NIPS'12, Curran Associates Inc., USA, 2012, pp. 1097–1105.
URL `http://dl.acm.org/citation.cfm?id=2999134.2999257`

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, D. Hassabis, Mastering the game of go with deep neural networks and tree search, Nature 529 (2016) 484 EP –.
URL http://dx.doi.org/10.1038/nature16961

[5] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (2015) 436 EP –.
URL http://dx.doi.org/10.1038/nature14539

[6] W. Sung, Resiliency of deep neural networks under quantization., CoRR abs/1511.06488.

[7] C. Sequin, R. Clay, Fault tolerance in artificial neural networks, in: 1990 IJCNN International Joint Conference on Neural Networks, 1990, pp. 703–708 vol.1. doi:10.1109/IJCNN.1990.137651.

[8] E. B. Tchernev, R. G. Mulvaney, D. S. Phatak, Investigating the fault tolerance of neural networks, Neural Computation 17 (7) (2005) 1646–1664. doi:10.1162/0899766053723096.

[9] J. Vialatte, F. Leduc-Primeau, A study of deep learning robustness against computation failures, CoRR abs/1704.05396. arXiv:1704.05396.
URL http://arxiv.org/abs/1704.05396

[10] C. Torres-Huitzil, B. Girau, Fault and error tolerance in neural networks: A review, IEEE Access 5 (2017) 17322–17341. doi:10.1109/ACCESS.2017.2742698.

[11] A. Ruospo, A. Bosio, A. Ianne, E. Sanchez, Evaluating convolutional neural networks reliability depending on their data representation, in: 2020 23rd Euromicro Conference on Digital System Design (DSD), 2020, pp. 672–679. doi:10.1109/DSD51259.2020.00109.

[12] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, Y. Huang, Resiliency of automotive object detection networks on gpu architectures, in: 2019 IEEE International Test Conference (ITC), 2019, pp. 1–9. doi:10.1109/ITC44170.2019.9000150.

[13] C. Chen, A. Seff, A. Kornhauser, J. Xiao, Deepdriving: Learning affordance for direct perception in autonomous driving, in: Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV), ICCV '15, IEEE Computer Society, Washington, DC, USA, 2015, pp. 2722–2730. doi:10.1109/ICCV.2015.312.
URL http://dx.doi.org/10.1109/ICCV.2015.312

[14] G. Di Natale, D. Gizopoulos, S. Di Carlo, A. Bosio, R. Canal (Eds.), Cross-Layer Reliability of Computing Systems, Institution of Engineering and Technology, 2020. doi:10.1049/PBCS057E.
URL https://digital-library.theiet.org/content/books/cs/pbcs057e

[15] V. Peluso, A. Cipolletta, A. Calimera, M. Poggi, F. Tosi, F. Aleotti, S. Mattoccia, Monocular depth perception on microcontrollers for edge applications, IEEE Transactions on Circuits and Systems for Video Technology (2021) 1–1doi:10.1109/TCSVT.2021.3077395.

[16] G. Ottavi, A. Garofalo, G. Tagliavini, F. Conti, L. Benini, D. Rossi, A mixed-precision risc-v processor for extreme-edge dnn inference, in: 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2020, pp. 512–517. doi:10.1109/ISVLSI49217.2020.000-5.

[17] L. Ravaglia, M. Rusci, A. Capotondi, F. Conti, L. Pellegrini, V. Lomonaco, D. Maltoni, L. Benini, Memory-latency-accuracy trade-offs for continual learning on a risc-v extreme-edge node, in: 2020 IEEE Workshop on Signal Processing Systems (SiPS), 2020, pp. 1–6. doi:10.1109/SiPS50750.2020.9195220.

[18] B. Moons, R. Uytterhoeven, W. Dehaene, M. Verhelst, 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi, in: 2017 IEEE International Solid-State Circuits Conference (ISSCC), 2017, pp. 246–247. doi:10.1109/ISSCC.2017.7870353.

[19] C. Schorn, A. Guntoro, G. Ascheid, Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators, in: 2018 Design, Automation Test in Europe Conference Exhibition (DATE), 2018, pp. 979–984. doi:10.23919/DATE.2018.8342151.

[20] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition (2014). arXiv:1409.1556.

[21] S. Mittal, A survey of techniques for approximate computing, ACM Comput. Surv. 48 (4) (2016) 62:1–62:33. doi:10.1145/2893356.
URL http://doi.acm.org/10.1145/2893356

[22] M. Courbariaux, Y. Bengio, J.-P. David, Binaryconnect: Training deep neural networks with binary weights during propagations (2015). arXiv:1511.00363.

[23] C. Zhu, S. Han, H. Mao, W. J. Dally, Trained ternary quantization (2016). arXiv:1612.01064.

[24] M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi, Xnor-net: Imagenet classification using binary convolutional neural networks (2016). arXiv:1603.05279.

[25] M. A. Neggaz, I. Alouani, S. Niar, F. Kurdahi, Are CNNs reliable enough for critical applications? an exploratory study, IEEE Design & Test 37 (2) (2020) 76–83. doi:10.1109/mdat.2019.2952336.
URL https://doi.org/10.1109/mdat.2019.2952336

[26] M. A. Neggaz, I. Alouani, P. R. Lorenzo, S. Niar, A reliability study on CNNs for critical embedded systems, in: 2018 IEEE 36th International

Conference on Computer Design (ICCD), IEEE, 2018. doi:10.1109/iccd.2018.00077.
URL https://doi.org/10.1109/iccd.2018.00077

[27] F. dos Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, P. Rech, Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures (2017) 169–176.

[28] Y. Ibrahim, H. Wang, J. Liu, J. Wei, L. Chen, P. Rech, K. Adam, G. Guo, Soft errors in DNN accelerators: A comprehensive review, Microelectronics Reliability 115 (2020) 113969. doi:10.1016/j.microrel.2020.113969.
URL https://doi.org/10.1016/j.microrel.2020.113969

[29] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, Y. Huang, Resiliency of automotive object detection networks on gpu architectures, in: 2019 IEEE International Test Conference (ITC), 2019, pp. 1–9.

[30] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, S. W. Keckler, Understanding error propagation in deep learning neural network (dnn) accelerators and applications, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, New York, NY, USA, 2017, pp. 8:1–8:12. doi:10.1145/3126908.3126964.
URL http://doi.acm.org/10.1145/3126908.3126964

[31] Tiny-cnn (2020).
URL https://github.com/nyanp/tiny-cnn

[32] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, G. Wei, Ares: A framework for quantifying the resilience of deep neural networks, in: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018, pp. 1–6.

[33] D. Palossi, A. Loquercio, F. Conti, E. Flamand, D. Scaramuzza, L. Benini, A 64mw dnn-based visual navigation engine for autonomous nano-drones, IEEE Internet of Things Journal (2019) 1–1doi:10.1109/jiot.2019.2917066.
URL http://dx.doi.org/10.1109/JIOT.2019.2917066

[34] B. Salami, O. Unsal, A. Cristal, On the resilience of rtl nn accelerators: Fault characterization and mitigation (2018). arXiv:1806.09679.

[35] Y. He, P. Balaprakash, Y. Li, Fidelity: Efficient resilience analysis framework for deep learning accelerators, in: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, Athens, Greece, 2020, pp. 270–281. doi:10.1109/MICRO50266.2020.00033.
URL https://doi.org/10.1109/MICRO50266.2020.00033

[36] F. Libano, P. Rech, B. Neuman, J. Leavitt, M. J. Wirthlin, J. S. Brunhaver, How reduced data precision and degree of parallelism impact the reliability of convolutional neural networks on fpgas, IEEE Transactions on Nuclear Science (2021) 1–1(Early Access). doi:10.1109/TNS.2021.3050707.
URL https://doi.org/10.1109/TNS.2021.3050707

[37] M. Sabbagh, C. Gongye, Y. Fei, Y. Wang, Evaluating fault resiliency of compressed deep neural networks, in: 2019 IEEE International Conference on Embedded Software and Systems (ICESS), IEEE, Las Vegas, NV, USA, 2019, pp. 1–7. doi:10.1109/ICESS.2019.8782505.
URL https://doi.org/10.1109/ICESS.2019.8782505

[38] B. Du, S. Azimi, C. de Sio, L. Bozzoli, L. Sterpone, On the reliability of convolutional neural network implementation on SRAM-based FPGA, in: 2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), IEEE, Noordwijk, Netherlands, 2019, pp. 1–6. doi:10.1109/DFT.2019.8875362.
URL https://doi.org/10.1109/DFT.2019.8875362

[39] A. Bosio, P. Bernardi, A. Ruospo, E. Sanchez, A reliability analysis of a deep neural network, in: 2019 IEEE Latin American Test Symposium (LATS), 2019, pp. 1–6.

[40] J. Redmon, Darknet: Open source neural networks in c, http://pjreddie.com/darknet/ (2013–2016).

[41] [Online], Libfixmath library, https://github.com/Petteri-Aimonen/libfixmath (2020).

[42] M. Kooli, F. Kaddachi, G. D. Natale, A. Bosio, Cache- and register-aware system reliability evaluation based on data lifetime analysis, in: 2016 IEEE 34th VLSI Test Symposium (VTS), 2016, pp. 1–6. doi:10.1109/VTS.2016.7477299.

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

[44] Y. LeCun, et al., The mnist database. (2020).
URL https://github.com/ashitani/darknet_mnist

[45] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, Statistical fault injection: Quantified error and confidence, in: 2009 Design, Automation Test in Europe Conference Exhibition, 2009, pp. 502–506. doi:10.1109/DATE.2009.5090716.

[46] J. Redmon, S. K. Divvala, R. B. Girshick, A. Farhadi, You only look once: Unified, real-time object detection, CoRR abs/1506.02640. arXiv:1506.02640.
URL http://arxiv.org/abs/1506.02640