Estimating Vulnerability of All Model Parameters in DNN with a Small Number of Fault Injections

Yangchao Zhang[1] Hiroaki Itsuji[2] Takumi Uezono[2] Tadanobu Toba[2] Masanori Hashimoto[3]
[1]Dept. Information Systems Engineering, Osaka University
[2]Center for Technology Innovation - Production Engineering and MONOZUKURI, R&D Group, Hitachi, Ltd.
[3]Dept. Communications and Computer Engineering, Kyoto University
E-mail: hashimoto@i.kyoto-u.ac.jp

***Abstract*—The reliability of deep neural networks (DNNs) against hardware errors is essential as DNNs are increasingly employed in safety-critical applications such as automatic driving. Transient errors in memory, such as radiation-induced soft error, may propagate through the inference computation, resulting in unexpected output, which can adversely trigger catastrophic system failures. As a first step to tackle this problem, this paper proposes constructing a vulnerability model (VM) with a small number of fault injections to identify vulnerable model parameters in DNN. We reduce the number of bit locations for fault injection significantly and develop a flow to incrementally collect the training data, i.e., the fault injection results, for VM accuracy improvement. Experimental results show that VM can estimate vulnerabilities of all DNN model parameters only with 1/3490 computations compared with traditional fault injection-based vulnerability estimation.**

***Index Terms*—deep neural network, network vulnerability, fault injection, bit flip, machine learning**

I. Introduction

Recent applications of DNN include safety-critical ones, such as autonomous driving. Therefore, the reliability of DNN and its hardware platform, where the representative one is GPU, is drawing attention. With this motivation, several experiments of neutron irradiation to GPU cards are reported [1]–[5] since soft error, which is primarily caused cosmic rays [6], is the dominant error source during the intermediate device lifetime [7]. Also, noise [8], aging and temperature [9] induce transient errors.

On the other hand, DNN has been proved to have high inherent error resilience; that is, the final outputs of DNN remain unchanged even some neuron outputs have been affected [10]. Nevertheless, some critical neurons have a higher probability of propagating error to the final output of DNN [11]. Once critical neurons are affected by temporal or permanent device defects, the DNN outputs wrong results, which may cause a severe failure in safety-critical applications. Also, such DNN vulnerability is drawing attention in the security community since injecting bit flips can trigger system failure and may expose critical or private information. Although such errors occuring in memory arrays can be mostly corrected by error correction code (ECC), memories in GPU are less covered by ECC compared with server CPU [4]. Therefore, finding out what are vulnerable points in the DNN is vital to improve the DNN system reliability and security since the ordinary hardware does not provide an efficient error checking mechanism with 100% coverage.

This work is supported by JST OPERA under Grant JPMJOP1721 and by Grant-in-Aid for Scientific Research (S) from JSPS under Grant JP19H05664.

Recently, the evaluation of DNN vulnerability is reported in the literature. It is mainly conducted by perturbating DNN model parameters as white box attack [11] or input data as black box attack [12]. Existing researches pointed out that the vulnerability could be measured by some indexes such as gradient [13] or neuron output value [14]. In contrast, methods for precisely estimating vulnerability, in other words, pinpointing vulnerable model parameters, are still under investigation. For example, the gradient cannot handle a bit flip in significant bits causing a large perturbation since the gradient is defined for a small perturbation despite DNN being a highly nonlinear system. For accurately evaluating the vulnerability, fault injection is utilized very commonly. However, the number of model parameters is enormous, and hence naive fault injection-based approach does not apply to modern DNNs.

This work aims to develop a scheme that can calculate the vulnerability of DNN within a reasonable time. We focus on minimum perturbation in the model parameters stored in memory, i.e., *single bit flip*. Single bit flip can be triggered by temporal and permanent faults due to, for example, soft error, noise and malicious attack mentioned above. We conduct vulnerability analysis on DNN (weight, bias) rather than the input data like "adversarial example problem" [12] or the intermediate computation data since this work emphasizes bit flips in memory storing the DNN parameters originating from soft error and malicious attack.

II. Construction of VM for DNN

A. Overview

We use a machine-learning algorithm to construct vulnerability model (VM) for estimating the vulnerability of each DNN parameter. Fig. 1 shows the proposed flow for constructing VM.

1) Select DNN parameters $\boldsymbol{p}$ randomly and calculate their vulnerability $\boldsymbol{V}$ performing fault injections, where $\boldsymbol{p}$ consists of N_p DNN parameters and $\boldsymbol{V}$ is a vector consisting of vulnerability of i-th DNN parameter $V_i (0 \leq i < N_p)$. The definition of vulnerability is given in Section II-B. The procedure of vulnerability calculation is explained in Section II-C.

2) Extract features for the DNN parameters selected in (1) as input variables $\boldsymbol{X}$ of VM, where $\boldsymbol{X}$ consists of feature vectors $\boldsymbol{X_i}(0 \leq i < N_p)$ and each feature vector $\boldsymbol{X_i}$ includes several feature values. The used features are explained in Section II-D
3) Train and test VM using a % of ($\boldsymbol{V}$, $\boldsymbol{X}$) as training data and remaining (1-a) % of ($\boldsymbol{V}$, $\boldsymbol{X}$) as test data, where VM is trained such that VM outputs $\boldsymbol{V}$ for $\boldsymbol{X}$ inputs. Parameter a is empirically determined, where $a = 0.7$ in this work. All data of ($\boldsymbol{V}$, $\boldsymbol{X}$) obtained until the current iteration are used as either training or test data.
4) Repeat (1) to (3) until a criterion is satisfied. For example, in Fig. 1, the iteration terminates when the improvement of the VM accuracy from the previous iteration is smaller than a threshold value, for example, 1%. Other criteria, such as time budget for VM construction can also be used.

B. Definition of vulnerability

This work defines vulnerability V_i for i-th DNN parameter as the sum of the accuracy degradation for individual bit flips.

$$V_i = \frac{1}{N_b} \sum_{j=1}^{N_b} (\triangle acc_{i,j}), \tag{1}$$

where $\triangle acc_{i,j}$ indicates the accuracy deviation between the original clean DNN and dirty DNN in which j-th bit of i-th DNN pamameter is flipped. N_b indicates the number of bits in one DNN parameter. The accuracy is calculated with forward propagation across batches of images in dataset. When all bit flips in i-th DNN parameter degrade the accuracy from 1 to 0, V_i becomes 1 as the worst case.

C. Efficient vulnerability calculation

For preparing $\boldsymbol{V}$ in the training and test data, we calculate $\boldsymbol{V}$ of a subset of DNN parameters $\boldsymbol{p}$ using fault injection. For efficient vulnerability calculation, we introduce an approximation below for reducing the number of fault injections, and regard V' as V.

$$V_i' = \frac{1}{N_b} \sum_{j \in \boldsymbol{bits_i}} (\triangle acc_{i,j}), \tag{2}$$

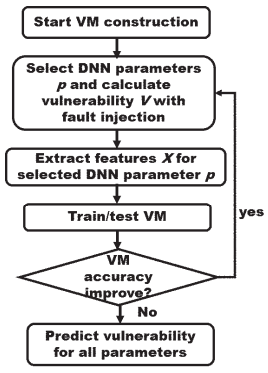

Fig. 1. VM construction flow.

where $\boldsymbol{bits_i}$ is a set that may contain integer numbers from 0 to $(N_b - 1)$. When $\boldsymbol{bits_i}$ consists 0 to $(N_b - 1)$, Eqs. (1) and (2) are identical. On the other hand, when we remove some elements of $\boldsymbol{bits_i}$, we can reduce the number of $\triangle acc_{i,j}$ calculations, i.e., the number of fault injections. The following explains how to organize $\boldsymbol{bits_i}$ for minimizing the number of fault injections while keeping the approximation error small. We exploit an observation that not all bits impact DNN output.

Let us first discuss the floating-point case. As Z. Yan et al. pointed out in [7], exponent bits have a much more significant impact on DNN output than sign and mantissa bits. Meanwhile, the value of each bit is another factor influencing the vulnerability of a parameter. Supposing an exponent bit, bit flip from 0 to 1 is serious since the parameter increases exponentially and drastically in consequence, while bit flip from 1 to 0 means the value approaches 0. In this case, the value change is at most 100% of the original value. The same discussion applies to the sign bit. As for the mantissa, the value change is smaller than 100%. We therefore put only the exponent bits whose values are 0 into $\boldsymbol{bits_i}$. Here, it should be mentioned that a similar idea is adopted for another purpose of pruning-based DNN compression [15]. H. Li et al. show that pruning the parameters that have fewer 1s has less impact on DNN output [15].

In the case of fixed-point format, MSB has the highest impact. Meanwhile, when we consider the bits that can change the parameter value by more than 100%, we put '0' bits locating in the left side of the topmost '1' bit into $\boldsymbol{bits_i}$ for a positive number case. We put '1' bits locating on the left side of the topmost '0' bits into $\boldsymbol{bits_i}$ for a negative number case.

D. Feature extraction

We have experimentally examined and selected several metrics as the key features (KF) determining the vulnerability of DNN parameters. The following explains KF for the i-th DNN parameter. Note that the KFs are normalized to restrict the value between 0 and 1. The normalized features are used as feature vector $\boldsymbol{X_i}$ for VM construction.

Absolute value of parameters. The most intensive computation in DNN is matrix computation. The magnitude of parameters for matrix computation has a large impact on the output. As a result, we select the magnitude of the parameter M_i as a KF, where $M_i = abs(parameter_i)$.

Gradient. It is reported that the vulnerability of a parameter is related to its gradient [13], which is calculated by back-propagation of DNN. Due to the nonlinearlity, the gradient is not always valid, but it could be used as a KF.

Calculation times. When a parameter participates in more computations, the probability that the error propagates to the output becomes higher. Compared with repeated calculations using the same parameter in the convolutional layer, each parameter in the fully-connected layer participates in only one calculation. We regard calculation times CT_i below as a *KF*.

$$CT_i = \begin{cases} OFW \times OFH & (\text{conv. layer}), \\ 1 \ (\text{fully connected layer}), \end{cases}$$

where OFW and OFH are the width and height of the output feature matrix.

Layer location. The location of the layer to which a DNN parameter belongs may impact vulnerability. The last layer is directly related to the output, and then it may possess a higher vulnerability. On the other hand, Z. Yan et al. indicate that the layers farther from the output are typically more sensitive and bring more significant change to the output [7]. We consider the distance from the input as ID_i, which is defined below, to reflect such observations.

$$ID = (layer\_index)/(total\ \#\ of\ layers).$$

III. Experimental results and analysis

A. Setup

We perform experimental evaluation of VM on resnet-18 [16], quantized resnet-18 and yolov3-tiny [17]. The datasets are cifar10 [18] for resnet-18 and quantized resnet-18 and COCO [19] for yolov3-tiny. Vulnerability is defined as deviation of top-5 accuracy for resnet-18, and mean average precision (mAP) deviation for yolov3-tiny. In resnet-18 and yolov3-tiny, the numbering format of DNN parameters is single precision floating-point (FP32). As for quantized resnet-18, the parameters are expressed in 8-bit fixed-point format (INT8). The quantization from i-th floating-point $P_{float,i}$ to i-th fixed-point $P_{fixed,i}$ is performed as follows.

$$\triangle P = \max_i(P_{float,i})/(2^N),$$

$$P_{fixed,i} = \mathrm{round}(P_{float,i}/\triangle P) \times \triangle P,$$

where N is the number of bits of fixed-point data, and it is 8 here. $\max()$ is a function that returns the maximum value in parameters. round() is a function returning a rounded integer. $\triangle P$ is the step size of quantization and shared by the parameters in the same layer.

When calculating gradient, which is one of KFs, 10000 images from COCO training set and 50 batches of images from cifar10 training set are used for yolov3-tiny and resnet-18, respectively. For $\boldsymbol{V'}$ calculation in Eq. (2), 40 images from COCO testing dataset and five batches of images from cifar10 testing dataset are used for yolov3-tiny and resnet-18, respectively. The batch size is set to 256 for cifar10.

For verifying the VM accuracy, we have conducted fault injections to all exponent bits of all DNN parameters in resnet-18 and calculated vulnerabilities of all DNN parameters, which are hereafter called golden vulnerability. As for quantized resnet-18 and yorov3-tiny, 1% of the parameters are fault-injected for accuracy verification due to the run-time limitation. The following compares the golden vulnerability and the vulnerability estimated by VM.

We use random forest for the vulnerability regression in the VM construction since it achieves better accuracy than other machine learning methods in our experimental setup. Meanwhile, any machine learning algorithms apply to the proposed methodology.

TABLE I
VM prediction results for three networks.

		Ratio of fault-injected params.(%)			
		0.001	0.01	0.1	1
resnet18	MAE	0.007	0.003	0.002	0.002
	R-squared	0.398	0.816	0.864	0.877
quantized resnet18	MAE	0.001	0.001	0.001	0.001
	R-squared	-0.095	0.177	0.343	0.574
yolov3-tiny	MAE	0.0003	0.0002	0.0001	-
	R-squared	0.0676	0.4682	0.6522	-

B. Vulnerability estimation results

We choose mean absolute error (MAE) and R-squared as evaluation metrics for VM prediction results. Table I presents the evaluation results. The first and second rows represent the ratio of fault-injected parameters (RoFIP) of which vulnerabilities are used for training VM. The third and fourth rows show MAE and R-squared.

Smaller MAE and higher R-squared mean the vulnerability is estimated more accurately. In resnet18, R-squared grows sharply to 0.816 when increasing RoFIP from 0.001% to 0.01%, and it slowly mounts to 0.877 when RoFIP becomes 1%. The R-squared values of quantized resnet-18 are lower than those of resnet-18, and increase steadily with the increase in RoFIP. The result of yolov3-tiny shows a similar trend as resnet-18. Here, we can see the suitable amount of training data varies depending on the network. This result indicates that the iterative VM construction in Fig. 1 is necessary.

C. Discussion

Validating $\boldsymbol{bits_i}$ selection introduced in Eq. (2).
In Section II-C, we have introduced an approximation for efficient vulnerability calculation, in which only $\boldsymbol{bits_i}$ are flipped instead of all N_b bits in a parameter. We here evaluate Δacc by injecting faults into the bits that are not included in $\boldsymbol{bits_i}$. Our expectation is that $\Delta acc = 0$ for the bits except for $\boldsymbol{bits_i}$. According to our experimental result for resnet-18 with FP32, 99.9996% bits attain $\Delta acc = 0$, and only 0.0004 % bits outside $\boldsymbol{bits_i}$ have non-zero, yet very small Δacc. This result verifies our selection of $\boldsymbol{bits_i}$. Also, it should be mentioned that this $\boldsymbol{bits_i}$ selection reduces the number of fault injections by 54.5%. As for INT8, 99.995% bits attain $\Delta acc = 0$, and only 0.005% bits outside $\boldsymbol{bits_i}$ have non-zero. This $\boldsymbol{bits_i}$ selection reduces the number of fault injections by 27.1%.

Comparison with gradient-based vulnerability estimation. Some works (e.g., [13]) have investigated the gradient-based vulnerability estimation. We here use the gradient values only as a feature for comparison. Table II shows that the R-squared values of the gradient-based estimation are negative values of -3.126 to -5.780, clearly indicating that the estimation only with the gradient is insufficient. The proposed VM with gradient and other features outperforms the gradient-based estimation.

Calculation time and accuracy compared with other fault injection-based method. Compared with the proposed VM that requires few fault injections for calculating vulnerability, conventional methods perform fault injection to

TABLE II
COMPARISON W/ GRADIENT-BASED ESTIMATION.

		Ratio of fault-injected params.(%)			
		0.001	0.01	0.1	1
Proposed	MAE	0.007	0.003	0.002	0.002
	R-squared	0.398	0.816	0.864	0.877
Gradient-Based	MAE	0.016	0.012	0.012	0.013
	R-squared	-3.126	-4.254	-4.255	-5.780

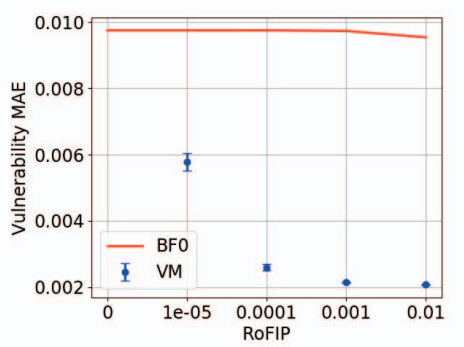

Fig. 2. Comparison between the proposed VM and conventional fault injection-based approach (BF0). Error bar indicates the standard deviation w.r.t 5 trials.

every bit. We here suppose a time-saving bit flip method for comparison. It only flips exponent bits with value 0 for floating-point data, which is hereafter called BF0. BF0 adopts the approximation regarding $\boldsymbol{bits_i}$ introduced in the VM construction (Section II-C). We compare the calculation time and accuracy between BF0 and VM.

BF0 can obtain precise vulnerability for the parameter to which faults are injected. VM can also return the exact vulnerability for the training and test data, while VM estimation may have deviation for the parameters to which faults are not injected. To consider these, we evaluate the error as follows.

$$Error = (1 - RoFIP) * \frac{1}{m} * \sum_{i=1}^{m} |GV_i - PV_i|, \quad (3)$$

where $RoFIP$ is percentage of the parameters whose golden vulnerability has been calculated by fault injection. This equation represents MAE supposing the error is zero for the parameters of $RoFIP$, thus error only happens on parameters belonging to $(1 - RoFIP)$ as shown in formula. GV_i and PV_i are golden and predicted vulnerabilities, respectively. m is the number of predicted vulnerabilities, i.e., $RoFIP \times$ (the number of all parameters). For example, if we injected faults to 0.1% parameters to train the VM, then $RoFIP$ is 0.1%. For BF0, since PV_i is not available for 99.9% parameters, we suppose $PV_i = 0$ for those parameters.

Fig. 2 shows the relation between $RoFIP$ and error defined above in resnet-18. Fig. 2 plots the error of VM for RoFIP of 0.001%, 0.01%, 0.1% and 1%. We can see VM is mitigating the large error of conventional fault injection-based method.

We finally compare the actual runtime. We analytically estimate the runtime using the following assumption. The vulnerability of ResNet-18 is analyzed with Nvidia GPU GeForce RTX 2080. The numbers of images for calculating gradient and implementing RV calculation are 10000 and 40, respectively. The runtimes of both forward and backward are around 0.002s. The numbers of bits to be flipped are 8 for VM and 3 on average for BF0. The number of parameters for fault injection is 1100 (RoFIP 0.01%), where the number of all parameters is 11,000,000. The times necessary for training and testing VM are very short, and then we ignore them. As a result, VM requires around 0.21 hours while BF0 requires 733 hours, which means 3490x speedup.

IV. CONCLUSION

This work proposed a methodology to estimate vulnerability for all parameters in DNN in a short time. We defined vulnerability and presented a procedure to construct the VM with fault injection and machine learning. Through extensive experiments and analysis, we demonstrated that the vulnerability of the DNN parameter could be predicted with reasonable accuracy in a short time, 0.21 hours for resnet-18, for example. Compared with a conventional fault injection-based approach, 3490x speedup is attained.

REFERENCES

[1] D. Oliveira *et al.*, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," *IEEE Trans. Computers*, vol. 65, no. 3, 2015.
[2] F. Santos *et al.*, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," *IEEE Trans. Reliability*, vol. 68, no. 2, 2019.
[3] A. Lotfi *et al.*, "Resiliency of automotive object detection networks on GPU architectures," *Proc. ITC*, 2019.
[4] C. Lunardi *et al.*, "On the efficacy of ECC and the benefits of FinFET transistor layout for GPU reliability," *IEEE Trans. Nuclear Science*, vol. 65, no. 8, 2018.
[5] P. Basso *et al.*, "Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs," *IEEE Trans. Nuclear Science*, vol. 67, no. 7.
[6] P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Trans. Nuclear Science*, vol. 50, no. 3.
[7] Z. Yan *et al.*, "When single event upset meets deep neural networks: Observations, explorations, and remedies," *Proc. ASP-DAC*, 2020.
[8] Y. Kim *et al.*, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, 2014.
[9] B. Nie *et al.*, "Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities," *Proc. MASCOTS*, 2017.
[10] G. Li *et al.*, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," *Proc. SC*, 2017.
[11] A. S. Rakin *et al.*, "Bit-flip attack: Crushing neural network with progressive bit search," *Proc. ICCV*, 2019.
[12] I. J. Goodfellow *et al.*, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.
[13] W. Choi *et al.*, "Sensitivity based error resilient techniques for energy efficient deep neural network accelerators," *Proc. DAC*, 2019.
[14] A. Mahmoud *et al.*, "HarDNN: feature map vulnerability evaluation in CNNs," *arXiv preprint arXiv:2002.09786*, 2020.
[15] H. Li *et al.*, "Pruning filters for efficient convnets," *arXiv preprint arXiv:1608.08710*, 2016.
[16] K. He *et al.*, "Deep residual learning for image recognition," *Proc. CVPR*, 2016.
[17] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.
[18] A. Krizhevsky *et al.*, "Learning multiple layers of features from tiny images," 2009.
[19] T.-Y. Lin *et al.*, "Microsoft coco: Common objects in context," *Proc. European conference on computer vision*, 2014.