PAPER *Special Section on VLSI Design and CAD Algorithms*

Vulnerability Estimation of DNN Model Parameters with Few Fault Injections

Yangchao ZHANG[†], **Hiroaki ITSUJI**[††], **Takumi UEZONO**[††], ***Nonmembers***, **Tadanobu TOBA**[††], ***and*** **Masanori HASHIMOTO**[†††a)], ***Members***

SUMMARY The reliability of deep neural networks (DNN) 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. We enumerate key features (KF) that characterize the vulnerability of the parameters and use KF and the collected training data to construct VM. 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.

key words: *deep neural network, fault injection, vulnerability model, soft error, malicious attack*

1. 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 by 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 error 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.

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 non-linear system. For accurately evaluating the vulnerability, fault injection is utilized very commonly. However, the number of model parameters is enormous. For example, the 18-layer ResNet (resnet-18) [15] has 11 million weight parameters, and hence naive fault injection-based approach does not apply to modern DNNs.

In this work, we aim to develop a scheme that can calculate the vulnerability of DNN within a reasonable time, for example, within an hour for resnet-18. 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. Here, considering the millions of parameters DNN possesses, it is impossible to straightforwardly evaluate the parameter one by one with fault injection as mentioned above. Therefore, finding a method that can estimate the vulnerability of DNN with a

Manuscript received March 10, 2022.
Manuscript revised July 13, 2022.
Manuscript publicized November 9, 2022.

[†]The author was with the Department of Information Systems Engineering, Graduate School of Information Science and Technology, Osaka University, Suita-shi, 565-0871 Japan.

[††]The authors are with Electronic Systems Research Dept., Production Engineering and MONOZUKURI Innovation Center, Center for Sustainability, Research & Development Group, Hitachi, Ltd., Yokohama-shi, 244-0817 Japan.

[†††]The author is with the Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University, Kyoto-shi, 606-8501 Japan.

a) E-mail: hashimoto@i.kyoto-u.ac.jp

DOI: 10.1587/transfun.2022VLP0004

small number of calculations is essential.

Thus, in this paper, we explore an approach for evaluating the vulnerability of DNN parameters with limited calculations. To the best of our knowledge, this is the first work that study the methodology of DNN vulnerability analysis with the machine learning, considering the trade-off between accuracy and computation time. For resnet-18, vulnerability prediction finishes in 0.21 hours, much less than 733 hours by smart fault injection that excludes non-sensitive bits prospectively. The contributions of this paper[†] include:

- Proposing to construct and use *vulnerability model (VM)* for estimating the vulnerability of each DNN parameter.
- Providing an efficient VM construction flow with fewer fault injections.
- Analyzing the *key features (KF)* for VM that influence the vulnerability.
- Experimentally comparing the efficiency between VM and conventional vulnerability evaluation methods for different network structures and different number formats (floating-point and fixed-point).

The rest of the paper is organized as follows. Section 2 provides the background knowledge of DNN and fault injection. The proposed DNN vulnerability evaluation method is presented in Sect. 3. Experiment results are shown in Sect. 4. Finally, Sect. 5 concludes this work.

2. Background

2.1 Deep Neural Network (DNN)

DNN is comprised of many layers, including convolutional layers, pooling layers, fully connected layers, etc. The first layer of DNN is called the input layer, and the final layer is the output layer. Other layers in the middle are hidden layers. The neurons in each layer are connected with the neurons in its upstream and downstream layers through the links having different weights and biases.

An input data is given to the first layer, doing matrix operations with weights and biases, and then operation results are obtained as the input data for the next layers. For a single neuron in a certain layer, its computation of Eq. (1) is summarized as follows; (1) receives many input data generated by neurons in the previous layer. (2) performs multiplications and additions using its own weights (and biases if available). (3) applies its activation function on the calculation result in (2), then (4) sends the result to the next layer.

$$y = Act\left(\sum_{i=1}^{N} w_i x_i + w_0\right), \tag{1}$$

where $x_1, x_2, \cdots, x_n$ are neuron outputs from the previous layer and $w_1, w_2, \cdots, w_n$ are weights corresponding to these outputs. N is the number of weights. $Act()$ represents activation function and y represents the output of this neuron. An activation function in a neural network defines how the weighted sum of the inputs is transformed into the output from a node.

2.2 Fault Injection

Faults arise unexpectedly in hardware due to voltage noise, device aging, temperature, and energetic particle radiation [6]. For analyzing the impact of such faults, fault injection experiments are widely carried out. It is typically performed by simulations on computers [17], or emulations on FPGAs [18]. Another approach is to actually irradiate the chips using a beam facility [2].

Other types of faults are injected intentionally to expose protected data, such as a private key for encryption and privacy data, and such fault injection is called a malicious attack. Such faults are injected at hardware-level and software-level. Hardware-level attacks include row-hammer attack [8], laser beam attack [19], radiation attack [2]. On software-level, adding noise [12] and bit-flip [11] are studied.

For analyzing the vulnerability of DNN, injecting a fault into DNN and evaluating the output is a common and widely-used approach because DNN is a highly non-linear system. Only using simple sensitivity-based (gradient-based in the DNN context) methods does not work well, which will be demonstrated in Sect. 4.4. Fault injection targets can be classified as input data [12], network parameters [11], network operations and instructions [20]. In this paper, fault injection is conducted by flipping a bit in network parameters supposing the radiation effect and malicious attack to the memory storing the network parameter is the primary concern.

Against both unintentional and intentional faults, identifying vulnerable DNN parameters is crucially important to improve system reliability. Without such identification, it is challenging to develop countermeasures. Alternatively, a very expensive countermeasure may need to be adopted if the actual vulnerability is unknown.

3. Construction of VM for DNN

This section proposes a methodology to construct VM using a limited number of fault injections.

3.1 Overview

We use a machine-learning algorithm to construct VM. First of all, we need to define the vulnerability of each DNN parameter. Also, we have to prepare the training data. The proposed methodology performs fault injections and uses their results as the training data. Here, the most challenging issue is that the fault injection is time-consuming, and hence the number of executable fault injections is limited. To construct VM, in summary, we need to have

- a definition of the vulnerability of individual DNN parameters, i.e., weight and bias (Sect. 3.2)

[†]A preliminary version of this paper is presented at [16].

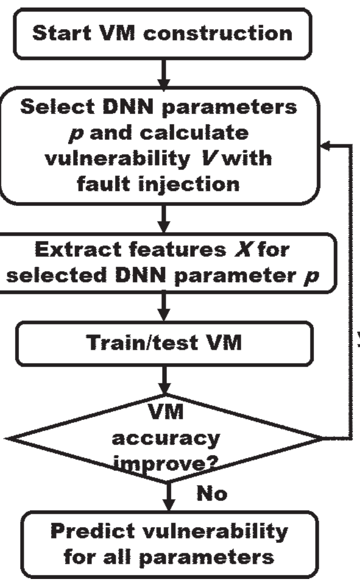

Start VM construction
Select DNN parameters p and calculate vulnerability V with fault injection
Extract features X for selected DNN parameter p
Train/test VM
VM accuracy improve?
yes
No
Predict vulnerability for all parameters

Fig. 1 VM construction flow.

- an efficient calculation procedure of vulnerability with a smart strategy to inject faults into the bit locations disturbing the DNN output (Sect. 3.3)
- a set of features that characterize the vulnerability of each neuron (Sect. 3.4)

Figure 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 Sect. 3.2. The procedure of vulnerability calculation is explained in Sect. 3.3.
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 Sect. 3.4
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. Note that 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 constructioncan also be used.

3.2 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.

$$V_i = \frac{1}{N_b} \sum_{j=1}^{N_b} (\triangle acc_{i,j}), \tag{2}$$

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.

3.3 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.

$$V'_i = \frac{1}{N_b} \sum_{j \in \boldsymbol{bits_i}} (\triangle acc_{i,j}), \tag{3}$$

where $\boldsymbol{bits_i}$ is a set that may contain integer numbers from 1 to N_b. When $\boldsymbol{bits_i}$ consists of 1 to N_b, Eqs. (2) and (3) 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. Algorithm 1 shows the detailed steps to calculate $\boldsymbol{V'}$ consisting of V'_i in Eq. (3). 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. They report that the mantissa bits have at most 1.3% of the impact of the exponent bits. The impact of the sign bit is mostly less than 10%, but it can be 35%. A simiar discussion is found in [10]. 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. A similar discussion applies to the sign bit, and the value change is smaller than 100%. As for the sign bit, the value change is 200%. 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 a different purpose of pruning-based DNN compression [21]. H. Li et al. show that pruning the parameters that have fewer 1s has less impact on DNN output [21]. Besides, if the impact of the sign bit concerns you, $\boldsymbol{bits_i}$ can include the sign bit.

In the case of fixed-point format, MSB has the highest impact. Meanwhile, when we consider the bits that can

Algorithm 1 calculate vulnerability $\boldsymbol{V'}$

```
Input: $\boldsymbol{p}$: parameters selected for fault injection, $\boldsymbol{bits}$: specified bits,
    $org\_acc$: clean DNN accuracy
Output: $\boldsymbol{V_p}$: vulnerabilities of parameters $\boldsymbol{p}$
 1: $\boldsymbol{V_p} = \boldsymbol{0}$
 2: for $i \leftarrow \boldsymbol{p}$ do
 3:     for $j \leftarrow \boldsymbol{bits}$ do
 4:         if $p_{i,j} = 0$ then
 5:             Bit flip $p_{i,j}$
 6:             Calculate $acc$
 7:             $V_i = V_i + abs(org\_acc - acc)$
 8:             Bit flip back $p_{i,j}$
 9:         end if
10:     end for
11:     $V_i = V_i \div N_b$
12: end for
```

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. This approximation will be experimentally validated in Sect. 4.4.

3.4 Feature Extraction

We examine several metrics as the key features (KF) determining the vulnerability of DNN parameters. The following discusses 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 $M_i = abs(parameter_i)$.

Number of dangerous bits. As mentioned in the previous section, not all bits have an impact on DNN output. Referring to the discussion in the previous section, the number of values included in $\boldsymbol{bits}$ in Eq. (3) is related to the vulnerability. We, therefore, regard metric D_i as a KF, where D = #of 0 in exp field.

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, as we mentioned, 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. Therefore, 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 both distances from the input and the output, ID_i and OD_i, respectively, as indicators of layer location to reflect such observations.

$$ID_i = (layer\_index)/(total\ \#\ of\ layers),\ OD_i = 1 - ID_i.$$

4. Experimental Results and Analysis

This section presents experimental results of VM accuracy and discusses VM construction procedure and VM construction time.

4.1 Setup

We perform experimental evaluation of VM on resnet-18 [15], quantized resnet-18 and yolov3-tiny [22]. The datasets are cifar10 [23] for resnet-18 and quantized resnet-18 and COCO [24] for yolov3-tiny. Vulnerability is defined as deviation of top-5 accuracy for resnet-18 and quantized 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.

$$\begin{aligned} \triangle P &= \max_i (P_{float,i})/(2^N), \\ P_{fixed,i} &= \text{round}(P_{float,i}/\triangle P) \times \triangle P, \end{aligned}$$

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 discussed in Sect. 3.4, 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. (3), 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 true vulnerability. Here, to verify this setup, we evaluated the impact of bit flips at the sign bit by injecting a bit flip into the sign bit of all DNN parameters in resnet-18. The result shows that the sign bit of all

Table 1 Spearman correlation between KFs and vulnerability for resnet-18, quantized resnet-18 and yolov3-tiny.

	Spearman correlation of KF to vulnerability					
	M_i	D_i	G_i	CT_i	OD_i	ID_i
resnet18	0.482	-0.065	0.292	0.593	-0.593	0.648
quantized resnet18	0.193	-	0.337	0.341	-0.342	0.428
yolov3-tiny	0.525	-0.165	0.137	0.257	-0.257	0.379

the parameters has zero impact. 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 true vulnerability and the vulnerability estimated by VM.

We use random forest for the vulnerability regression in the VM construction. Random forest can avoid overfitting to some extent since it consists of many models and uses average prediction of the individual models. The fast training speed and easy implementation also make it useful under many circumstances. We have experimentally found that random forest outperforms other machine learning methods, such as support vector machine and k-nearest neighbors regression, for our purpose. The hyperparameters we tuned for the experiment are mainly the max depth of the trees, max features, and minimum number for splitting samples in Python scikit-learn Library [25], and the values for these hyperparameters are 10, 3, and 10, respectively. Meanwhile, any machine learning algorithms apply to the proposed methodology.

4.2 Feature Selection

We first choose useful features from the KFs introduced in Sect. 3.4. For this purpose, we evaluated the Spearman correlation between V_i and each KF. Tabel 1 lists the Spearman correlation coefficients of individual KFs. From this table we can know that M_i, G_i, CT_i and ID_i have high correlation coefficients while D_i and OD_i remain low. Let us explain why D_i is not relevant. Since the values of almost all DNN parameters are between $-1 \sim 1$, which means the significant bits might be the same. As a result, most of the parameters have the same D_i, making D_i useless. Therefore, we choose M_i, G_i, CT_i and ID_i as KFs for training VM.

4.3 Vulnerability Estimation Results

We choose mean absolute error (MAE) and R-squared as evaluation metrics for VM prediction results. Table 2 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. Besides, an R-squared value below 0.4 would generally show a low correlation. One approach for the model construction could be, for example, increasing RoFIP as long as R-squared is lower than this threshold.

Table 2 VM prediction results of 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	-

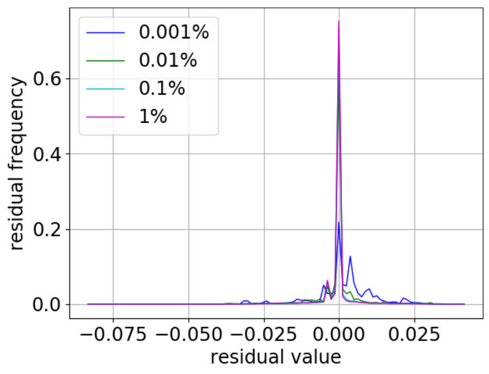
Figure 2 shows the distributions of residuals of VM. Here, the residual is calculated as the difference between the true vulnerability in Eq. (2) and the estimate obtained by VM. In Fig. 2(a), some fluctuations are found outside the range of one standard deviation when RoFIP is 0.001%. On the other hand, as the RoFIP increases, the distributions become smoother and larger errors decrease. The values of most of the residuals concentrate at around 0, meaning that the vulnerability error between predicted and actual values is small. In Fig. 2(b), the increase in RoFIP reduces the disturbance while the reduction is not significant. In Fig. 2(c), fluctuations are visible with small RoFIP. Meanwhile, as RoFIP becomes larger, the distribution becomes tighter.

4.4 Discussion

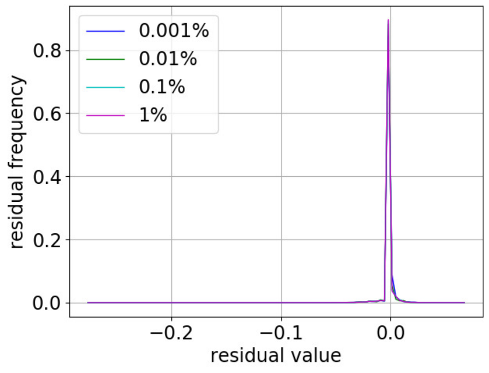
Validating $\boldsymbol{bits_i}$ selection introduced in Eq. (3). The vulnerability varies depending on the bit location significantly, especially in the case of floating-point format. Figure 3 shows the average vulnerability of 8 individual exponent bits of FP32 and all bits of INT8. For FP32, almost all vulnerability is centralized on the highest exponent bit. For INT8, on the other hand, the vulnerability decreases exponentially from high bit to low bit.

In Sect. 3.3, 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%.

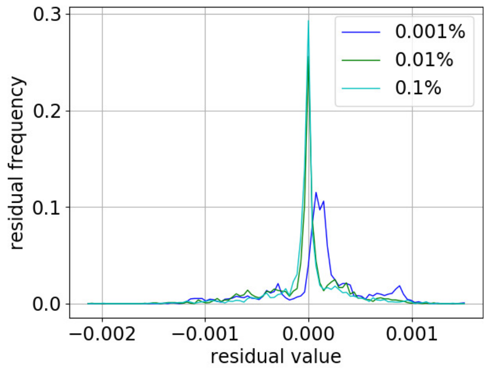
Number of test images for vulnerability evaluation.



(a) resnet-18



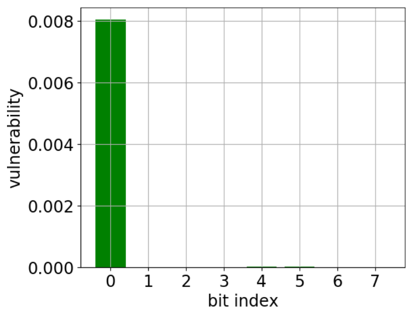
(b) quantized resnet-18



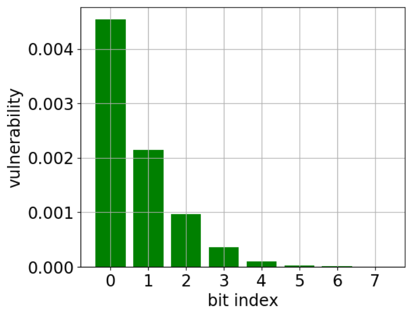
(c) yolov3-tiny

Fig. 2 Distributions of VM residual. As the RoFIP increases, the distributions become tighter and smoother, and large errors decrease.

For speeding up the VM construction, evaluation with fewer images is desirable for fault injection while the evaluated vulnerability may include large uncertainty. We here evaluate the vulnerability uncertainty for different numbers of images. Figure 4 shows the vulnerabilities of 100 DNN parameters, where the black bar represents one standard deviation across the 100 trials. The error bar becomes smaller with the increase in the number of images while increasing the image number means longer computation time. The computation time increases linearly according to the number of images. Considering this trade-off, we need to determine the number of images for fault injection.



(a) resnet-18



(b) quantized resnet-18

Fig. 3 Average vulnerability at each bit location.

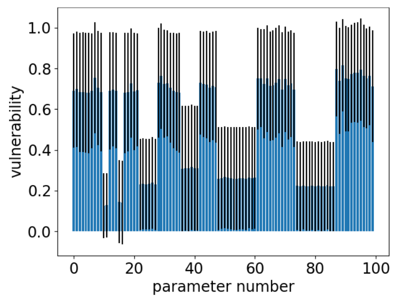
Comparison with gradient-based vulnerability estimation. Some works (e.g., [13]) have investigated the gradient-based vulnerability estimation. We here use the gradient values only as a feature for comparison. Table 3 shows that the R-squared values of the gradient-based estimation are negative values 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 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 (Sect. 3.3). We compare the calculation time and accuracy between BF0 and VM.

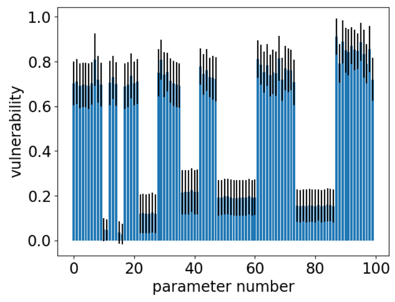
BF0 can obtain precise vulnerability for the parameter to which faults are injected. VM can also return the exact vulnerability for the training and test data, while VM estimation may have deviation for the parameters to which faults are not injected. To consider these, we evaluate the error as follows.

$$Error = \frac{1}{N_p} \times \sum_{i=1}^{N_p} |TV_i - PV_i|. \tag{4}$$

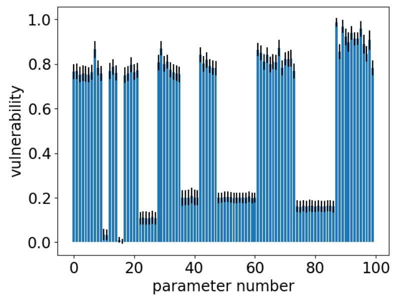
Remind that N_P is the number of parameters. TV_i and PV_i are true and predicted vulnerabilities, respectively. Note that $GV_i = PV_i$ for the $RoFIP \times N_p$ parameters thanks to the fault



(a) 10 test images



(b) 100 test images



(c) 1000 test images

Fig. 4 Vulnerability distribution of selected 100 parameters obtained by using different numbers of images.

Table 3 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

injection. Thus, for the proposed method, $Error$ consists of the prediction errors for $(1 - RoFIP) \times N_p$ parameters. On the other hand, for BF0, the predicted vulnerability PV_i is not available for $(1 - RoFIP) \times N_p$ parameters. Then, we suppose $PV_i = 0$ for those parameters.

Figure 5 shows the relation between $RoFIP$ and error defined above in resnet-18. Figure 5 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 estimate the runtime using the analytical models in Table 4. To obtain

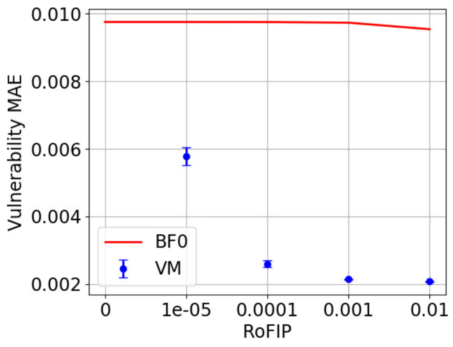


Fig. 5 Comparison between the proposed VM and conventional fault injection-based approach (BF0). Error bar indicates the standard deviation w.r.t 5 trials.

Table 4 Runtime expectation.

Method	Runtime estimate
VM	sub_test[1] * (forward + backward[2]) + sub_train[1]* forward * N_bit[3]* N_param[4] + VM_train[5]+ VM_test[5]
BF0	sub_train * forward * N_bit * All_param[6]

[1] Part of train/test dataset.
[2] Average runtime for a single forward/backward propagation.
[3] Number of bits to be flipped for every parameter.
[4] Number of parameters used for calculating real vulnerability.
[5] Runtime for training/testing VM.
[6] Number of total parameters in DNN.

concrete runtime, we assume the vulnerability of ResNet-18 is analyzed with Nvidia GPU GeForce RTX 2080. The sub_test and sub_train for calculating gradient and implementing RV calculation is 10000 and 40 images. The runtimes of both forward and backward are around 0.002s. N_bit is 8 for VM and 3 on average for BF0. N_param is 1100 and All_param is 11,000,000 for resnet-18 if we choose 0.01% as RoFIP. The times necessary for training and testing VM are very short, and then we ignore VM_train and VM_test here. As a result, VM requires around 0.21 hours while BF0 requires 733 hours, which means 3490x speedup.

5. 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. We examined the features of each DNN parameter. 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.

Acknowledgments

This work is supported in part by the Japan Science and Technology agency, Program on Open Innovation Platform with

Enterprises, Research Institute and Academia (JST-OPERA) Program, Japan, under Grant Number JPMJOP1721; and in part by the JSPS Grants-in-Aid for Scientific Research (KAKENHI), Japan, under Grant Number 19H05664.

References

[1] D.A.G. Gonçalves de Oliveira, L.L. Pilla, T. Santini, and P. Rech, "Evaluation and mitigation of radiation-induced soft errors in graphics processing units," IEEE Trans. Comput., vol.65, no.3, pp.791–804, 2016.

[2] F.F. dos Santos, P.F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," IEEE Trans. Rel., vol.68, no.2, pp.663–677, 2019.

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

[4] C. Lunardi, F. Previlon, D. Kaeli, and P. Rech, "On the efficacy of ECC and the benefits of FinFET transistor layout for GPU reliability," IEEE Trans. Nucl. Sci., vol.65, no.8, pp.1843–1850, 2018.

[5] P.M. Basso, F.F. dos Santos, and P. Rech, "Impact of tensor cores and mixed precision on the reliability of matrix multiplication in GPUs," IEEE Trans. Nucl. Sci., vol.67, no.7, pp.1560–1565, 2020.

[6] P.E. Dodd and L.W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," IEEE Trans. Nucl. Sci., vol.50, no.3, pp.583–602, 2003.

[7] Z. Yan, Y. Shi, W. Liao, M. Hashimoto, X. Zhou, and C. Zhuo, "When single event upset meets deep neural networks: Observations, explorations, and remedies," 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), IEEE, pp.163–168, 2020.

[8] Y. Kim, R. Daly, J. Kim, C. Fallin, J.H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," ACM SIGARCH Computer Architecture News, vol.42, no.3, pp.361–372, 2014.

[9] B. Nie, J. Xue, S. Gupta, C. Engelmann, E. Smirni, and D. Tiwari, "Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities," 2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp.22–31, 2017.

[10] 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," Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, 2017.

[11] A.S. Rakin, Z. He, and D. Fan, "Bit-flip attack: Crushing neural network with progressive bit search," Proc. IEEE/CVF International Conference on Computer Vision, pp.1211–1220, 2019.

[12] I.J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," arXiv preprint arXiv:1412.6572, 2014.

[13] W. Choi, D. Shin, J. Park, and S. Ghosh, "Sensitivity based error resilient techniques for energy efficient deep neural network accelerators," 2019 56th ACM/IEEE Design Automation Conference (DAC), pp.1–6, 2019.

[14] A. Mahmoud, S.K.S. Hari, C.W. Fletcher, S.V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M.B. Sullivan, T. Tsai, and S.W. Keckler, "HarDNN: Feature map vulnerability evaluation in CNNs," arXiv preprint arXiv:2002.09786, 2020.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," Proc. IEEE Conference on Computer Vision and Pattern Recognition, pp.770–778, 2016.

[16] Y. Zhang, H. Itsuji, T. Uezono, T. Toba, and M. Hashimoto, "Estimating vulnerability of all model parameters in DNN with a small number of fault injections," Design, Automation and Test in Europe Conference (DATE), 2022.

[17] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "BinFI: An efficient fault injector for safety-critical machine learning systems," Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, pp.1–23, 2019.

[18] F. Benevenuti, F. Libano, V. Pouget, F.L. Kastensmidt, and P. Rech, "Comparative analysis of inference errors in a neural network implemented in SRAM-based FPGA induced by neutron irradiation and fault injection methods," 2018 31st Symposium on Integrated Circuits and Systems Design (SBCCI), pp.1–6, 2018.

[19] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," Proc. IEEE, vol.100, no.11, pp.3056–3076, 2012.

[20] J. Breier, X. Hou, D. Jap, L. Ma, S. Bhasin, and Y. Liu, "Practical fault attack on deep neural networks," Proc. 2018 ACM SIGSAC Conference on Computer and Communications Security, pp.2204–2206, 2018.

[21] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H.P. Graf, "Pruning filters for efficient ConvNets," arXiv preprint arXiv:1608.08710, 2016.

[22] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," arXiv preprint arXiv:1804.02767, 2018.

[23] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Technical Report, University of Toronto, Toronto, Ontario, 2009.

[24] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C.L. Zitnick, "Microsoft COCO: Common objects in context," European Conference on Computer Vision, pp.740–755, Springer, 2014.

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol.12, pp.2825–2830, 2011.

Yangchao Zhang received the B.E. degree from Wuhan University and M.E. degree in information systems engineering from Osaka University, Suita, Japan in 2019 and 2021, respectively. She is currently with Micron Inc. Data Science Department, Hiroshima, Japan. Her research interests include reliability of neural network.

Hiroaki Itsuji received the M.S. and the Ph.D. degree in electrical engineering and information systems from the University of Tokyo, Tokyo, Japan. He is currently a Senior Researcher in the Center for Technology Innovation - Production Engineering and MONODUKURI, Research and Development Group, Hitachi, Ltd., Yokohama, Japan. His current research interests include the system design of highly-reliable fault-tolerant systems.

Takumi Uezono received the B.E., M.E., and Ph.D. degrees in computer science and in electrical and electronic engineering from the Tokyo Institute of Technology, Tokyo, Japan, in 2005, 2007, and 2010, respectively. He was a Post-Doctoral Researcher with Kyoto University, Kyoto, Japan, from 2010 to 2011. He was also a Research Fellow with the Japan Society for the Promotion of Science, from 2009 to 2011. He is currently a Chief Researcher in the Center for Technology Innovation - Production Engineering and MONODUKURI, Research and Development Group, Hitachi, Ltd., Yokohama, Japan. His current research interests include the circuit and system design for highly reliable system and the physical design of high-performance integrated circuits.

Tadanobu Toba received the M.S. and the Ph.D. degree in system safety and information science from the Nagaoka University of Technology, Niigata, Japan. He is currently a Chief Researcher in the Center for Technology Innovation - Production Engineering and MONODUKURI, Research and Development Group, Hitachi, Ltd., Yokohama, Japan. His current research activity is dependable technology and system design.

Masanori Hashimoto received the B.E., M.E. and Ph.D. degrees in communications and computer engineering from Kyoto University, Kyoto, Japan, in 1997, 1999, and 2001, respectively. He is currently a Professor in Graduate School of Informatics, Kyoto University, Kyoto, Japan. His current research interests include the design for manufacturability and reliability, timing and power integrity analysis, reconfigurable computing, soft error characterization, and low-power circuit design. Dr. Hashimoto was a recipient of the Best Paper Awards from ASP-DAC in 2004 and RADECS in 2017, and the Best Paper Award of the IEICE Transactions in 2016. He served as the TPC chair and co-chairs for ASP-DAC 2022 and MWSCAS 2022, respectively. He serves/served as the Editor-in-Chief for Elsevier Microelectronics Reliability and an Associate Editor for the IEEE Transactions on VLSI Systems, IEEE Transactions on Circuits and Systems I, and ACM Transactions on Design Automation of Electronic Systems.