Efficient Identification of Critical Faults in Memristor Crossbars for Deep Neural Networks

Ching-Yuan Chen and Krishnendu Chakrabarty
Department of Electrical and Computer Engineering, Duke University, Durham, NC 27708, USA

***Abstract*—Deep neural networks (DNNs) are becoming ubiquitous, but hardware-level reliability is a concern when DNN models are mapped to emerging neuromorphic technologies such as memristor-based crossbars. As DNN architectures are inherently fault-tolerant and many faults do not affect inferencing accuracy, careful analysis must be carried out to identify faults that are critical for a given application. We present a misclassification-driven training (MDT) algorithm to efficiently identify critical faults (CFs) in the crossbar. Our results for two DNNs on the CIFAR-10 data set show that MDT can rapidly and accurately identify a large number of CFs—up to 20× faster than a baseline method of forward inferencing with randomly injected faults. We use the set of CFs obtained using MDT and the set of benign faults obtained using forward inferencing to train a machine learning (ML) model to efficiently classify all the crossbar faults in terms of their criticality. We show that the ML model can classify millions of faults within minutes with a remarkably high classification accuracy of over 99%. We present a fault-tolerance solution that exploits this high degree of criticality-classification accuracy, leading to a 93% reduction in the redundancy needed for fault tolerance.**

I. Introduction

Deep learning (DL) applications, e.g., self-driving cars, image recognition, and medical diagnosis, are becoming an increasingly important part of our daily lives. A promising implementation pathway for DL models is to map them to specialized neuromorphic hardware, e.g., memristor-based crossbars [1]. Such a mapping provides in-memory computing capability, high energy efficiency, and a means to overcome the memory wall arising from the von Neumann bottleneck.

However, recent research has highlighted hardware-level reliability concerns associated with memristor-based architectures. Memristor devices are subject to various types of manufacturing defects and process variations [2]. As demonstrated in [3] and [4], both training and inference on the memristor crossbar are significantly degraded by defects, variations, and non-idealities. These defects, variations, and non-idealities can be viewed as *faults*, defined as any deviation from the nominal expected behavior of memristor cells in the crossbar.

To ensure fault tolerance, techniques that detect, locate, and recover from faults have been proposed for memristor-based crossbars [5] [6]. Despite the benefits offered by theses solutions, there are some key challenges that have yet to be addressed:

Incomplete fault analysis. Conventional fault simulation for crossbars enumerates faults to understand their impact on classification accuracy for a given set of input data [5]. A problem with this approach is that a realistic set of crossbars has a massive number of modeled faults (typically tens of millions) and exhaustive functional fault simulation is computationally impractical—simulation has to iterate through tens of millions of faults and each simulation run needs to consider the full input data set.

However, it has been shown that not all faults in the crossbars are functionally critical [6]. We show later that less than 0.1% of the faults are critical for a large DNN that is mapped to memristor crossbars, i.e., these critical faults (CFs) lead to misclassification; other faults are benign. The identification of CFs is therefore akin to finding the proverbial needle in a haystack.

Cost of fault tolerance. Since most crossbar faults are benign, it is an overkill to make every memristor cell in the crossbar robust to faults. For example, spare crossbar columns and rerouting circuits can be incorporated to enable remapping, but this solution is ineffective without an understanding of the locations of the small number of CFs in the crossbar.

In this paper, we present a fault-criticality assessment technique that allows us to efficiently identify the CFs in a crossbar. The main contributions of this paper are as follows.

- **Misclassification-driven training algorithm.** We propose a misclassification-driven training (MDT) algorithm to efficiently identify CFs in the crossbar. We use repeated runs of MDT to locate CFs for the CIFAR-10 data set.
- **Machine learning for criticality prediction.** After a sufficient number of CFs are identified using MDT and a corresponding number of benign faults are identified using randomly sampling followed by forward inferencing, we train a machine learning (ML) model to predict fault criticality without explicitly running fault simulation or MDT.
- **Criticality-aware fault tolerance.** We present a fault-tolerance solution that exploits this high degree of criticality-classification accuracy to target only CFs, leading to a 93% reduction in the redundancy needed for fault tolerance.

II. Preliminaries

A. Memristor Crossbars and Parameter Mapping

A memristor can be viewed as a programmable resistor. A crossbar with memristor cells that can have 64 conductance levels was demonstrated in [1]. A crossbar can carry out vector-matrix computation in O(1) time, and crossbar-based in-memory computing architectures, e.g., PRIME [8], have been proposed in the literature.

A DNN typically has tens of millions of model parameters (weights), and because of this large size, it is a challenge to map the full network to on-chip crossbars. Therefore, model-pruning techniques have been proposed to remove more than

978-3-9819263-5-4/DATE21/©2021 EDAA

TABLE I: Notation used in the description of MDT.

Symbol	Explanation	Symbol	Explanation
F	DNN architecture	$\mathcal{L}$	loss function
$\mathbf{\Theta}$	DNN parameter matrix	$\mathbf{\Delta}$	deviation matrix
$\mathcal{D}$	data set	$\|\cdot\|_0$	0-norm function [12]
x	input data	$\|\cdot\|$	number of matrix (vector) elements
y	output label	B_i	i-th data batch
$\mathcal{M}(\cdot)$	mapping function	∇	gradient
$S(\cdot)$	significance function	l	parameter/fault location

90% of the parameters without any accuracy loss [9]. In this paper, we adopt the solution proposed in [10] to map each parameter to a physical memristor in the crossbar. Specifically, we use a linear mapping $\mathcal{M}$ to map a parameter value θ to a conductance value g of the corresponding memristor, where

$$\mathcal{M} : g = \begin{cases} \frac{g_{\text{on}} - g_{\text{off}}}{\theta_{\text{max}}} \cdot \theta, & \text{if } \theta > 0 \\ \frac{g_{\text{on}} - g_{\text{off}}}{\theta_{\text{min}}} \cdot \theta, & \text{if } \theta < 0 \end{cases} \quad (1)$$

From (1), we can see conductance of the memristor is in the range $[g_{\text{on}}, g_{\text{off}}]$.

Although model pruning and sparse mapping significantly reduce the size of the crossbar needed for DNNs, millions of memristor cells are still required for pruned DNN models. We show later that we need as many as 2,088 crossbars of size 128×128 (34 M cells) for mapping the VGG-16 DNN [11] even after pruning. Efficient fault-criticality analysis is therefore also important for pruned DNNs mapped to crossbars.

B. Fault Models

Faults in the crossbar result in the deviation of the DNN model parameters (referred to using matrix $\mathbf{\Theta}$) from their original values. We refer to the deviation in the model parameters using the matrix $\mathbf{\Delta}$. The model parameter matrix for a DNN with crossbars faults is therefore $\mathbf{\Theta} + \mathbf{\Delta}$. In this work, our focus is on stuck-on/off faults [2].

When a memristor cell is stuck-on or stuck-off, i.e., its conductance is stuck at either g_{on} (stuck-on) or g_{off} (stuck-off). Correspondingly, (1) implies that a stuck-on (off) cell will force the associated parameter to be θ_{max} or θ_{min} (0).

III. Misclassification-driven Training

In this section, we first provide details about the MDT algorithm to identify CFs. Following this, we specify how we train the ML-based CF classifier. In each MDT run, we identify a set of faults that impact the DNN model accuracy. To achieve this objective, we attempt to perturb the model parameters to force misclassifications. After we derive the parameter deviations, we map them to physical memristor faults.

A. Problem Formulation

We formulate CF identification as an optimization problem, which is subsequently tackled using MDT. First, we introduce the notation used in the subsequent discussion; see Table I. Given a DNN architecture F, its parameters $\mathbf{\Theta}$, data set $(x, y) \in \mathcal{D}$, where x stands for input set and y denotes the output label, our objective is to determine the *significance* of each parameter (fault location). We target deviations in the most significant parameters that can cause misprediction. This problem can be mapped to the following optimization problem in terms of the parameter deviation $\mathbf{\Delta}$:

$$\max_{\Delta} \sum_{(x,y)\in\mathcal{D}} \mathcal{L}(F_{\mathbf{\Theta}+\mathbf{\Delta}}(x), y) \text{ such that } \|\mathbf{\Delta}\|_0 < c \quad (2)$$

where $\mathcal{L}$ is the loss function used to evaluate the deviation between the model output and the correct label y. We map the matrix $\mathbf{\Delta}$ obtained in this way to faults in the underlying memristor-based crossbar. Note that the constraint $\|\mathbf{\Delta}\|_0 < c$ is imposed to ensure that the optimization problem will converge. In the MDT algorithm, we do not explicitly confine the 0-norm of the resulting deviation $\|\mathbf{\Delta}\|_0$.

To train an accurate ML model, we carry out a verification step by performing forward inferencing to verify that each CF fault identified by MDT is actually critical. Our results show that only a small fraction of faults (less than 8%) identified as critical by MDT is actually benign; these faults are appropriately treated as benign during training.

1: **Inputs:** DNN architecture F, parameters $\mathbf{\Theta}$, data set $(x, y) \in \mathcal{D}$, mapping $\mathcal{M}$ from parameters value to conductance
2: **Output:** CF list CF
3: $\mathbf{\Theta} \leftarrow \mathbf{\Theta} + \mathcal{N}(0, \sigma^2)$ /*Slightly perturb parameters*/
4: $\mathbf{\Delta} \leftarrow \mathbf{0}; CF \leftarrow \emptyset$
5: **for** Batch $B_i \in \mathcal{D}, i = 0, ..., T-1$ **do**
6: **while** $\exists (x, y) \in B_i$ s.t. $F_{\Theta+\Delta}(x) = y$ **do**
7: Compute gradient $\nabla_{\mathbf{\Delta}} = \nabla\mathcal{L}(F_{\Theta+\Delta}(x), y)$
8: $l^* = \text{argmax}_{0 \le l < |\mathbf{\Theta}|} S(\nabla_{\mathbf{\Delta}})[l]$
9: Modify $\mathbf{\Delta}[l^*]$ and infer fault type f
10: CF.push(l^*, f, $\mathcal{M}(\mathbf{\Delta}[l^*])$) /*Add newly identified CF*/
11: **end while**
12: **end for**
13: **return** CF

Fig. 1: Pseudocode for the MDT algorithm.

B. MDT based on Gradient Ascent

We solve the optimization problem (2) using a gradient-based algorithm. In each gradient step, we greedily pick the parameter that has the most significant impact on the predicting results. The injected deviations are mapped to physical memristor faults.

Figure 1 presents the pseudocode for the MDT algorithm. Based on [13], we perturb the model parameter $\mathbf{\Theta}$ to prevent our optimization algorithm from getting stuck at a local minima (Line 3 in Figure 1). This step also ensures that each run of MDT will yield a different CF. Note that the perturbation step should not affect the model predicting accuracy. Specifically, we add zero-centered Gaussian noise to each parameter with the standard deviation $\sigma = 0.0025 \times (\theta_{\text{max}} - \theta_{\text{min}})$. Next, we use the i-th batch B_i sampled from the data set $\mathcal{D}$ to identify CFs. We iteratively inject CFs until all data points in B_i are mispredicted by the faulty model $F_{\Theta+\Delta}$.

In each iteration for the sampled batch B_i, we derive the gradient to determine the parameter with the most significance (Line 7). We denote the derived gradient corresponding to the current deviation $\mathbf{\Delta}$ as $\nabla_{\mathbf{\Delta}}$. We then calculate the significance function $S(\cdot)$ for each parameter (or fault location) based on

$\nabla_{\mathbf{\Delta}}$. The significance of a parameter is defined in this paper as the absolute value of the gradient, i.e.,

$$S(\nabla_{\mathbf{\Delta}}) = |\sum_{(x,y)\in B_i} \nabla\mathcal{L}(F_{\Theta+\Delta}(x), y)|, \quad (3)$$

where $|\cdot|$ is the element-wise absolute function. We select the location with the most significance, denoted by l^*, to inject a fault (Line 8).

We then update the fault-induced deviation Δ at l^* according to the direction (sign) of the gradient. This is a *gradient-ascent* step aimed at maximizing the loss function $\mathcal{L}(\cdot)$. By 'ascent', we mean that the change in direction of the deviation $\mathbf{\Delta}$ is the same as the gradient at l^*, or $\nabla_{\mathbf{\Delta}}[l^*]$.

Figure 2(a) shows a working example of the MDT algorithm. The 2×2 fully connected layer with two input neurons (i_1 and i_2) and two output neurons (o_1 and o_2) has four parameters θ_{11}, θ_{12}, θ_{21}, and θ_{22}. Each of the parameters is mapped to a memris tor. Suppose we have $[\theta_{11}, \theta_{12}, \theta_{21}, \theta_{22}] = [0.15, 0.20, 0.35, 0.25]$. After we obtain the gradient for each parameter $[\nabla_{11}, \nabla_{12}, \nabla_{21}, \nabla_{22}] = [0.10, 0.20, -2.00, -0.10]$ using backpropagation, we identify θ_{21} as the most significant parameter using $S(\cdot)$ in (3) (Line 8 in Figure 1). Next, we introduce the deviation to θ_{21} in the same direction as the obtained gradient and $\mathbf{\Delta} = [0, 0, -0.35, 0]$. The deviated parameter $\theta_{21}+\Delta_{21}$ now become zero as it's a stuck-off fault. We then obtain the deviated memristor conductance using the mapping $\mathcal{M}$, i.e., the linear mapping mentioned in Section II-A. We show in the right-hand side of Figure 2(a) that the corresponding conductance g_{21} in the crossbar (marked with a cross) is deviated by a magnitude of $\mathcal{M}(\Delta_{21}) = \mathcal{M}(-0.35)$ because of the stuck-off fault. Finally, we record the newly identified CF (Line 10 in Figure 1). A CF is specified by its location l^*, fault type f, and the deviation value $\mathcal{M}(\mathbf{\Delta}[l^*])$. Note that we do not physically map a pruned (zero) parameter to a crossbar cell—this parameter is not used for inferencing—therefore faults are not injected at pruned parameter locations.

C. Classification of Faults Using Machine Learning

We next describe an ML-based classifier that can predict whether a fault is catastrophic. For the collection of training data, we used both MDT and the random-sampling baseline. Using MDT, we are able to quickly generate CFs for training, and we complement this data set with benign faults generated using sampling-based fault injection and forward inferencing.

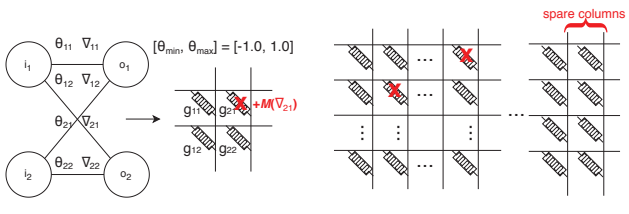


Fig. 2: (a) Illustration of MDT; (b) the proposed fault-tolerance solution.

MDT allows us to diversify (balance) the extremely unbalanced data points generated by the baseline method.

In each forward-inferencing pass, we randomly inject a crossbar fault and observe the accuracy drop. If the inferencing accuracy is lower than a preset threshold A_{th}, we label the fault as catastrophic, otherwise it is benign. Note that an overwhelming majority of faults analyzed in this way are benign, so this step is used to generate training data with respect to benign faults.

We use the following features to train the ML model: 1) fault location, 2) fault type, 3) parameter significance (described in Section III-B), and 4) deviation amount. The fault location is given by the DNN layer of the neuron that is mapped to the corresponding memristor. The significance of the deviated parameter is an important feature for correctly predicting the fault criticality. We use the absolute value of the gradient at the injected fault location, as described in (3), as the parameter significance. We use a neural network with one hidden layer as our ML-based binary classifier. All layers in our network are fully connected and the output non-linear layer is Softmax. The hidden layer has 64 neurons.

We use *recall* (percentage) as a metric for our CF classifier; it is defined as the ratio of the number of CFs correctly identified as being critical to the total number of CFs in the data set. Clearly, a high value of recall is desirable because it indicates low misclassification of CFs, and eventually low *test escape* in the context of testing and fault tolerance.

IV. Criticality-aware Fault Tolerance

We use the trained CF classifier to reduce the cost of fault tolerance for a crossbar on which a DNN is mapped. Fault tolerance can be added to the crossbar to support online fault detection and recovery. For example, spare columns are utilized for remapping after fault detection [6]. In our solution, we introduce a spare column for remapping only when the classifier predicts that at least one of the cells in the column has a CF. Figure 2(b) shows an example. Suppose there are two CFs in the crossbar (marked with crosses), as determined using the CF classifier; only two spare columns are required to ensure the fault tolerance.

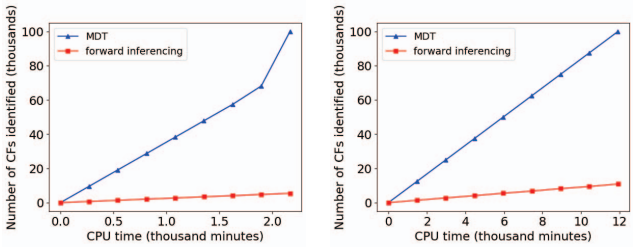
V. Experimental Results: Training Data

We use AlexNet [7] and VGG-16 [11] for our experiments (Table II). We use the CIFAR-10 data set, with 10 classes in total, for evaluation. The training set $\mathcal{D}_{train}$ for CIFAR-10 consists of 50,000 data points and there are 10,000 points in the testing set $\mathcal{D}$. We use $\mathcal{D}$ to derive CFs, and $\mathcal{D}_{train}$ to train the DNN model. We use [9] to prune 90% of the model parameters for AlexNet and VGG-16 without any accuracy loss. We list details of the pruned models in Table II, using the names AlexNet-P and VGG-16-P, respectively.

As described in Section III-C, we use MDT and the baseline method to generate CF and non-CF data points, respectively, to train the ML classifier. We use the setting of the classification accuracy threshold A_{th} to determine whether a fault is critical: 70% for AlexNet-P and 80% for VGG-16-P. We injected

TABLE II: DNN models used in our experiments

	AlexNet	VGG-16	AlexNet-P	VGG-16-P
Baseline top-1 Accuracy (%)	79.96	90.17	81.10	90.80
No. convolution layers	5	13	5	13
No. fully connected layers	3	3	3	3
No. parameters	61 M	138 M	5.3 M	10.5 M



(a) (b)

Fig. 3: CPU time needed for the generation of CFs (training data) for (a) AlexNet-P, (b) VGG-16-P.

100,000 faults in AlexNet-P and 0.82% of these faults were CFs. For VGG-16-P, we injected 100,000 faults and 0.84% of these faults were critical. We recorded the CPU time spent in identifying CFs by the two methods, as shown in Figure 3. MDT can identify up to 20 times more CFs within the same period of time.

VI. Evaluation of Trained ML Model

In this section, we present classification results (CF versus benign) obtained using the ML model for a large number of crossbar faults in both AlexNet-P and VGG-16-P. We collected a balanced data set with 100,000 CF and 100,000 benign fault samples by the MDT and baseline methods.

Table III shows the classification results for the given accuracy threshold setting. The results are obtained using the trained ML model for AlexNet-P and VGG-16-P. In each experiment, we randomly draw 30% of the balanced data set to evaluate the trained ML model. The remaining 70% of the data is used for training in each experiment, and we repeat the experiment 10 times. The table shows the mean accuracy results; the standard deviations are less than 0.1%. The ML model achieves over 99% accuracy and over 99% recall for AlexNet and VGG-16.

We also classified each possible fault in the crossbar as being either benign or critical using the ML model. Table IV shows the results for AlexNet-P and VGG-16-P.

A. Cost-effective Crossbar Column Redundancy

With the criticality information of DNNs obtained using MDT and the ML-based CF classifier, we can deploy spare crossbar columns only for memristors associated with CFs to reduce overhead. We reuse the classification results in Section VI. DNNs are mapped to crossbars of sizes of 128×128

TABLE III: Classification accuracy of trained ML models for AlexNet-P and VGG-16-P.

AlexNet-P		VGG-16-P	
Accuracy (%)	Recall (%)	Accuracy (%)	Recall (%)
99.91	99.71	99.36	99.01

TABLE IV: Number (and percentage) of CFs identified by the trained ML model for the crossbar faults considered.

	AlexNet-P	VGG-16-P
No. faults classified as CFs	2,877	5,355
CPU time (seconds)	13	17
Percentage of CFs (%)	0.03	0.04

TABLE V: Criticality-aware redundancy for DNN models

	AlexNet-P	VGG-16-P
No. 128×128 crossbars	825	2,088
No. spare columns deployed using [6]	43,805	55,055
No. spare columns deployed using proposed method	2,814	3,619
Percentage overhead reduction using proposed methods	93.58	93.43

using [10]. We use [6] as the baseline in the redundancy evaluation experiment; 5% of the faults are reported to be critical in [6]. Table V shows the number of crossbars used for each DNN and the number of the spare columns required. Our results show that we can reduce hardware overhead by 93% compared to [6].

VII. Conclusion

We have presented the MDT algorithm to efficiently identify CFs in a memristor crossbar used for DNNs. We have used the set of CFs obtained using MDT and the set of benign faults obtained using forward inferencing to train an ML model for efficient fault classification. The ML model can classify tens of millions of faults within minutes with a high accuracy of over 99%. The proposed fault-tolerant design that targets only CFs reduces hardware overhead by 93% for redundant crossbar columns. The probability of incorrectly classifying CFs is less than 0.3% for AlexNet and less than 1% for VGG-16, implying that the proposed method is unlikely to cause test escape.

References

[1] M. Hu et al., "Dot-product engine for neuromorphic computing: Programming 1T1M crossbar to accelerate matrix-vector multiplication," in *DAC*, 2016.

[2] A. Chaudhuri and K. Chakrabarty, "Analysis of process variations, defects, and design-induced coupling in memristors," in *ITC*, 2018.

[3] X. Sun and S. Yu, "Impact of non-ideal characteristics of resistive synaptic devices on implementing convolutional neural networks," *IEEE JETCAS*, vol. 9, pp. 570–579, 2019.

[4] Z. He et al., "Noise injection adaption: End-to-end ReRAM crossbar non-ideal effect adaption for neural network mapping," in *DAC*, 2019.

[5] M. Fieback et al., "Device-aware test: A new test approach towards DPPB level," in *ITC*, 2019.

[6] C. Liu et al., "Rescuing memristor-based neuromorphic design with high defects," in *DAC*, 2017.

[7] A. Krizhevsky et al., "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[8] P. Chi et al., "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," in *ISCA*, 2016.

[9] H. Mao et al., "Exploring the granularity of sparsity in convolutional neural networks," in *CVPR Workshops*, 2017.

[10] J. Lin et al., "Learning the sparsity for ReRAM: Mapping and pruning sparse neural network for ReRAM based accelerator," in *ASPDAC*, 2019.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *Computing Research Repository*, 2014.

[12] S. Rolewicz, *Functional Analysis and Control Theory: Linear Systems*. Springer, 1987.

[13] A. Madry et al., "Towards deep learning models resistant to adversarial attacks," in *ICLR*, 2018.