A Reliability Analysis of a Deep Neural Network

[1,2]Alberto Bosio
[1]*INL, Ecole Centrale de Lyon*
Lyon, France
alberto.bosio@ec-lyon.fr

[2]Paolo Bernardi, [2]Annachiara Ruospo, [2]Ernesto Sanchez
[2]*DAUIN, Politecnico di Torino*
Torino, Italy
{paolo.bernardi, annachiara.ruospo, ernesto.sanchez}@polito.it

***Abstract*—Deep Learning, and in particular its implementation using Convolutional Neural Networks (CNNs), is currently one of the most intensively and widely used predictive models for *safety-critical applications* like autonomous driving assistance on pedestrian, objects and structures recognition. Today, ensuring the reliability of these innovations is becoming very important since they involve human lives. One of the peculiarities of the CNNs is the inherent resilience to errors due to the iterative nature of the learning process. In this work we present a methodology to evaluate the impact of permanent faults affecting CNN exploited for automotive applications. Such a characterization is performed through a fault injection enviroment built upon on the *darknet* open source DNN framework. Results are shown about fault injection campaigns where permanent faults are affecting the connection weights in the LeNet and Yolo; the behavior of the corrupted CNN is classified according to the criticality of the introduced deviation.**

***Index Terms*—Deep Learning, Test, Reliability, Fault Injection, Safety, Automotive**

I. Introduction

Deep Learning [1], and in particular Convolutional Neural Networks (CNNs), are currently one of the most intensively and widely used predictive models in the field of machine learning. In most of the cases, CNNs provide very good results for many complex tasks such as object recognition in images/videos, drug discovery, natural language processing up to playing complex games [2]–[4].

One of the peculiar characteristics of the CNNs is the inherent resilience to errors due to their iterative nature and learning process [5]. Thus, these techniques are now deeply used for *safety-critical applications* like autonomous driving [6]. In order to use an electronic device in a safety-critical application, the reliability must be evaluated. More in detail, the probability that a fault may cause a failure is computed. The reliability analysis and its evaluation is regulated by standards depending on the application domain (e.g., IEC 61508 for industrial systems, DO-254 for avionics, ISO 26262 for automotive) [7].

Usually, in-field test solutions have to be embedded and activated in mission-mode to detect possible permanent faults before these may produce any failure. Examples of such test solutions are Design for Testability techniques (e.g., BIST), self-test functional approaches (e.g, Software-based Self-test [8]), or a combination of both. Independently on the adopted test solution, the key point is the achieved fault coverage with respect to the adopted fault model(s). A higher fault coverage leads to ensure a higher level of safety. It is computed as the ratio between the number of faults detected by the test solution over the number of possible faults. The fault list includes all the possible faults except those that cannot produce any failure in the operational mode. Some examples can be found in [9]. In the ISO26262 terminology, these faults are called "Safe Faults Application Dependent" (SFAD). As stated in [7], identifying SFAD is crucial, because it allows to remove them from the fault list and to focus the test efforts towards faults leading to application failures, only.

The goal of this paper is to characterize the impact of permanent faults affecting a CNN by means of a fault injection campaign on the *darknet* open source DNN framework [10].

In the literature few works targeted the reliability of DNN, but only for soft errors (i.e., bit flip). In [11], the authors evaluated the reliability of one CNN executed on three different GPU architectures (Kepler, Maxwell, and Pascal). The soft errors injection have been done by exposing the GPUs running the CNN under controlled neutron beams.

In [12], the authors presented a more detailed analysis. They characterized the propagation of soft errors from the hardware to the application software of different CNNs. The injections were performed by using a DNN simulator based on open-source DNN simulator framework, Tiny-CNN [13]. Thanks to the flexibility of the simulator, it was possible to characterize each layer to have a more precise analysis.

In our study, we consider the injection of permanent faults with the final goal of evaluating the SFAD distribution and to classify the criticality of a CNN prediction deviation according to the corrupted layer. The main contributions of this manuscript with respect to the state-of-the-art are listed in the following:

- Fault Injection is carried at software layer, in order to be independent from any potential HW architecture finally running the CNN;
- Fault effects are classified in SFAD (masked), Safe and Unsafe, according to several criteria
- Fault Injection allows to identify the most sensible layers for which a safety mechanism may be purposely devised.

Two different CNN topologies have been characterized using the general darknet framework, the LeNet and Yolo CNNs.

978-1-7281-1756-0/19/$31.00 ©2019 IEEE

The experimental results show limited computational costs to achieve a good accuracy in the faulty behavior classification.

The rest of the paper is organized as follows. Section II presents the background about the two case studies and Section III details the Fault Injection environment. Experimental results are presented in Section IV while Section V concludes the paper.

II. Background

This section presents the case studies used in this work. As previously stated, we exploit the *darknet* open source DNN framework [10] implemented in C language. Darknet, as most CNNs, is composed of three types of layer: Convolution layer, Max pooling layers and Fully connected layer (for classification). Two CNNs are used as case studies. The first one is LeNeT [14] used as classifier for handwritten digit recognition task (we used the MNSIT database of training and validation). The second one, named YOLO [15], is a Deep Convolutional Neural framework capable of detecting objects in real time, analyzing up to 45 frames per second. YOLO is a predictor CNN exploited to detect a certain set of objects (i.e., again the list of recognizable objects came from [10]). In Figure 1 the prediction results are shown highlighting the 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.

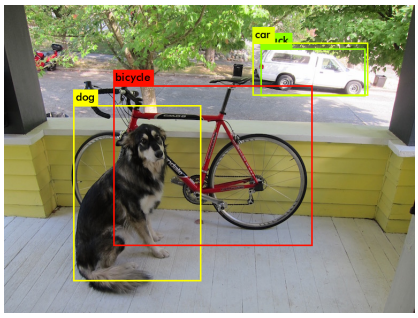


Fig. 1: Yolo CNN predition result

Details about the networks topologies are given in Table I and II. For both tables, the number layer is reported in the first column. The layer type is specified in the second column as Convolution (*Conv*) or Fully Connected (*FC*). Then, the third column details the number of connections for each layer; while the last column indicates the bit width used for storing each weight. As stated in the table, all the weights are represented as 32-bit floating point numbers.

III. Fault Injection

This section provides the details of a Fault Injection framework built on the Darknet framework. This is a generic environment that is used as a generic engine to run several types of neural networks. In our case, the results will be reported about the cases of study introduced in the background

TABLE I: LeNet Characteristics

Layer	Type	Connections	Width
0	Conv	2400	32
1	Conv	51200	32
2	FC	3211264	32
3	FC	10240	32

TABLE II: Tiny YOLO Characteristics

Layer	Type	Connections	Width
0	Conv	432	32
1	Conv	4608	32
2	Conv	73728	32
3	Conv	294912	32
4	Conv	1179648	32
5	Conv	4718592	32
6	Conv	262144	32
7	Conv	1179648	32
8	Conv	130560	32
9	Conv	32768	32
10	FC	884736	32
11	FC	65280	32

paragraph. Generally speaking, the hardware system can be affected by faults caused by physical manufacturing defects. Faults propagate through the different hardware structures composing the full system (see Fig. 2). However, faults can be masked during this propagation either at the technological or at architectural level [16]. When a fault reaches the software 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 hang. The software stack can play an important role in masking errors; at the same time, this phenomena is implicitly important for the system reliability but a hard challenge for test engineers that have to ensure safeness of their systems.

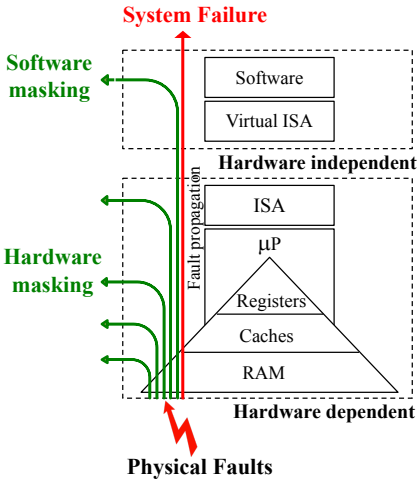


Fig. 2: System Layers and Fault Propagation.

As stated in the introduction, the goal of this paper is to investigate the effect of permanent faults at software layer

in order to be independent from the hardware architecture running the CNN (i.e., CPU, GPU or HW accelerator). As permanent fault, we consider the Stuck-at Fault (SaF) model at 0/1 ($SaF0$ and $SaF1$). The Fault Location (FLo) is defined by (1).

$$FLo =< Layer, Connection, Bit, Polarity > \quad (1)$$

where $Layer$ corresponds to the CNN layer, $Connection$ is the the edge connecting one node of the $Layer$ and Bit is one the bits of the weight associated to the $Connection$. Finally, the $Polarity$ can be '0' or '1' depending on the SaF.

The Fault Injector actually works at software layer, and its pseudo-code is provided in the $Pseudo - Code$ (1). It corresponds to a simple serial fault injector that modifies the CNN topology as described by (1).

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

Listing 1: Fault Injection Pseudo-Code

The fault injection process consists in the following: Once the CNN is fully trained, a golden run is performed collecting the golden results (aka golden_prediction), line 1 in 1. Then, the actual fault injection process is performed. The initial step requires to generate the list of faults to be injected. This fault list should be seen as a list of places where to inject the faults as described previously. Then, for any fault in the fault list (line 2 in 1), a prediction run is performed and the results collected and named as faulty_prediction. At this point (line 5 in 1), the obtained results are compared with the expected ones, and the results logged for a later analysis.

In details, the function *compare* of the Pseudo-code (1) classifies the prediction/classification of the faulty CNN w.r.t. the golden one. The classification is done as follows:

- **Masked:** no difference is observed from the faulty CNN and the golden one.
- **Observed:** a difference is observed from the faulty CNN and the golden one. Depending on how much the results diverge, we further classify these as:
  - **Safe:** the confidence score of the top ranked element varies by less than +/-5% w.r.t. the golden one;
  - **Unsafe:** the confidence score of the top ranked element varies by more than +/-5% w.r.t. the golden one, or the top ranked element predicted by the faulty CNN is different from that predicted by the golden one. As already discussed in [12] this is the most critical observed fault;

As reported in the introduction, the goal of this paper is to identify the "Safe Faults Application Dependent" (SFAD) accordingly to the ISO 26262 standard. SFAD faults can be simply computed by the union between Masked and Safe-Observed fault as depicted in (2).

$$SFAD = Masked \cup Safe\_Observed\_Fault \quad (2)$$

IV. Experimental Results

This section reports the gathered experimental results. Whereas, the next subsections will present the results for both the CNNs. Finally, the last subsection discuss about the obtained results.

TABLE III: LeNet Fault List

Layer	Connections	Width	#Faults	#Injections
0	2400	32	153600	9039
1	51200	32	3276800	9576
2	3211264	32	205520896	9604
3	10240	32	655360	9465

TABLE IV: Tiny YOLO Fault List

Layer	Connections	Width	#Faults	#Injections
0	432	32	27648	7128
1	4608	32	294912	9301
2	73728	32	4718592	9584
3	294912	32	18874368	9599
4	1179648	32	75497472	9603
5	4718592	32	301989888	9604
6	262144	32	16777216	9599
7	1179648	32	75497472	9603
8	130560	32	8355840	9593
9	32768	32	2097152	9560
10	884736	32	56623104	9602
11	65280	32	4177920	9582

Tables III and IV present the fault list size of each layer of the two CNN topologies. The number of possible faults is simply the multiplication between the connections number (column 2) and the weight size times 2 (i.e., stuck-at-0 and stuck-at-1). As shown in column 4, the overall number is very high reflected in a non-manageable fault injection campaign execution time. Thus, in order to reduce the fault injection execution time, we randomly select a subset of faults. To obtain statistically significant results with an error margin of 1% and a confidence level of 95%, and average of 9K fault injections have to be considered. The precise numbers are given in the last column of tables III and IV and they were calculated by using the approach presented in [17].

A. LeNet

For LeNet we used the pre-trained weights available from [18]. For the injection campaign, we randomly selected 37 validation images from the MNIST database. Figures 4, 5 and 6 depict the obtained results in terms of percentage of Masked, Safe Observed and Unsafe Observed. For each chart, we differentiate the injections per layer (from 0 to 3). As it can be seen, the percentage of Masked faults is higher for layer 2 and 3 (i.e., the fully connected layers), while the first two layers have shown a lower percentage of Masked faults. A similar analysis can be done for Unsafe Observed faults. Fully connected layers show the lowest percentage of Unsafe

Observed faults. This means that the less "critical" layers are the fully connected ones (for the LeNet topology).

Concerning the Unsafe Observed faults (Figure 6), we would like to stress the fact that their percentage is very low, varying from 0.4% up to 1.8%. Moreover, when considering "Layer 0", the two drops of Unsafe Observed faults percentage (corresponding to workload 17 and 23) only represent a difference of 1.4% meaning that we can consider this variation as negligible. This means that we cannot notice a particular dependency on the input workload.

This result is quite interesting because it is showing a different trend with respect to the effect of soft errors. Indeed, convolutional layers (i.e., the first two layers for LeNet) are supposed to be the more resilient to the presence of a fault, according to results shown in [12]. This is due to the fact that their role is to extract the features from the source image, while the full connected layers are supposed to be the less resilient because they classify the features extracted by the first two levels. On the other hand these results seem to confirm the conclusion of [11] in which the authors claim this trend. However, in [12] the result presented another trend: a slightly higher resilience was found for the convolutional layer. This can be explained by two factors: first of all the fault model, here we experimented permanent faults while in [12] the fault model was transient, and the network topology.

The last part of the experiments carried out on the LeNet is the analysis of the most critical bits of the variable storing the weights. As previously described, the weights are represented as single-precision binary floating-point format as described by the IEEE 754 standard as depicted in the figure 3.

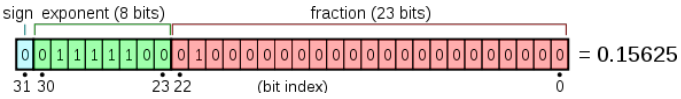


Fig. 3: IEEE 754 example.

As expected, all the Unsafe Observed faults have been due to faults affecting the 8 bits used for storing the exponent (i.e., from bit 30 down to bit 23). The sign and the mantissa bits do not have significant impacts (i.e., they led either to Masked or Safe Observed faults).

B. YOLO

For YOLO we used the pre-trained weights available from [10] and more in detail we use the *yolov3-tiny.weights*. The workloads were also downloaded from [10] and consist of seven different images. For these experiments, we keep the same fault classification as for LeNet, with only one exception. LeNet classifies the input image as one precise digit. Thus only the highest top rank output were considered. Using YOLO, we may have more than one object that can be detected on the image, for example we may have several cars present in the image. Therefore, we have to updated the Unsafe Observed faults definition as follows:

1) The number of detected objects is different from the golden and faulty prediction;

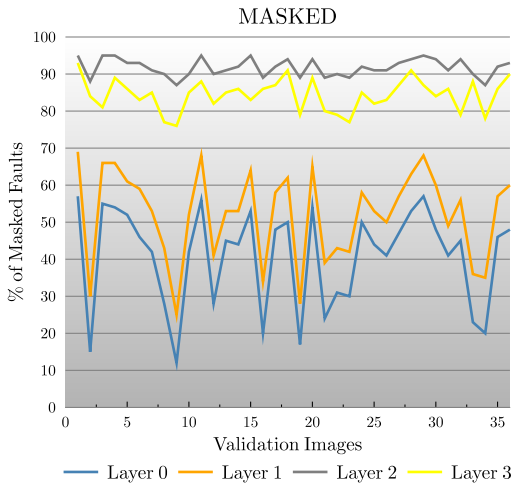


Fig. 4: LeNet Masked faults.

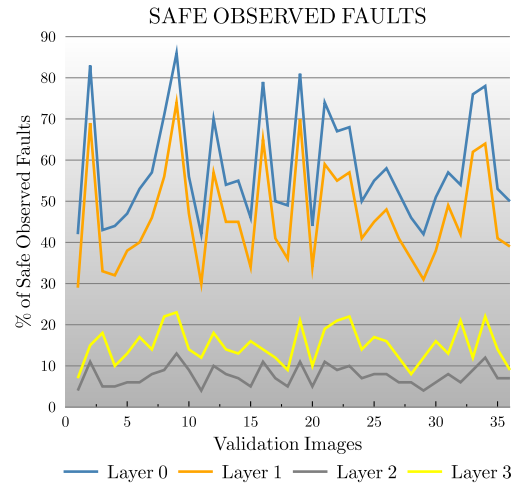


Fig. 5: LeNet Safe Observed faults.

2) The number of detected objects is equal, but the label associated with are different (i.e., wrong detection);
3) The "location" of the top ranked element varies by more than +/-5% w.r.t. the golden one.

The other definitions (Masked and Safe) do not change.

To better clarify this definition, let us resort to and example depicted in figure 10 and compare with the golden prediction shown in figure 1.

In the case of Safe Observed faults. Indeed, the faulty YOLO is able to correctly detect the four objects, but their "location" is slightly different (less than 5%) w.r.t. the golden one. The term "location" means the rectangle identifying the object in the picture.

Conversely, Figure 10b and figure 10c present cases of Unsafe Observed faults. In these pictures we can note that the CNN only recognizes two objects (the bike and the car)

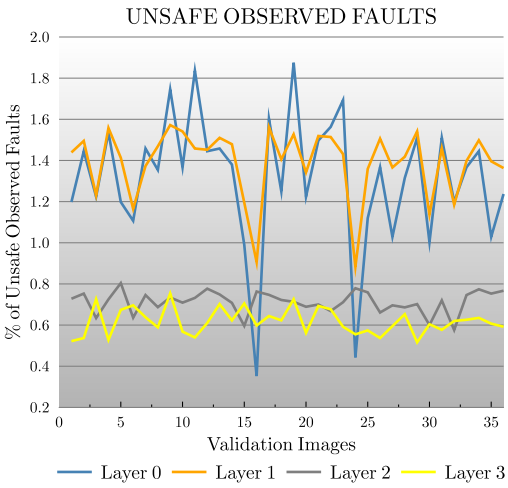


Fig. 6: LeNet Unsafe Observed faults.

and the prediction is wrong in general terms.

Figures 7 and 8 depict the obtained results in terms of percentage of Masked and Unsafe Observed faults. Contrarily to the LeNet experiments, we do not have a significant percentage of Safe Observed faults (i.e., the obtained percentage was less than 0.05% in average). As it can be observed, we have a different population of Masked faults compared to the LeNet data. First of all, the distribution of Masked faults do not vary so much among the layers, except for the first one. We cannot thus claim that full connected layers are the less "critical" since also the most of convolutional layers shown high percentage of Masked faults. On the other hand it is also interesting to note that depending on the workload (i.e., the input image as plotted on the X-axis), the distribution of Masked faults significantly changes, while for LeNet it was not the case.

A final comment is related to the image number 4 for which we can notice a peak of Unsafe Observed faults for all layers, but especially for the "Layer 0". As for LeNet results, "Layer 0" is the most sensible layer to the input workload. However, in this case, the variation can not be neglected since we observed a increase of 10% of Unsafe Observed faults. We thus deeply investigated the input workload number 4. Figure 9 depicts an example of Unsafe Observed faults, where instead of detecting the horses, the faulty network detect one sheep. First of all, the input image is "complex" since there are many horses to detect. For this specific image, faults seems to lead to more Unsafe Observed faults for "Layer 0". This means that depending on the network topology, the input workload can play a significant role.

Once again, this result is interesting in the sense that clearly proves that it is not possible to generalize the results. It means that each CNN has to be analyzed in order to identify the most critical elements (i.e., layers). Despite of the difference between the results obtained by the two networks, the analysis of the most critical bit in the weights confirm the LeNet results since we obtained again that all the Unsafe Observed faults are due to faults affecting the 8 bits used for storing the exponent (i.e., from bit 30 down to bit 23). The sign and the mantissa bits do not have significant impacts (i.e., they led either to Masked or Safe Observed faults).

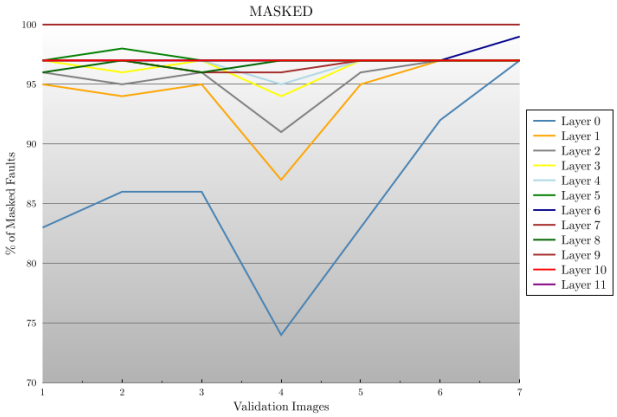


Fig. 7: YOLO Masked faults.

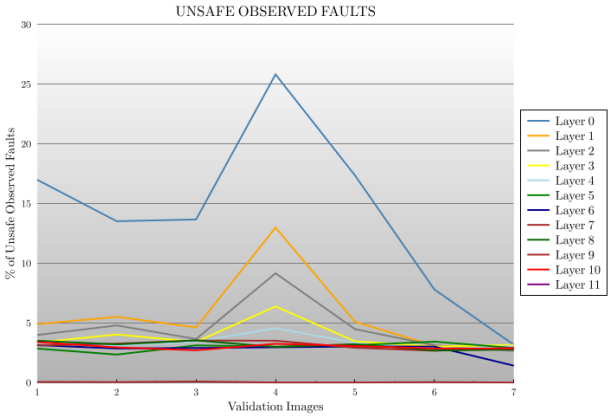


Fig. 8: YOLO Unsafe Observed faults.

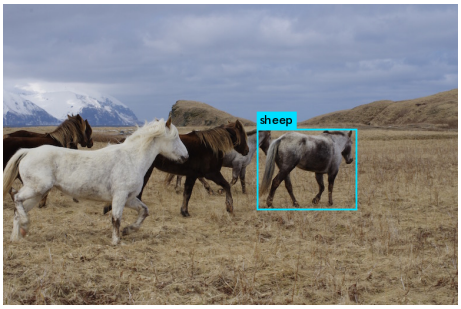
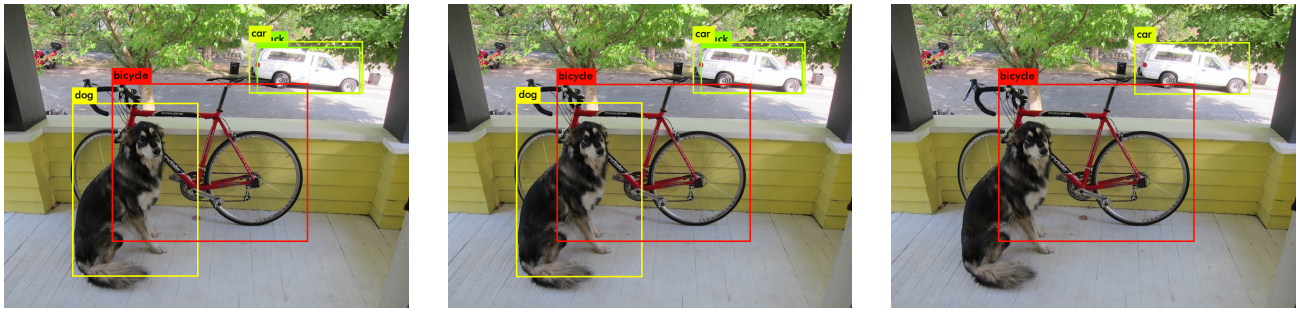


Fig. 9: YOLO Workload #4.

C. Discussion

According to the presented results, the general conclusion about the criticality of the analyzed CNNs is related to the bits of the variables representing the weights. More in particular,



(a) Golden Prediction

(b) Safe Observed faults Prediction - missing recognition

(c) Unsafe Observed faults Prediction - strong corruption of prediction result

Fig. 10: YoLo Predictions example.

the analysis shown that we have to take care only of 8 bits (i.e., the exponent bits from bit 30 down to bit 23) that cause critical behaviors. This could be one solution were the redundancy is applied only on the critical part of the system. From the test point of view, the conclusion can be the same, in the sense that our test solution has to be able to mainly test these particular bits only. In this paper, the description of the test solution as well as the redundancy techniques used for hardening the CNN is not presented. Indeed, our analysis has the advantage to be hardware independent so that the designer will have to opportunity to carefully select the HW architecture by carefully taking into account the most critical bits. Consequently, the test engineer will be able to focus the test effort only on that critical parts in order to reduce the cost of the test solution.

V. Conclusions

In this paper we presented a characterization framework for analyzing the impact of permanent faults affecting CNN intended to be used in automotive application. The characterization was done by means of fault injection campaign on the *darknet* open source DNN framework [10]. The experiments were performed at software level with the aim of being independent on the HW architecture and to derive a common characterization of the behavior of these networks in the presence of these faults. We believe that this is an important outcome since the designer, starting from these results, could be able to select the most convenient HW architecture for a given CNN.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436 EP –, 05 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14539

[2] *Recent Advances in Deep Learning for Speech Research at Microsoft*. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), May 2013. [Online]. Available: https://www.microsoft.com/en-us/research/publication/recent-advances-in-deep-learning-for-speech-research-at-microsoft/

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 1097–1105. [Online]. Available: 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, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484 EP –, 01 2016. [Online]. Available: http://dx.doi.org/10.1038/nature16961

[5] W. Sung, "Resiliency of deep neural networks under quantization." *CoRR*, vol. abs/1511.06488, 2015.

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

[7] R. Cantoro, A. Firrincieli, D. Piumatti, M. Restifo, E. Sánchez, and M. S. Reorda, "About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications," *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pp. 1–6, 2018.

[8] M. Psarakis, D. Gizopoulos, E. Sánchez, and M. S. Reorda, "Microprocessor software-based self-testing," *IEEE Design and Test of Computers*, vol. 27, pp. 4–19, 2010.

[9] P. Bernardi, M. Bonazza, E. Sanchez, M. S. Reorda, and O. Ballan, "On-line functionally untestable fault identification in embedded processor cores," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '13. San Jose, CA, USA: EDA Consortium, 2013, pp. 1462–1467. [Online]. Available: http://dl.acm.org/citation.cfm?id=2485288.2485636

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

[11] F. dos Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech, "Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures," pp. 169–176, 06 2017.

[12] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and 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*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 8:1–8:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126964

[13] Tiny-cnn. [Online]. Available: https://github.com/nyanp/tiny-cnn

[14] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[15] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

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

[17] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, April 2009, pp. 502–506.

[18] [Online]. Available: https://github.com/ashitani/darknet_mnist