Fault-Tolerant Deep Neural Networks for Processing-In-Memory based Autonomous Edge Systems

Siyue Wang*[1], Geng Yuan*[1], Xiaolong Ma[1], Yanyu Li[1], Xue Lin[1], and Bhavya Kailkhura[2]
[1]*Dept. of Electrical and Computer Engineering, Northeastern University. Boston, MA, USA.*
[2]*Lawrence Livermore National Laboratory. Livermore, CA, USA.*
{wang.siy, yuan.geng, ma.xiao1, li.yanyu, xue.lin}@northesatern.edu kailkhura1@llnl.gov

***Abstract*—In-memory deep neural network (DNN) accelerators will be the key for energy-efficient autonomous edge systems. The resistive random access memory (ReRAM) is a potential solution for the non-CMOS-based in-memory computing platform for energy-efficient autonomous edge systems, thanks to its promising characteristics, such as near-zero leakage-power and non-volatility. However, due to the hardware instability of ReRAM, the weights of the DNN model may deviate from the originally trained weights, resulting in accuracy loss. To mitigate this undesirable accuracy loss, we propose two stochastic fault-tolerant training methods to generally improve the models' robustness without dealing with individual devices. Moreover, we propose *Stability Score*—a comprehensive metric that serves as an indicator to the instability problem. Extensive experiments demonstrate that the DNN models trained using our proposed stochastic fault-tolerant training method achieve superior performance, which provides better flexibility, scalability, and deployability of ReRAM on the autonomous edge systems.**

***Index Terms*—Fault-Tolerant DNNs, DNN Accelerators, Autonomous Edge Systems**

I. Introduction

Delivering enough computing power, dealing with system redundancy, and providing more accurate decisions are the key elements to guarantee the safety of edge autonomous systems. DNN accelerators, given their potential of low-cost and "in-situ" data processing ability, are gaining increasing research attention and expectations to satisfy the need for the safety of edge autonomous systems in resource-constrained environments. Many research efforts have been made on developing DNN accelerators, including CMOS/non-CMOS based ASICs. Among non-CMOS based accelerators, the emerging resistive random access memory (ReRAM) [1], [2], is promising with near-zero leakage-power, high active-power efficiency, non-volatility, and concise circuit structure. However, one critical challenge is its poor stability. The weights of a DNN can be easily distorted by the inherent physical limitations of ReRAM devices, resulting in accuracy drop. Besides, current solutions only focus on optimizations for individual devices, such as permuting the order of the crossbar rows and columns [3], retraining the network by considering the defect locations [4], [5], which may introducing complex control circuits and resulting in additional hardware overhead. For mass-produced edge products, the efforts and resources spent in applying optimization for each individual product (edge system) can be tremendous, and therefore it is of utmost importance to be able to apply more general solutions that can be widely used to mass-produced products while keeping its effectiveness in mitigating the accuracy drop caused by the stuck-at-fault defects.

To better address the aforementioned stability problem of ReRAM and versatility problem of current approaches, this work investigates and proposes a unique stochastic fault-tolerant training scheme to improve the robustness of DNNs used in ReRAM-based systems. By randomly injecting faults with different levels, we simulate the stuck-at-fault defects during the training in order to increase the robustness of our models against the stability problem while maintaining the performance (e.g., model accuracy), making our fault-tolerant models have better adaptability on mass-produced products. Compared with other conventional methods, the advantages of our solution lie in (i) the robustness improvement is statistically achieved for all models, which avoids conducting optimizations for each individual device; (ii) by injecting faults with different levels, our method can demonstrate better trade-off between robustness and accuracy which provide better flexibility, scalability, and deployability; (iii) our method can tolerate almost an order of magnitude higher failure rate than the other methods in representative DNN tasks; and (iv) our method intends to address the stability problem without involving expensive retraining and optimizations for different devices once the robust model is trained.

Besides, the DNN weight pruning techniques, which can effectively reduce the model size and computation costs, have been widely used for efficient DNN system designs [6]–[10]. The state-of-the-art pruning algorithms [11], [12] can successfully maintain the model accuracy under a high sparsity ratio. However, as the model's sparsity increases, the model's fault tolerance will decrease, indicating a more significant accuracy drop will happen when the defects are introduced to the model [13]. Therefore, in this work, we also investigate the impact of weight pruning on the model's fault tolerance. And we find that our proposed stochastic fault-tolerant training methods can also significantly improve the fault tolerance of the pruned models in general.

Below we summarize our main contributions and advantages.

*These authors contributed equally.

978-3-9819263-6-1/DATE22/©2022 EDAA

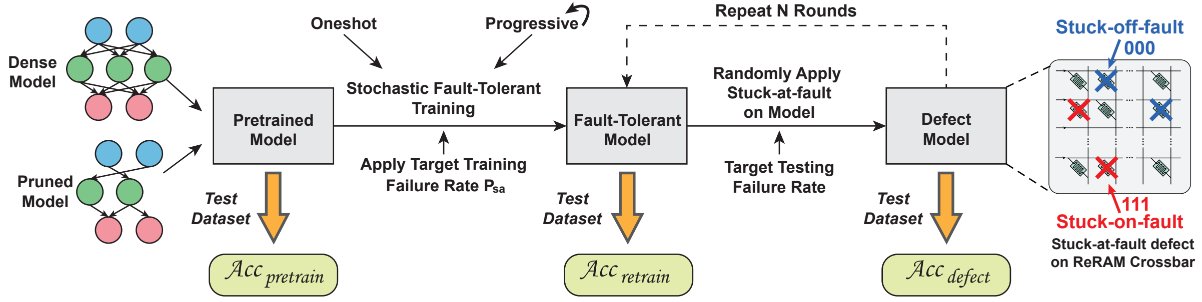

Fig. 1. System Overview of a Fault-Tolerant DNN for ReRAM based Systems.

- We propose a novel and practical stochastic fault-tolerant training scheme that improves the robustness of DNNs used in ReRAM-based edge systems without focusing on optimizations for individual devices. In particular, we develop two kinds of stochastic fault-tolerant training methods: *one-shot stochastic fault-tolerant training* and *progressive stochastic fault-tolerant training* to further improve the DNN fault-tolerant performance.
- To better evaluate the trade-off between model robustness and accuracy on ReRAM, we propose a novel metric called *Stability Score (SS)* that quantifies the effectiveness of DNNs' fault tolerance. We are the first work that thoroughly considers the scalability and robustness-accuracy trade-off in designing robust models for ReRAM-based systems. Our mechanisms featuring high-scalability and better trade-off between model fault tolerance and its accuracy degradation can greatly improve the current stability and versatility problem for current ReRAM-based applications.
- Compared with models without applying stochastic fault-tolerant training, experiments on CIFAR-100 dataset demonstrate that both the pretrained and pruned models exhibits much stronger resiliency to stack-at-fault defects and simultaneously sacrifices less accuracy. Moreover, the proposed training schemes is an easy approach to generally increase the model fault tolerance without causing extra computations for device specific optimizations.

II. Background

A. ReRAM-based DNN Accelerator

Resistive RAM (ReRAM) [1], [2] is a emerging technology that has attracted great attention in recent years. It has many promising characteristics, including non-volatility, nearly zero leakage power, high integration density, and high scalability. The ReRAM cells can form an array structure [14], as known as the ReRAM based crossbar, that is capable of performing in-situ dot product between an input vector (i.e., input feature maps) and stored numbers (i.e., weights). Several ReRAM-based in-situ DNN accelerator architectures [15]–[21] have been proposed in recent years, which employ various techniques for in-memory processing to minimize data movement to provide significant energy savings and performance improvements.

B. Stability Problem of ReRAM-based DNNs

The hardware failure is a challenging issue of the ReRAM-based DNN system designs that may significantly degrade the accuracy of the neural network when deployed on ReRAM devices [3], [4]. Some prior works propose to detect and model the faults of ReRAM crossbars, such as using a fault model with testing scheme [22], a march algorithm to cover defect [23], and analyzing impacts of stuck-at-fault defects on accuracy of a sparse coding network [24].

To mitigate the impact of hardware failure, several fault-tolerant methods are proposed. In [25], they propose to isolate the faulty ReRAM by switching off the access transistor. But this will introduce considerable routing and area overheads, especially for the large ReRAM crossbars used by deep neural networks. Another solution is to use redundant columns of ReRAM crossbars as the alternative to the defect ReRAM columns [4]. This kind of method still introduces nontrivial area overhead and increases the design complexity of the peripheral circuit. There are also some software level solutions [5], which propose to retrain the neural network by forcing the weights to match the fault values on the corresponding crossbar locations. However, this method is device-specific, which means the time-consuming retraining and remapping process is required for each ReRAM device. Obviously, it is not desired or even not practical for the mass-produced edge products.

III. Improving DNN Fault Tolerance with Stochastic Training Scheme

In this section, we propose our stochastic training scheme to improve the robustness of our models against the instability problem while maintaining model accuracy. The overall flow is shown in Figure 1. we consider the following scenarios that may change the model accuracy: ① $Acc_{pretrain}$: the ideal accuracy of pretrained models without introducing any stuck-at-fault defects on weights. ② $Acc_{retrain}$: the ideal accuracy of retrained fault-tolerant models using the proposed stochastic training schemes. Here, the model is tested without incurring stuck-at-faults on weights, reflecting the impact of the stochastic fault-tolerant training on the accuracy of the pretrained model. ③ Acc_{defect}: the accuracy of DNN models with the weights under stuck-at-faults defects, indicating the model performance on the actual devices.

Our final target is to narrowing the gap between $Acc_{pretrain}$ and Acc_{defect}, where in most cases, there is usually a huge gap due to the instability of ReRAM devices. In order to achieve this goal, we need fault-tolerant models which has appealing accuracy $Acc_{retrain}$.

Algorithm 1: Stochastic Fault-Tolerant Training.

```
1  Initialization:
2  D: dataset,
3  F(w, ·) : well-trained model with Acc_1,
4  P_sa : random stack-at-fault ratio added on to the model
     during training phase,
5  P_sa_list = [P_sa^0, P_sa^1, P_sa^2, ..., P_sa^T], candidate
     stack-at-fault ratios in ascending order, where the P_sa^T
     is the final target stack-at-fault ratio for the stochastic
     fault-tolerant training,
6  w_p : trained weights of model after stochastic
     fault-tolerant training,
7  M_epoch : total number of training epochs,
8  num_of_runs : total number of runs in testing phase.
9  Training:
10 if One-Shot Stochastic Fault-Tolerant Training then
11   | for ep in M_epoch do
12   |   | w' ← Apply_Fault(w, P_sa^T);
13   |   | Forward_Prop();            // on F(w', D)
14   |   | Back_Prop();
15   |   | Update_Weights(w);
16   | end
17   | w_p ← w
18 end
19 if Progressive Stochastic Fault-Tolerant Training then
20   | for P_sa^i in P_sa_list do
21   |   | for ep in M_epoch do
22   |   |   | w' ← Apply_Fault(w, P_sa^i);
23   |   |   | Forward_Prop();        // on F(w', D)
24   |   |   | Back_Prop();
25   |   |   | Update_Weights(w);
26   |   | end
27   | end
28   | w_p ← w
29 end
30 return F(w_p, x)     // with validation accuracy of Acc_2
31 Testing:
32 Acc_sum ← 0
33 for num_of_runs do
34   | w' ← Apply_Fault(w, P_sa^i);
35   | Acc_sum ← Acc_sum + Accuracy(F(w', D));
36 end
37 Acc_3 ← Acc_sum/num_of_runs;
38 return Acc_3
```

A. Proposed Stochastic Fault-Tolerant Training Scheme

Facing the poor stability problem of ReRAM based DNN accelerators, we propose the following stochastic training algorithm that can appropriately simulate the stuck-at-fault defects during training in order to increase the robustness of our models. Meanwhile, we also consider simulating the stuck-at-fault defects for different devices by randomly injecting faults with different levels.

Our main idea is to simulate the stuck-at-fault defects in the model retraining process. More specifically, we introduce a novel "stochastic model retraining scheme" that fuses the model weights with randomly injected faults. According to the ReRAM stuck-at-fault model reported by C. Chen *et al.* [**?**], the probability of stuck-on-fault and stuck-off-fault of a ReRAM device is different. We denote the overall stuck-at failure rate as $P_{sa} = P_{sa0} + P_{sa1}$, where the P_{sa0} and P_{sa1} represent the probability of stuck-off-fault and stuck-on-fault, respectively.

We denote the original pre-trained neural network as $F(\mathbf{w}, \mathbf{x})$ where $\mathbf{w} \in \mathbb{R}^{d_\mathbf{w}}$ is the weights and $\mathbf{x} \in \mathbb{R}^{d_\mathbf{x}}$ is the input image. With the stuck-at-fault rate P_{sa}, the representation of the model on the ReRAM crossbar can be expressed as $F(\mathbf{w}_p, \mathbf{x})$.

In the process of pursuing a better trade-off between model fault tolerance and accuracy and satisfying all the system requirements, we propose two kinds of stochastic fault training methods: *one-shot stochastic fault-tolerant training* and *progressive stochastic fault-tolerant training*, which both methods will be evaluated and discussed in our experiments. The training approach is summarized in Algorithm 1.

Our proposed *one-shot stochastic fault-tolerant training* uses a fixed target stack-at-fault ratio P_{sa}^T during training, providing an easy and efficient approach to the fault-tolerant model. Whereas the *progressive stochastic fault-tolerant training* boosts the model fault-tolerant performance by gradually increasing the stack-at-fault ratio in an iterative training manner. This makes the model have better adaptability to larger target stack-at-fault ratio P_{sa}^T, leading to higher ideal accuracy to fault-tolerant model and better performance facing stack-at-fault defects. In real scenarios, the edge system designer can choose from the two stochastic fault-tolerant training methods according to the system requirements.

B. Stability Score (SS): Quantifying Model Fault Tolerance & Accuracy Trade-off

For most current methods on improving model fault tolerance on ReRAM-based crossbar, there are factors controlling the fault-tolerant strength. For example, in previous work that utilizes unstructured pruning to improve fault tolerance of the ReRAM-based DNNs [13], one can achieve different fault-tolerant strength by pruning the model with different pruning ratios. Analogously, for stochastic fault-tolerant training, the controlling factors are the randomization schemes, such as the stuck-at-fault rate P_{sa}, different training schemes (one-shot training and progressive training) in our proposed approach.

We note that for conventional methods, although the fault-tolerant of the model can be improved by using a stronger defense controlling factor at first, it is traded by either sacrificing the test accuracy of the model on clean examples or extra computational resources. The fault-tolerant promotion tends

to plateau when the controlling factors are made increasingly stronger while the cost tends to amplify.

In this paper, we want to study the trade-off between model fault tolerance (represented by Acc_{defect}) and its upper-bound without incurring stack-at-faults defects (represented by $Acc_{retrain}$). Specifically, as we are pursuing a solution that can provide an easy and more general approach, we are interested in not only the fault tolerance performance of the model, but also the general performance of the model accuracy. It is worth noting that even facing the same stack-at-fault defects, comparing the fault-tolerant performance of different models is not an easy task as they vary in both $Acc_{retrain}$) and Acc_{defect}). In order to tackle this difficulty, we propose *Stability Score (SS)* as:

$$SS_{Acc}(P_{sa}) = Acc_{retrain}/(Acc_{pretrain} - Acc_{defect}). \quad (1)$$

Given a target testing failure rate on ReRAM crossbar, a higher *SS* for a model represents less accuracy degradation from the ideal accuracy facing stack-at-fault defects while having the ability to maintain an appealing retrained accuracy. We summarize our results for stability evaluations in Section IV-C.

IV. Evaluation

A. Experimental Setup

In this section, we evaluate the effectiveness of our proposed stochastic fault training methods in improving general model fault tolerance. We conduct experiments on the ResNet networks [26], which are widely used as the benchmark network and adopted as the backbone network in different computer vision tasks such as object detection for autonomous driving. In specific, we use ResNet-20 on CIFAR-10 dataset and ResNet-32 on CIFAR-100 dataset. We evaluate both the dense models and pruned models under different pruning ratios.

All our models are trained on a GPU server with 8× NVIDIA 2080Ti GPUs. The PyTorch framework is used for model training. We follow the standard training recipe with 160 training epochs for all the training process including the model pretraining, stochastic fault-tolerant training, and pruned model fine-tuning. We use the cosine learning rate scheduler with initial learning rate of 0.1. We adopt the ReRAM stuck-at-fault model reported in [23], and set a fixed ratio for the stuck-off-fault rate P_{sa0} and stuck-on-fault rate P_{sa1} as $1.75 : 9.04$. To simulate the defect models, we randomly apply stuck-at-fault on the trained model weights with target failure rate. The defect model accuracy is the average accuracy of 100 runs (i.e., repeating applying defects on a given model).

B. Results and Analysis of the Proposed Stochastic Fault-Tolerant Training Methods

Table I shows the accuracy comparison of the fault-tolerant models obtained using the proposed one-shot stochastic fault-tolerant training method and progressive fault-tolerant training method, respectively. We train the models using different target training failure rate (P^T_{sa}) and evaluate the defect models under different target testing stuck-at-fault failure rates. First of all, we can observe that our proposed stochastic fault-tolerant training methods can significantly improve the model robustness to the stuck-at-fault defects. The baseline pretrained models, which are not trained by any of our stochastic fault-tolerant training, drops about 2.5% accuracy (Acc_{defect}) under a failure rate of 0.001 (i.e., 0.1%), while most of fault-tolerant models maintains the similar accuracy or even higher accuracy under the same failure rate. On CIFAR-10, our best-performance fault-tolerant model can tolerant a failure rate up to 0.005 and preserves the same accuracy as the ideal model accuracy ($Acc_{pretrain}$), where the baseline pretrained model only has 61.74% accuracy. With 1% accuracy degradation allowed, our fault-tolerant model can tolerant a failure rate up to 0.01, where the baseline pretrained model accuracy is as low as 25.83%. We can find similar results on the CIFAR-100 dataset, where the fault-tolerant model can tolerant a failure rate up to 0.003 without accuracy drop and up to 0.005 with 1% accuracy drop, respectively.

Besides, we can also observe that the ideal accuracy (under failure rate of 0) of the fault-tolerant model ($Acc_{retrain}$) can be improved by both of the proposed stochastic fault-tolerant training methods compared to the ideal accuracy of baseline pretrained model. The reason is that the stochastic fault-tolerant training methods can mitigate the over-fitting issue and improve the model generalization ability.

In table I, we highlight the top-3 highest fault-tolerant model accuracy (Acc_{defect}) under each target testing failure rate, and we have two important observations: ① when given a target testing failure rate, it is more desired to use a larger target training failure rate in stochastic fault-tolerant training to deliver a better fault-tolerant model with higher accuracy (Acc_{defect}). ② The target training failure rate should not be too much larger than the target testing failure rate. For example, the best-suited target training failure rates are 0.02, 0.05, and 0.1 for the both of the cases on CIFAR-10 and CIFAR-100 dataset that the target testing failure rate is 0.005, and both smaller or larger target training failure rates will lead to accuracy degradation. Thus, suppose the target devices for deployment have a relatively smaller failure rate. In this case, we should not use a very large target training failure rate in the stochastic fault-tolerant training, which helps preserve higher accuracy and save training time. By comparing the results of the one-shot fault-tolerant training and the progressive fault-tolerant training, we can find that the later method generally provides higher accuracy of fault-tolerant models, especially under a relatively higher failure rate.

C. Stability Evaluations

In this section, we explore the stability of different models under stack-at-fault defects. In most cases, for ReRAM-based autonomous edge systems, due to the limited computational resources, we often apply the weight pruning technique to the DNN models before its deployment. Thus, we are interested in the stability performance of not only the pretrained models, but also pruned models with our proposed stochastic fault-tolerant training. Figure 2 shows the accuracy of pretrained dense model and its two pruned versions (one-shot pruning method [27] and

TABLE I
ACCURACY OF IMPLEMENTED FAULT-TOLERANT MODELS BY DIFFERENT P_{sa}^{T} UNDER DIFFERENT TARGET TESTING FAILURE RATE ON CIFAR-10 AND CIFAR-100 DATASETS, RESPECTIVELY.

Method & Training Stack-At-Fault Rate	Target Testing Stack-At-Fault Failure Rate													
/	0	0.001	0.0015	0.002	0.003	0.005	0.01	0.02	0.03	0.05	0.075	0.1	0.15	0.2
CIFAR-10 Dataset, ResNet-20 (pretrained model accuracy = 92.53%)														
Baseline Pretrained Model	92.53	90.17	87.82	85.25	78.06	61.74	25.83	10.75	9.94	9.69	10.04	10.14	10.06	9.93
One-Shot P_{sa}^{T} = 0.005	92.88	**92.65**	**92.40**	**92.24**	91.50	89.97	85.18	71.94	54.83	25.98	17.18	12.03	10.58	10.01
Progressive P_{sa}^{T} = 0.005	92.99	92.63	92.38	92.26	91.56	90.02	85.84	71.86	55.09	29.14	17.55	12.83	10.59	10.52
One-Shot P_{sa}^{T} = 0.01	92.94	**92.76**	**92.51**	**92.36**	**91.90**	90.90	87.43	79.47	68.15	41.73	22.05	16.02	11.75	10.80
Progressive P_{sa}^{T} = 0.01	92.90	**92.86**	**92.71**	**92.51**	**92.09**	91.04	87.40	79.37	66.79	39.29	24.28	15.74	11.47	9.99
One-Shot P_{sa}^{T} = 0.02	92.38	**92.36**	**92.18**	**92.08**	**91.69**	**91.42**	89.20	84.71	76.68	55.19	33.70	20.25	13.85	11.06
Progressive P_{sa}^{T} = 0.02	93.24	**93.03**	**92.95**	**92.75**	**92.45**	**91.91**	89.38	83.30	74.73	50.81	31.48	20.42	13.07	11.07
One-Shot P_{sa}^{T} = 0.05	92.41	92.24	92.15	92.00	**91.77**	**91.38**	**90.07**	84.86	80.49	63.99	46.97	26.82	15.49	12.26
Progressive P_{sa}^{T} = 0.05	93.01	**92.77**	**92.78**	**92.62**	**92.06**	**91.69**	**90.93**	85.04	79.79	62.24	42.16	26.93	16.51	12.06
One-Shot P_{sa}^{T} = 0.1	92.42	92.20	92.08	92.02	91.67	**91.58**	**90.53**	**88.02**	**85.16**	**74.86**	**54.37**	**35.88**	**18.92**	**13.96**
Progressive P_{sa}^{T} = 0.1	93.29	92.68	92.59	92.32	92.06	**92.52**	**91.34**	**89.03**	**86.11**	**75.08**	**57.14**	**39.84**	**21.37**	**14.57**
One-Shot P_{sa}^{T} = 0.15	91.91	91.81	91.74	91.70	91.54	91.27	**90.57**	**88.20**	**85.26**	**76.34**	**58.09**	**40.18**	**21.59**	**14.96**
Progressive P_{sa}^{T} = 0.15	92.79	92.61	92.37	92.29	92.21	91.87	**90.89**	**88.62**	**85.49**	**76.72**	**59.48**	**45.84**	**23.84**	**15.37**
One-Shot P_{sa}^{T} = 0.2	91.52	91.36	91.28	91.20	91.04	90.84	90.04	**88.24**	**85.49**	**76.92**	**61.21**	**44.21**	**23.14**	**14.17**
Progressive P_{sa}^{T} = 0.2	92.03	92.37	92.33	92.24	92.07	91.62	90.85	**88.82**	**86.87**	**77.87**	**63.67**	**49.29**	**26.65**	**15.78**
CIFAR-100 Dataset, ResNet-32 (pretrained model accuracy = 75.10%)														
Baseline Pretrained Model	75.10	69.28	65.75	60.66	47.15	25.83	2.97	0.99	1.01	0.99	1.00	1.01	1.00	1.00
One-Shot P_{sa}^{T} = 0.005	75.28	74.82	74.51	74.29	73.75	72.65	69.08	59.91	48.45	21.94	7.95	3.57	1.45	1.17
Progressive P_{sa}^{T} = 0.005	75.57	75.14	74.91	74.60	74.12	72.94	69.31	60.16	48.70	26.42	8.95	3.58	1.52	1.15
One-Shot P_{sa}^{T} = 0.01	75.23	**74.98**	**74.83**	**74.64**	74.16	73.31	70.83	65.09	57.00	36.68	15.62	7.05	2.27	1.50
Progressive P_{sa}^{T} = 0.01	75.48	**75.43**	**75.18**	**75.00**	74.51	73.63	71.18	65.17	56.75	37.51	16.96	7.24	2.28	1.53
One-Shot P_{sa}^{T} = 0.02	75.38	**75.27**	**75.16**	**74.98**	**74.69**	**74.09**	72.18	67.78	61.62	46.39	25.74	12.04	3.37	1.81
Progressive P_{sa}^{T} = 0.02	75.78	**75.56**	**75.41**	**75.20**	**74.84**	**74.09**	72.23	68.08	62.45	47.00	26.43	13.11	3.73	1.78
One-Shot P_{sa}^{T} = 0.05	75.38	**75.19**	**75.04**	**74.99**	**74.74**	**74.30**	**73.03**	**70.01**	**66.34**	55.64	36.58	20.07	5.51	2.26
Progressive P_{sa}^{T} = 0.05	75.56	**75.40**	**75.33**	**75.21**	**74.99**	**74.52**	**73.30**	**70.46**	66.56	55.25	38.50	22.31	6.27	2.74
One-Shot P_{sa}^{T} = 0.1	74.96	74.63	74.54	74.49	**74.24**	**73.91**	**72.84**	**69.93**	**66.50**	**56.84**	**40.08**	**24.10**	**6.72**	**2.76**
Progressive P_{sa}^{T} = 0.1	74.63	74.77	74.69	74.64	**74.53**	**74.15**	**73.18**	**70.61**	**67.78**	**59.09**	**44.12**	**28.64**	**8.85**	**3.82**
One-Shot P_{sa}^{T} = 0.15	74.20	74.74	74.64	74.53	74.13	73.89	**72.87**	**70.34**	**67.16**	**58.37**	**43.22**	**26.62**	**7.84**	**2.72**
Progressive P_{sa}^{T} = 0.15	75.05	74.96	74.85	74.76	74.37	74.08	**73.30**	**70.93**	**68.10**	**60.05**	**45.82**	**30.41**	**10.06**	**3.83**
One-Shot P_{sa}^{T} = 0.2	73.87	73.71	73.63	73.53	73.39	73.00	71.91	69.15	65.80	**57.51**	**40.85**	**27.36**	**7.46**	**3.21**
Progressive P_{sa}^{T} = 0.2	74.63	74.59	74.48	74.38	74.23	73.77	72.76	70.34	**67.18**	**58.77**	**44.71**	**31.19**	**9.70**	**4.13**

ADMM-based pruning method [12], respectively) with 40% and 70% sparsity under different testing failure rates without stochastic fault-tolerant training. We can clearly observe that without stochastic fault-tolerant training, a severe accuracy drop happens when a relatively low failure rate is introduced, especially on the CIFAR-100 dataset. With the increase of pruning ratios, the model with higher sparsity tends to have a more severe accuracy drop. What we can also observe is that: for models with the same sparsity level, there is little difference in their fault-tolerance performance. So we will only present the stability evaluations for ADMM pruned models as it is more popular for autonomous edge systems. To characterize the model fault tolerance and accuracy trade-offs of different models, we compare the *SS* (see Section III-B) of different models under two selected target testing failure rates in Table II.

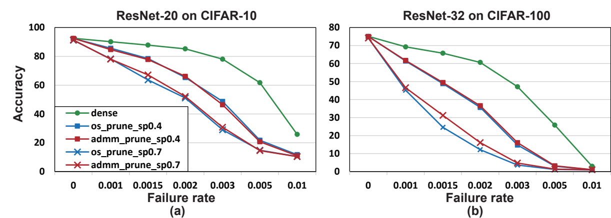

Fig. 2. Accuracy of dense model and pruned model (without stochastic fault-tolerant training) under different failure rates.

We summarize our findings from Table II as follows:

1) In general, the proposed stochastic fault-tolerant training achieves superior *SS* performance under all settings. Given 0.01 target testing failure rate, compared with the pre-trained ResNet-32 model, even applying the target training stack-at-fault ratio $P_{sa}^{T} = 0.01$ will dramatically resulted in $16.7\times$ better SS score for one-shot training scheme and $18.5\times$ better *SS* score for progressive training scheme, respectively.
2) The advantage of stochastic fault-tolerant training over the pretrained and pruned models becomes more apparent using larger target training stack-at-fault ratio P_{sa}^{T}. For example, given the ADMM pruned ResNet-32 model as the pretrained model, model trained with progressive fault-tolerant training using $P_{sa}^{T} = 0.01$ achieves 3.54 *SS* under 0.01 target testing failure rate, while model trained with $P_{sa}^{T} = 0.1$ gives over $2X$ *SS* that of model trained with $P_{sa}^{T} = 0.01$.
3) With same retrained accuracy of 73.03%, we observe that the progressive training scheme outperforms the one-shot training scheme, attributing to higher Acc_{defect}.
4) Compared with dense models, the accuracy of pruned models are more likely to be influenced by the stack-at-fault defects, making it more important to apply the stochastic fault-tolerant training in order to boost the model performance.

TABLE II
ACCURACY AND STABILITY SCORE (SS) OF IMPLEMENTED FAULT-TOLERANT MODELS DERIVED FROM THE PRETRAINED AND ADMM PRUNED RESNET-32 MODELS UNDER DIFFERENT TARGET TESTING FAILURE RATE ON CIFAR-100 DATASET.

Methods	$Acc_{pretrain}$	$Acc_{retrain}$	Acc_{defect} (0.01)	Acc_{defect} (0.02)	SS (0.01)	SS (0.02)
Pretrained ResNet-32 Model (accuracy = 75.10%)						
/	75.10	75.10	2.97	0.99	1.04	1.01
One-Shot $P_{sa}^{T} = 0.01$	75.10	75.23	70.83	65.09	17.62	7.52
One-Shot $P_{sa}^{T} = 0.05$	75.10	75.38	73.03	70.01	36.42	14.81
One-Shot $P_{sa}^{T} = 0.1$	75.10	74.96	72.84	69.93	33.17	14.50
Progresive $P_{sa}^{T} = 0.01$	75.10	75.48	71.18	65.17	19.26	7.60
Progresive $P_{sa}^{T} = 0.05$	75.10	75.75	73.03	70.46	36.59	16.33
Progresive $P_{sa}^{T} = 0.1$	75.10	74.63	73.18	70.61	**38.87**	**16.62**
ADMM Pruned ResNet-32 Model with 70% Sparsity (model accuracy = 74.89%)						
/	74.89	74.89	1.00	0.99	1.01	1.01
One-Shot $P_{sa}^{T} = 0.01$	74.89	75.01	52.53	24.99	3.35	1.50
One-Shot $P_{sa}^{T} = 0.05$	74.89	74.40	61.87	44.24	5.71	2.43
One-Shot $P_{sa}^{T} = 0.1$	74.89	74.30	64.61	50.92	7.23	3.10
Progresive $P_{sa}^{T} = 0.01$	74.89	74.93	53.71	26.03	3.54	1.53
Progresive $P_{sa}^{T} = 0.05$	74.89	74.73	62.07	44.49	5.83	2.46
Progresive $P_{sa}^{T} = 0.1$	74.89	74.7	65.37	52.00	**7.85**	**3.26**

In addition to the superior stability performance of our proposed methods, it is worth emphasizing that our proposed methods can generally improve the model fault tolerance without the extra cost of the retraining process for each device. Our method with significantly higher *SS* is also compatible with prior methods such as using error correction output code [28] or other device-specific optimizations [4], [5], [25], [29].

V. CONCLUSIONS

Towards addressing the instability problem and achieving better trade-off between fault tolerance and accuracy of models deployed on the ReRAM-based systems, we design two kinds of stochastic fault-tolerant training schemes that can generally improve the model fault tolerance while preventing accuracy drop. To fairly characterize the trade-off between fault tolerance and accuracy, we propose an evaluation metric named Stability Score, which can be used as an indicator for the system designers. Extensive experiments demonstrate that the proposed methods have superior performance for both pretrained and compressed models in achieving better fault-tolerant models with less accuracy degradation facing the stack-at-fault defects.

ACKNOWLEDGEMENTS

This work was conducted with the computing facilities acquired with National Science Foundation (NSF) Award CNS-1920020. This work was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 and LLNL LDRD Program Project No. 20-SI-005.

REFERENCES

[1] Y. Ho, G. M. Huang, and P. Li, "Nonvolatile memristor memory: Device characteristics and design implications," in *2009 IEEE/ACM International Conference on Computer-Aided Design Digest of Technical Papers*, 2009.
[2] A. G. Radwan, M. A. Zidan, and K. N. Salama, "HP Memristor mathematical model for periodic signals and DC," in *2010 53rd IEEE International Midwest Symposium on Circuits and Systems*, aug 2010.
[3] L. Chen *et al.*, "Accelerator-friendly neural-network training: Learning variations and defects in rram crossbar," in *DATE*. IEEE, 2017.
[4] C. Liu *et al.*, "Rescuing memristor-based neuromorphic design with high defects," in *DAC*. IEEE, 2017.
[5] L. Xia *et al.*, "Fault-tolerant training with on-line fault detection for rram-based neural computing systems," in *DAC*, 2017.
[6] Y. Cai *et al.*, "Yolobile: Real-time object detection on mobile devices via compression-compilation co-design," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2021.
[7] H. Li *et al.*, "Real-time mobile acceleration of dnns: From computer vision to medical applications," in *ASP-DAC*, 2021.
[8] G. Yuan *et al.*, "A dnn compression framework for sot-mram-based processing-in-memory engine," in *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*, 2020.
[9] X. Ma *et al.*, "Sanity checks for lottery tickets: Does your winning ticket really win the jackpot?" *Advances in Neural Information Processing Systems*, vol. 34, 2021.
[10] G. Yuan *et al.*, "Mest: Accurate and fast memory-economic sparse training framework on the edge," *Advances in Neural Information Processing Systems*, vol. 34, 2021.
[11] S. Guo *et al.*, "Dmcp: Differentiable markov channel pruning for neural networks," in *CVPR*, 2020.
[12] T. Zhang *et al.*, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *ECCV*, 2018.
[13] G. Yuan *et al.*, "Improving dnn fault tolerance using weight pruning and differential crossbar mapping for reram-based edge ai," in *ISQED*, 2021.
[14] M. Hu *et al.*, "Hardware realization of bsb recall function using memristor crossbar arrays," in *DAC*, 2012.
[15] A. Shafiee *et al.*, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *ISCA*, 2016.
[16] G. Yuan *et al.*, "Forms: Fine-grained polarized reram-based in-situ computation for mixed-signal dnn accelerator," in *ISCA*, 2021.
[17] A. Nag *et al.*, "Newton: Gravitating towards the physical limits of crossbar acceleration," 2018.
[18] G. Yuan, P. Behnam *et al.*, "Tinyadc: Peripheral circuit-aware weight pruning framework for mixed-signal dnn accelerators," in *DATE*, 2021.
[19] A. Ankit *et al.*, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *ASPLOS*, 2019.
[20] G. Yuan *et al.*, "An ultra-efficient memristor-based dnn framework with structured weight pruning and quantization using admm," in *ISLPED*. IEEE, 2019.
[21] X. Ma *et al.*, "Tiny but accurate: A pruned, quantized and optimized memristor crossbar framework for ultra efficient dnn implementation," in *ASP-DAC*, 2020.
[22] S. Kannan *et al.*, "Modeling, detection, and diagnosis of faults in multilevel memristor memories," *IEEE T-CAD*, 2015.
[23] C.-Y. Chen *et al.*, "Rram defect modeling and failure analysis based on march test and a novel squeeze-search scheme," *IEEE Transactions on Computers*.
[24] P. Sheridan and W. D. Lu, "Defect consideratons for robust sparse coding using memristor arrays," in *IEEE/ACM NANOARCH*. IEEE, 2015.
[25] H. Manem *et al.*, "Design considerations for variation tolerant multilevel cmos/nano memristor memory," in *Great lakes symposium on VLSI*, 2010.
[26] K. He and et al., "Deep residual learning for image recognition," in *CVPR*, 2016.
[27] S. Han *et al.*, "Learning both weights and connections for efficient neural network," in *Advances in NeurIPS*, 2015.
[28] T. Liu *et al.*, "A fault-tolerant neural network architecture," in *DAC*, 2019.
[29] F. S. Hosseini *et al.*, "Tolerating defects in low-power neural network accelerators via retraining-free weight approximation," *ACM Transactions on Embedded Computing Systems (TECS)*, 2021.