Fault injections and their impact on Image classification DNN's

Adam Ghadban[†1]

[1] Department of Computer Science – Technion, Haifa, Israel
[†] equal contribution

Abstract. In the current era characterized by the ubiquitous integration of machine learning algorithms across a variety of sectors, the question of functional safety (FuSa) is gaining significant prominence. This is particularly relevant in scenarios such as autonomous vehicles and Internet of Things (IoT) devices, wherein the accuracy of machine learning models can directly influence the safety of the end-users. Owing to the common deployment of these models on Application-Specific Integrated Circuit (ASIC) architectures like Google's Tensor Processing Unit (TPU), it is imperative to evaluate their robustness against potential faults. This research project concentrates on the possible deterioration of Convolutional Neural Networks (CNN) performance, including models based on ResNet18 architecture, due to the injection of faults into the data-path and memory of these networks. Our investigations were grounded on a CNN model trained using the CIFAR-10 dataset, enabling us to delve into the classification errors prompted by diverse fault models. We scrutinized faults injected into various components of the CNN architecture: the batch normalization layers for bias-based faults, the weights in the convolutional layers, and the inputs of the convolutional layers. Furthermore, we examined fault models based on multiple parameters such as bit-granularity, the quantity of faults, and the distribution of faults across targets. This empirical approach allowed us to garner a comprehensive understanding of the impact of these factors on the functional safety of machine learning models. Ultimately, our findings contribute to the broader discourse on ensuring the reliability and safety of machine learning applications in real-world scenarios. Source code of fault injector can be found at https://github.com/Adam11072000/236509_project

Keywords: FuSa violations · Fault injection · Data-path faults · Memory faults.

1 Introduction

Fault injections. Fault injection (FI) is a class of physical attacks usually used to extract cryptographic keys from security systems. It is a powerful technique used to evaluate the reliability, availability, and fault-tolerance of computer systems, both hardware and software. It involves the intentional introduction of faults or errors into a system to assess how it behaves under such circumstances. Fault

injection can be categorized into hardware and software-based techniques, each offering its own advantages for specific use cases [1].

Recently, fault injection has gained interest in the field of machine learning, particularly to investigate the robustness and fault-tolerance of neural networks [2]. By injecting faults into the weights, biases, or activations of a neural network, researchers can study the impact of such faults on the model's performance and make necessary adjustments to improve its reliability [3].

Faults can be divided into two types: Permanent faults, where the fault usually burns a transistor, thus getting stuck at either 1 or 0, and transient faults, which are faults that can be changed back, either occurring in memory where one needs to change back the bit, or a corruption that will subside in time. Faults that are life-like, e.g. electromagnetic waves are usually distributed in a Gaussian manner, thus faults are usually aggregated around a certain neuron, a certain weight or a certain layer. On the other hand, adversaries who have the budget can actually target with a lot more granularity than life-like faults or low budget faults, thus it can be distributed uniformly too, or being deterministic.

Fault injection techniques on DNN's In the present work, we expand upon [4] and investigate the robustness of CNN based model architectures, ResNet18, against FI attacks and its relation to FuSa violations. The present work was done with fault injection simulation via software, due to the project's limited timeline. Our assumption that a fault into a model parameter or an input image can be translated to a fault model across a systolic array. The training of ResNet18 was done in float32 and the network was checked for robustness in float32 datatype. The assumption in this paper is that our model is a white-box, i.e. the adversary knows the architecture of the network. Below, we outline our contributions.

Firstly, we have developed a generic fault injector into a wide-range of networks, mainly taken from robustbench project, for the list of supported architectures, check https://github.com/RobustBench/robustbench. The simulator supposes that these networks have been trained on CIFAR-10 dataset. The simulator has a very wide range of fault injection models, and can be controlled via CLI.

Secondly, we expand the fault models to be more life-like, meaning less strict models, and we allow for multiple adjacent bit-flips, randomized faults across a certain fault target, choosing a target layer randomly and Uniform or Gaussian distribution of faults. These parameters are usually pre-chosen by natural causes for natural attacks such as electromagnetic faults (Gaussian, multiple adjacent bit-flips).

Lastly, we aggregated empirical data of these fault models and check the accuracy drop of the model in each fault model, analyize the data and reach an empirical decision as to which components in the fault model affect the network's FuSa, and how do each component affect the network's FuSa.

The rest of the paper is organized as follows: Section 2 describes related work, Section 3 describes our fault models and our CLI, Section 4 provides the experimental results of our project and Section 5 concludes the paper.

2 Related work

This particular topic have become prominent in the last few years. Multiple fault models and experiments have been proposed to assess the robustness of DNN models. in [4], authors presented fault models into a 3-layer Multi Layer Perceptron (MLP) on a systolic array, fault model included bit-wise vulnerability analysis, which introduced a fault into a single register in a Processing Element (PE) in the systolic array, which included bit-flipts from MSB to 14th bit and Unit-wise vulnerability analysis which included injecting faults into the MSB of multiple PE's. in [5], authors proposed to attack biases of different layers across the network, with the assumption that the attacker can target a single bit granularity. in [2], authors proposed injecting faults into weights in the last hidden layer of an MLP, which included a single neuron fault or multiple-neuron fault.

3 Method

In the subsequent sections, we first establish a clear understanding of the concept of a fault model within our specific context and describe the various fault models considered for our study. Subsequently, we delve into the methodology behind the fault injection (FI) simulator, detailing its mechanism for fault introduction. We further elaborate the protocols for data aggregation. Finally, we touch upon the efficiency and speed metrics of the simulator, highlighting its computational proficiency.

3.1 Fault Models

Definition. In our study, a "fault model" is defined as a structured representation, encompassing various parameters that elucidate the manner and location of the fault's impact on our target Deep Neural Network (DNN) model. the set of parameters used are:

1. number_of_faults: This quantitative parameter details the count of faults we aim to introduce into the selected target.
2. num_bits_to_flip: Denoting the quantum of bits the researcher intends to invert for each fault, this parameter can be set to an integer value.
3. fault_distribution: This defines the pattern in which faults are scattered across the selected target. The distribution can adhere to either a Uniform or Gaussian model.
4. fault_target: This parameter indicates the specific component of the DNN that will be subjected to the fault. It can take on values such as bias, weight, or memory—with "memory" signifying the targeting of the input of a given layer.
5. targe_bits_to_flip: This parameter lists specific bits intended for inversion. For instance, the sequence of bits 1, 2, 5, 9, and 23 would be represented as [1, 2, 5, 9, 23].

6. fault_distribution_per_target: This parameter indicates the distribution of faults across an element of the target. This parameter can take either Gaussian or Uniform, and does not have a meaning when targe_bits_to_flip is set.
7. target_layer: Represents the specific layer within the DNN model that we aim to compromise, this parameter is indicative of our focus area for the fault injection.

It is noteworthy that the predominant distribution for fault indices on the target is Gaussian. We only diverge to a uniform distribution in unique scenarios where our objective is to discern the repercussions of variations in fault distribution.

Proposed fault models. In our research, we introduced a series of fault models, with each model uniquely designed to explore the implications of individual fault model parameters on the cascading of errors to the network's output and its subsequent Functional Safety (FuSa) violations. For each of these fault models, we strategically selected four distinct layers within the network for fault injection, each serving a specific functional role:

1. Initial Layer: Positioned at the network's onset, this is primarily a rudimentary feature extraction layer, laying the foundation for more intricate analyses in subsequent layers.
2. Mid-Network Start Layer: Positioned at the start of the middle section of the DNN. This layer is an intermediate feature extraction layer in terms of complexity.
3. Mid-Network End Layer: Positioned towards the culmination of the middle segment of the network, this layer continues with intricate feature extraction. Given its placement, it often zeroes in on a different set of features compared to the preceding intermediate layer.
4. Terminal Layer: Anchored at the network's finale, this is a complex feature extraction layer, drawing from the insights gathered by preceding layers.

It's crucial to note that our experiments were grounded on the ResNet18 model from the PyTorch library. Consequently, bias as a target exists exclusively within the BatchNorm layers. Furthermore, since we have 3 different target types (memory/bias/weights) and 4 different layers for each type, each fault model resulted in 12 specific fault models.

1. Bit Granularity This set of fault models focuses on the effect of bit-flips placement on FuSa violations, as such, we proposed to check the effect of faults on mantissa, exponent and sign bit. We have proposed faults on the LSB section of the mantissa (0-5), middle section of the mantissa (10-14) and MSB section of the mantissa (19-23). As for the exponent, we proposed to flip bits (24-25) for the LSB section, (27-28) for the middle section and (29-30) for MSB section. As for the sign bit, bit number 31. The rest of the parameters of the fault model were as follows:

1. number_of_faults = 20.
2. num_bits_to_flip = 6/5/5/2/1, depending on the model.
3. fault_distribution = Gaussian.
4. target_layer = as described above

2. Distribution of faults This set of fault models focuses on the effect of the distribution of faults across target on FuSa violations, thus each fault model was run with Gaussian and Uniform distribution. The rest of the parameters of the fault model were as follows:

1. number_of_faults = 20.
2. num_bits_to_flip = 5.
3. fault_distribution = Gaussian/Uniform.
4. target_layer = as described above.
5. target_bits = [19, 20, 21, 22, 23].

3. Distribution of bit-flips in a fault This set of fault models focuses on the distribution of randomized bit-flips across a single fault. The available distributions were Gaussian/Uniform. The rest of the parameters of the fault model were as follows:

1. number_of_faults = 20.
2. num_bits_to_flip = 5.
3. fault_distribution_per_target = Uniform/Gaussian.
4. fault_distribution = Gaussian/Uniform.
5. target_layer = as described above.

Note that we did not provide target_bits, thus they are randomized according to the distribution.

4. Fault number This set fault models focuses on the effect of number of faults where number of faults ranged from 1 to 500. The rest of the parameters of the fault model were as follows:

1. num_bits_to_flip = 5.
2. target_bits = [19, 20, 21, 22, 23].
3. fault_distribution = Gaussian.
4. target_layer = as described above.

3.2 FI Simulator

The fault injection simulator, designed primarily for in-depth neural network analysis, operates via a Command Line Interface (CLI). It accommodates a range of parameters, encompassing the neural network structure, fault targets (memory, weight, or bias), the specific number of bits intended for flipping, the bit-flip distribution characteristics, the spatial distribution of the faults across

targets, and the overall number of faults to be injected. An inherent feature of this simulator is its ability to dynamically parse layer names from the provided neural network model. This functionality allows for more precise and interactive selection of specific layers to undergo fault injection.

Upon initialization, the simulator identifies the required layer and fault target based on user-defined specifications. It subsequently generates an index mask, aligning with the target's dimensional attributes and reflecting the designated fault distribution. Concurrently, a corresponding bit mask, derived from the fault model, is generated. This mask is then applied, using a bitwise 'AND' operation, to the elements of the previously defined index mask.

For fault implementations on weight or bias targets, the simulator overlays the created mask on the chosen fault target and captures the modified network state. When memory is designated as the fault target, the simulator swaps the target layer with a modified version. During the forward pass of this modified layer, the fault mask modifies the input data.

The modified network undergoes evaluation over 100 epochs using a subset of 50 images from the CIFAR-10 dataset. The accuracy metrics of each epoch are recorded. At the conclusion of the process, this data, alongside the pertinent fault model details, is consolidated and stored in a JSON file. The choice of 100 epochs ensures a more robust statistical analysis, minimizing variance introduced by the randomized fault distribution.

4 Experiments

In this section, we outline the empirical findings of our study. For every fault model discussed, there will be one or more figures. Each figure contains multiple subplots, and every subplot features four distinct labels, corresponding to the layers selected for that specific fault. The blue line indicates an **Initial Layer**, orange indicates a **Mid-Network Start Layer**, green indicates a **Mid-Network End Layer**, and red indicates a a **Terminal Layer**.

Y axis represents the accuracy of the model while X axis represents the fault index number. The faults are sorted by either fault_number or num_bits_to_flip depending on the fault model. Note that a drop of 2% accuracy is a FuSa violation since we used 50 samples.

4.1 Distribution of bit flips in a fault and FuSa violations

The accuracy differs between fault target, which can be attributed to multiple factors:

1. A DNN is usually robust to perturbations in the images due to the diversity in the training set, thus injecting faults into memory targets resulted in 1 FuSa violation in average, as opposed to weights and biases, which induces 3 to 4 FuSa violations.

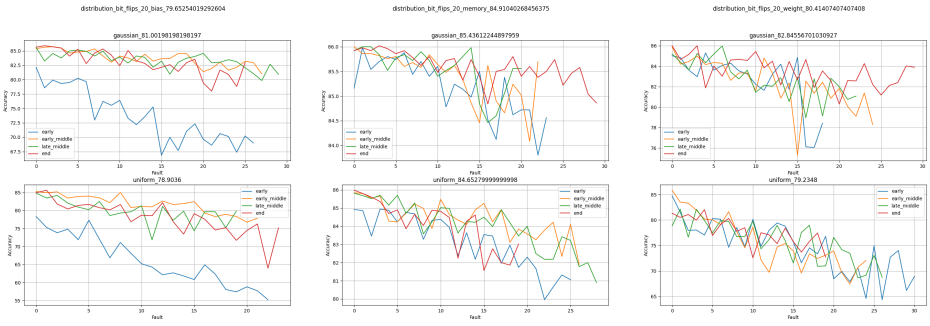
2. Bias faults will produce different behaviour due to activation functions like ReLU, thus a change in bias can potentially affect the output of the neuron drastically.
3. Weight faults can alter the relationship between neurons, thus altering the feature extraction process.

We can see that accuracy is much affected by target layer location in bias target, although not much by weight and memory, which could be attributed to the fact that activation functions like ReLU are very sensitive to bias faults, thus affecting the feature extraction process.

Intuitively, if the simple features were damaged due to the faults, the network will not infer the more complex features correctly, thus will result in more FuSa violations. As such, we can see that by average, faults in the simple feature extraction layer are the most dominant

As for the distribution, Uniform distribution resulted in a lower accuracy as opposed to a Gaussian distribution. This phenomenon can be attributed to the fact that in Gaussian, bit-flips are aggregated around the center of the float32 element, thus middle mantissa bits, thus not changing the value of the target drastically, as opposed to Uniform, where bit-flips are randomly selected across all the bits, which could result in an exponent bit-flip, thus a drastic change of the value.

The faults are sorted from left to right by the number of bits flipped, thus we can see that the graphs are monotonically non-increasing functions. We can see the results of this fault model below.



(a) Bias target, with an average accuracy of 79.65. Average accuracy for Gaussian distribution is 81, and 78.9 for Uniform distribution.

(b) Memory target, with an average accuracy of 84.91. Average accuracy for Gaussian distribution is 85.43, and 84.65 for Uniform distribution.

(c) Weights target, with an average accuracy of 80.41. Average accuracy for Gaussian distribution is 82.84, and 79.2348 for Uniform distribution.

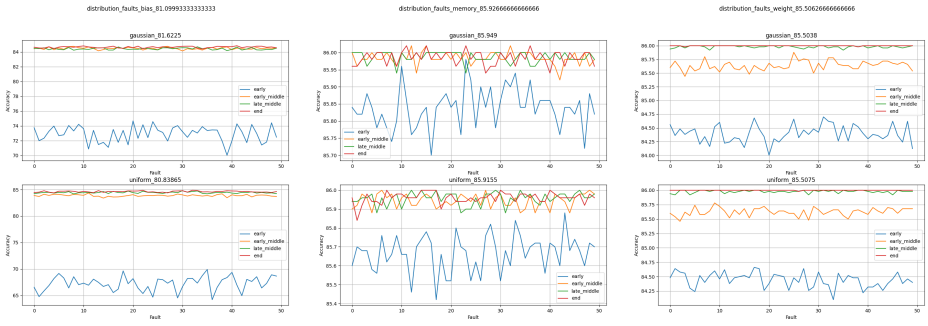
Fig. 1: Here we present the effect of distribution of bit-flips on the accuracy of the model.

4.2 Distribution of faults across target and FuSa violations

The most prominent attribute of these graphs is that faults are much more dominant in the early layers of the network. This could be again attributed to the fact that damaging simple feature extraction will have a much more global error propagation as opposed to other layers. Here we also see a lower accuracy in the start layer of the middle section as opposed to the last layers, thus also damaging an intermediate feature extraction can damage the accuracy of the model, thus inducing FuSa violations.

In average, we do not see a big difference between Gaussian and Uniform distributed faults, although in bias target, we can see that the accuracy of the model on the **Initial Layer** was much lower in Uniform as opposed to Gaussian, which can be attributed to bias being a global value, thus uniform faults across bias can damage more neurons globally as opposed to Gaussian, which will be aggregated around the middle neuron, i.e., affecting less features extraction process. In the case of the **Initial Layer**, we got 8-10 FuSa violations in Uniform as opposed to 6-8 FuSa violations in Gaussian.

Furthermore, we got 2-3 FuSa violations in average when target is bias as opposed to 0-1 FuSa violations when target is memory or weights, which could again be attributed to the fact that activation functions like ReLU are sensitive to bias faults.



(a) Bias target, with an average accuracy of 81.09 Average accuracy for Gaussian distribution is 81.6255, and 80.83 for Uniform distribution.

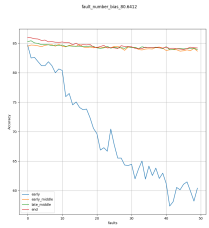
(b) Memory target, with an average accuracy of 85.92. Average accuracy for Gaussian distribution is 85.94, and 85.91 for Uniform distribution.

(c) Weights target, with an average accuracy of 85.5. Average accuracy for Gaussian distribution is 85.5, and 85.5 for Uniform distribution.

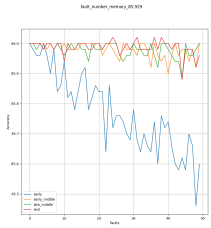
Fig. 2: Here we present the effect of distribution of faults across target on the accuracy of the model.

4.3 Number of faults and FuSa violations

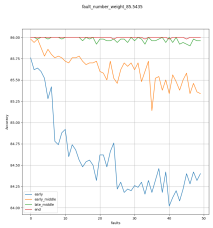
The most prominent attribute of this fault model is the clear monotonous decreasing functions across all layers, since the fault models are sorted via number of faults, which indicates an intuitive correlation between number of faults and FuSa violations. Moreover, we can see that bias target is more prone to produce FuSa violations than other targets which can be attributed to the same reasons as before. We can also see the correlation between target layer's location in the DNN and FuSa violations, which can be attributed to the same reasons as before.



(a) Bias target, with an average accuracy of 80.64.



(b) Memory target, with an average accuracy of 85.92.

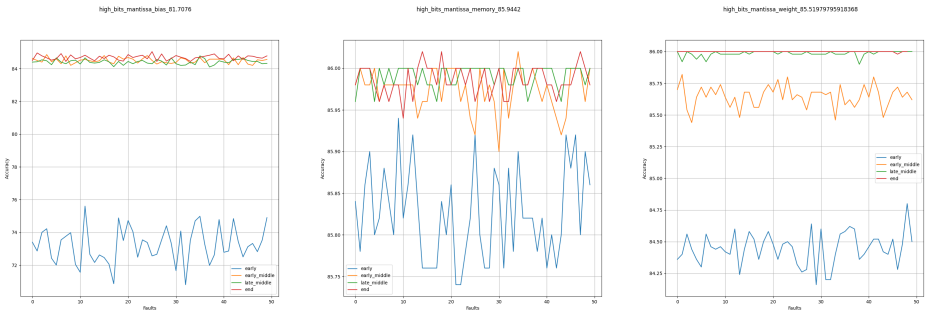


(c) Weights target, with an average accuracy of 85.54.

Fig. 3: Here we present the effect of number of faults across target on the accuracy of the model.

4.4 Higher section bit-flips in mantissa and FuSa violations

Bias accuracy can be attributed to the same reasons as before, as well as target layer. It is worth noting that we got 5-7 FuSa violations when injecting faults into the bias of the **Initial Layer**.



(a) Bias target, with an average accuracy of 81.707.

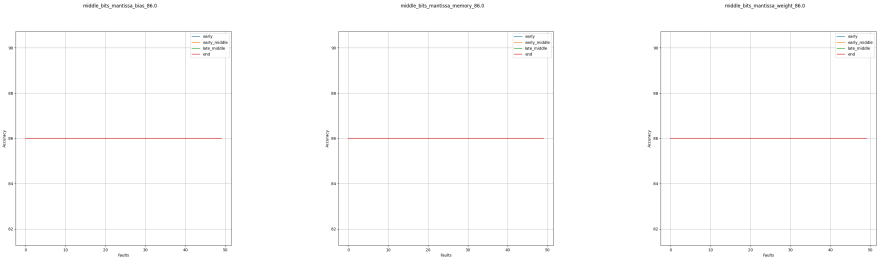
(b) Memory target, with an average accuracy of 85.94.

(c) Weights target, with an average accuracy of 85.51

Fig. 4: Here we present the effect of higher section bit-flips in mantissa and the effect on model accuracy.

4.5 Middle section bit-flips in mantissa and FuSa violations

In this fault model, we got 0 FuSa violations. This can be attributed to the fact that mantissa bits are responsible for the precision of float32, and bit-flipping the middle section will not result in a major change of the precision of float32.



(a) Bias target, Accuracy not affected.

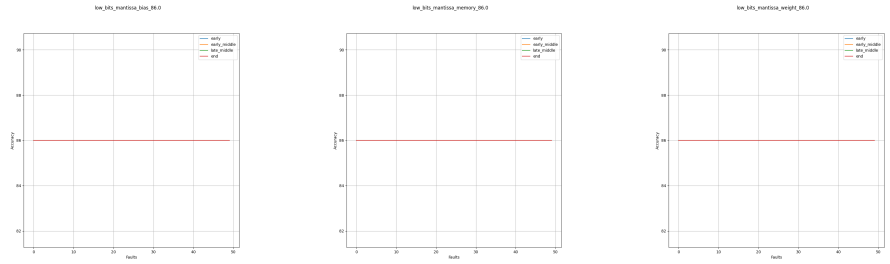
(b) Memory target, Accuracy not affected.

(c) Weights target, Accuracy not affected.

Fig. 5: Here we present the effect of middle section bit-flips in mantissa and the effect on model accuracy.

4.6 Lower section bit-flips in mantissa and FuSa violations

In this fault model, we got 0 FuSa violations, same reason as Middle section bit-flips in mantissa.



(a) Bias target, Accuracy not affected.

(b) Memory target, Accuracy not affected.

(c) Weights target, Accuracy not affected.

Fig. 6: Here we present the effect of lower section bit-flips in mantissa and the effect on model accuracy.

4.7 Lower section bit-flips in exponent and FuSa violations

In this fault model, we got a positive number of FuSa violations as opposed to middle and lower bit flips of mantissa. Exponent in float32 contributes to the magnitude of the number, which means that bit-flipping the exponent could drastically change the magnitude of the number. We can see again that bias is the most prone to induce FuSa violations due to reasons mentioned above. A floating-point number in IEEE 754 single-precision format (float32) is calculated as:

$$f = (-1)^{\text{sign}} \times (1 + \text{mantissa}) \times 2^{\text{exponent}-127}$$

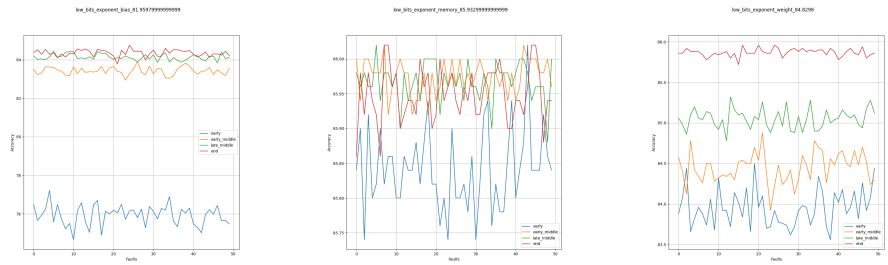
where:

- **sign** is the sign bit (1 bit).
- **exponent** is an 8-bit number.
- **mantissa** is a 23-bit fractional part.

Flipping the first two bits could result in 4 scenarios:

1. Both bits are flipped from 0 to 1: This scenario increases the exponent value by 3, thus multiplying the value of the number by 8.
2. First bit is flipped from 0 to 1 and 2nd bit from 1 to 0: This scenario increases the exponent value by 1 and decreases it by 2, thus decreasing the exponent value by 1, which means multiplying the value of the number by 0.5.
3. First bit is flipped from 1 to 0 and 2nd bit from 0 to 1: This scenario increases the value of the exponent by 1, thus multiplying the number by 2.
4. Both bits are flipped to 0: This scenario decreases the value of the exponent by 3, thus dividing the number by 8.

We can see the bias phenomena and the layer position phenomena here too.



(a) Bias target, average accuracy is 81.959.

(b) Memory target, average accuracy is 85.93.

(c) Weights target, average accuracy is 84.82.

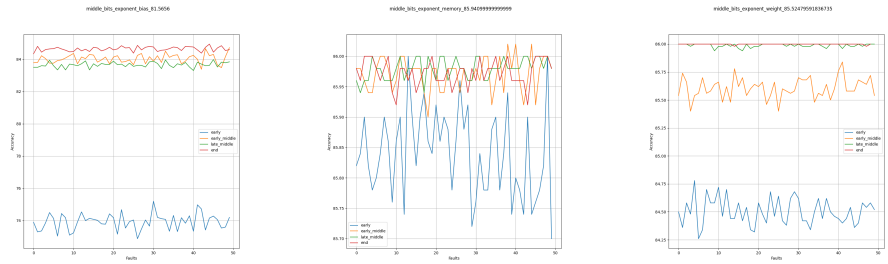
Fig. 7: Here we present the effect of lower section bit-flips in exponent and the effect on model accuracy.

4.8 Middle section bit-flips in exponent and FuSa violations

The same explanation could be provided for middle section bit-flips in exponent as the lower section, but with the the potential change of

$$2^{\pm 32 \pm 16}$$

multiplied by the number, which is a significant magnitude change. Same phenomenons seen in lower section bit-flips of exponent



(a) Bias target, average accuracy is 81.56.

(b) Memory target, average accuracy is 85.94.

(c) Weights target, average accuracy is 85.52.

Fig. 8: Here we present the effect of middle section bit-flips in exponent and the effect on model accuracy.

4.9 Higher section bit-flips in exponent and FuSa violations

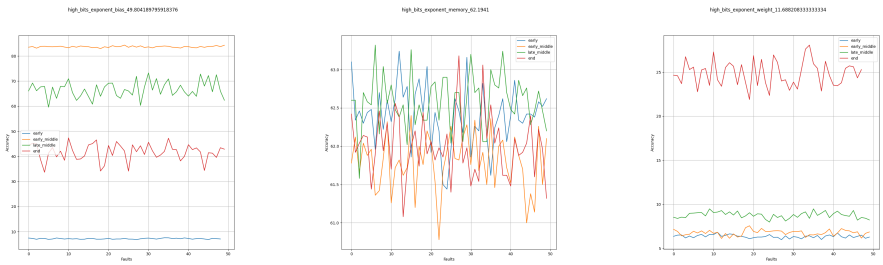
In this fault model, the number can be multiplied by

$$2^{\pm 64 \pm 128}$$

depending on the scenario of the bit-flips, which will change the number drastically, thus we see drastic results in this fault model.

Several observations can be drawn from the results, challenging some previous assumptions. Initially, it remains evident that the model exhibits some resilience to faults in memory. However, given the extensive alterations to the images, the correlation between the layer position and its impact has diminished. This could be attributed to the disruption of multiple features these layers target. It's noteworthy that approximately 12-13 functional safety (FuSa) violations arose from memory fault injections.

In contrast, fault injections into weights have a more pronounced effect compared to biases. By significantly altering weight values, the entire feature extraction process can be compromised. The location of the layer continues to influence the count of FuSa violations, with instances reaching up to 40 violations. Regarding bias injections, the layer's position still plays a pivotal role in the number of FuSa violations. Intriguingly, pronounced faults in the bias of the Terminal Layer resulted in higher FuSa violations compared to middle layers. This might be because Terminal Layers typically handle complex feature extraction. When these are compromised, it could significantly skew the model's predictions, especially when classes might have more shared intermediate features than complex ones.



(a) Bias target, average accuracy is 49.80.

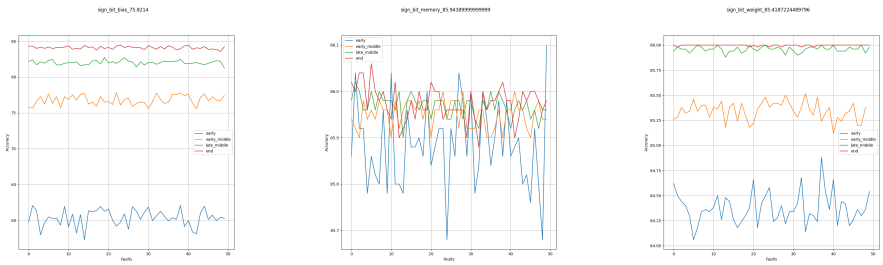
(b) Memory target, average accuracy is 62.19.

(c) Weights target, average accuracy is 11.68.

Fig. 9: Here we present the effect of high section bit-flips in exponent and the effect on model accuracy.

4.10 Sign bit flip and FuSa violations

In this fault model, we can see the bias and layer position phenomenons. Sign bit effect on the number of FuSa violation is different than the rest, due to the simple fact that the network is resilient to sign flips due to activation functions like ReLU, which will cancel the effect of the fault. Moreover, ResNet18 used Batch Normalization layers, which ensures that activation functions do not reach extreme positive and negative values, thus normalization can mitigate the sign bit flip effect.



(a) Bias target, average accuracy is 75.82.

(b) Memory target, average accuracy is 85.94.

(c) Weights target, average accuracy is 85.41.

Fig. 10: Here we present the effect of sign bit-flip and the effect on model accuracy.

5 Conclusions

This paper proposed a new robustness check of ResNet18 models against multiple fault models. We have used a wide range of fault models, including distribution of bit-flips, distribution of faults across target, number of faults and bit-granularity of faults. We have used a simulator that we developed with a CLI in order to inject faults with the settings the user needs, and dumps the fault model results into a JSON file for further inspection.

Per our findings, our results show that the location of the bit is the most dominant factor in deciding the number of FuSa violations, as flipping an exponent-related bit could devastate the network's accuracy. Moreover, since the network is sensitive to the bit location of the fault, a random fault, specially a Gaussian with a high mean, or a Uniform distributed fault will result in a drastic fall in accuracy.

In addition, the next dominant factor on the number of FuSa violations is "how deep is the layer". Our findings show that when a network parameters are not altered drastically, a layer at the beginning of the network will have much more impact than any other layer.

Lastly, the fault target plays a significant role in deciding the number of FuSa violations. We have observed that bias faults are usually the most problematic and can induce a higher number of FuSa violations as opposed to memory and weights.

We conclude that in order to preserve the FuSa of a ResNet, the most important step is to prevent FI against exponent and MSB's of the mantissa, as well as the first layers of network. This could be acheived via temporal/spatial redundancy in the network.

Acknowledgements This is the final project for 236509, Hands on Hardware Accelerators for Machine learning, supervised by Prof. Avi Mendelson and Mr Yaniv Nemcovsky, Spring 2023.

References

1. Avizienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING (2004)
2. Hou, X., Breier, J., Jap, D., Ma, L., Bhasin, S., Liu, Y.: Security evaluation of deep neural network resistance against laser fault injection (2020)
3. Javaheripi, M., Koushanfar, F.: Hashtag: Hash signatures for online detection of fault-injection attacks on deep neural networks. IEEE/ACM International Conference On Computer Aided Design (2021)
4. Kundu, S., Banerjee, S., RahaSuriyaprakash Natarajan, A., Basu, K.: Toward functional safety of systolic array-based deep learning hardware accelerator. IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS (2021)
5. Liu, Y., Wei, L., Luo, B., Xu, Q.: Fault injection attack on deep neural network. ACM International Conference on Computer-Aided Design (ICCAD) (2017)