TRAINING FAULT-TOLERANT MODEL WITH ADVERSARIAL ATTACK GENERATED PATTERN

Min-Hsin Liu, Chu-Yun Hsiao

1. INTRODUCTION

In recent years, machine learning has become a state-of-the-art technique in many fields. Including computer vision, natural language processing, robotic, and even healthcare. Consequently, an increasing number of products are being equipped with embedded hardware to support machine learning purpose. Due to the limited power resources of nowadays edge devices, neuromorphic circuits are considered one of the promising neural network architectures because of the low power by in-memory computation.

However, while training and performing inference on devices, we cannot assume the structure to be fault-free. There are several reasons that might cause the network to be faulty.

1. Manufacturing imperfection and process variation might introduce unexpected faults.
2. The reliability of the devices would drop over time because of limited read and write operations and equipment aging.
3. Even though the neural networks are inherently fault-tolerant, the defects at some critical locations might lead to severe accuracy drop if it lacks corresponding design. Thus, considering faults before production becomes a crucial topic.

In [5], the author proposed a fault-tolerant training method to enhance the accuracy under faulty scenario and maintain a comparable accuracy under fault-free scenario. However, for some fault models, the accuracy would still drop severely compared to fault-free scenario. Besides, the fault tolerant training might need large epoch number to meet required accuracy, costing lots of time. Enhancing the accuracy and improving the convergence speed may be two possible improvements. Thus, our proposed techniques aims to achieve these goals. The results of our proposed technique guarantee the better accuracy compared with [5] in the presence of faults under the same environment setup. Besides, it also achieves approximately the same accuracy with less epochs.

Section 2 reviews previous researches related to our work. Section 3 introduce our proposed method. Section 4 demonstrates the experimental results. Section 5 discusses the problem we observe while implementing. Section 6 concludes our works.

2. BACKGROUND

This section covers three topics. First, the spiking neural network, which is the neural network our proposed method is based on in Section 2A. Second, the introduction of adversarial attack and pattern generation based on this concept will be presented in Section 2B. Third, the fault-tolerant training, our proposed method, is developed based on the foundation of this work, and we will be comparing our results with this method.

A. Spiking Neural Network (SNN)

A spiking neural network is a type of neural network inspired by the mechanisms of biological neurons. Unlike traditional neural networks, SNNs are activated by spikes that hold time information. Thus, the input dataset is encoded to perform in spike form. SNNs consist of two kinds of components: synapses and neurons. We introduce a neural model called Leaky-Integrate-and-Fire (LIF), which is illustrated in Fig. 1[6]. Each synapse holds a weight, and spikes passed through it are multiplied by the weight. Neurons process spikes passed from input synapses and decide whether to activate the output synapse or not. If the sum of the inputs exceeds the threshold, the neuron is activated and creates spikes. Otherwise, the neuron does not create spikes, and the effect of input spikes decays over time.

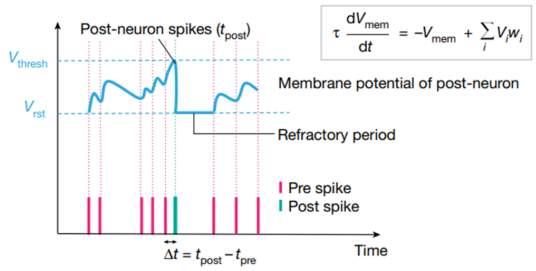


Fig. 1: Leaky-Integrate-and-Fire model illustration[6]

B. Adversarial Attack Pattern Generation

Adversarial attacks refer to deceiving a machine learning model by adding noise to the input pattern. The noise added is imperceptible to humans, but it causes the model to output incorrect answers. Pattern generation based on adversarial

attacks involves adding noise to the original patterns in order to induce discrepancies between faulty and fault-free models' outputs. These new patterns can mislead the model and are subsequently added to the training dataset. The purpose is to enhance dataset diversity and create a more robust model.

C. Fault-Tolerant Training

The section introduces fault-tolerant training as the basis for the work. It is observed that the accuracy of the model drops significantly in a faulty scenario. The motivation is to train a fault-tolerant model. The method proposed by [5] trains a model that has been injected with faults. The results showed that the model trained using this method performs better than the model that has not been injected with faults under the faulty scenario. As for fault-free scenario, the proposed model demonstrates an acceptable decrease in accuracy, considering the significant improvement in the faulty scenario.

3. PROPOSED METHOD

A. Overview

This section consists of four part. Section A gives an overview of the proposed method. The goal of this paper is to improve the convergence speed of fault-tolerant model training. The training process is illustrate by Fig. 2.

At the beginning of our process, a faulty model is trained with a initial dataset generate by MNIST. If the accuracy of the trained model meet a accuracy target we set, stop the process. Else, we generate new patterns using the concept of adversarial attack. The generated new pattern is added to training dataset and the model is trained again by new dataset. Implementation details of model training and pattern generation will be mentioned in section 3B and 3C. To evaluate our method performance, we choose two method to compare with. The details is mentioned in section 3D.

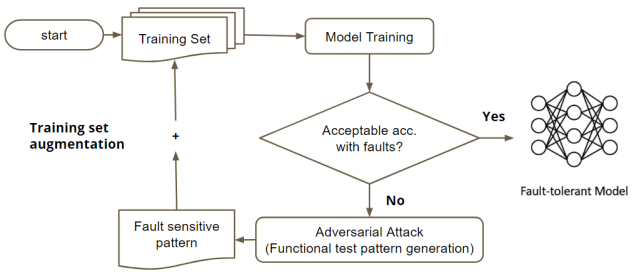


Fig. 2: generate pattern flow chart

B. Fault-Tolerant Training

The fault-tolerant model is trained by injecting fault into model. First, get a batch from training dataset, inference and get the fault-free result. Second, insert fault, inference and get the faulty result. Third, recover fault and update model parameters based on faulty and fault-free output gradient. Move on to next batch until all batch is used. Loss function is calculated with faulty and fault-free by cross entropy.

C. Pattern Generation

Pattern generation is based on the concept of adversarial attack. Dataset and a model is passed into generate pattern function. Pattern is generated by adding noisy to original data, which aims to distinguish model is faulty or fault-free.

The flow of the pattern generation is mentioned in Algorithm 1. There are three loops in the process. The most outer loops ends when one of the two condition meets:

1. All batch have been used to generate new patterns.(ln.1)
2. The amount of the new patterns has exceeds M, a parameter we set to control the amount of patterns generated per process.(lns.17-19)

The second loop is the inference time. The time perturbation will be add to original pattern to generate new pattern. It is limit to N balancing between the run time and new pattern amount. The inference process is similar to training process. We also calculate loss.

$$loss = \sum (ft\_output - ff\_output)^2$$

The difference is no back propagation is done. The gradient is not used for update model parameter but to calculate perturbation.

The most inner loop is designed to adapt to stochastic property of SNN model. If the condition(faulty output $\neq$ fault-free) is met enough times, we believed that the result,pattern is able to distinguish faulty and fault-free model, is trustable.

The good_inference(ln.5) means to get the output of the model without injection fault. The bad_inference(ln.6) on the other side means to get the output of model that has been injected faults. To determine whether faulty and fault-free is the same, the average of repetition is first calculated for faulty and fault-free result. Then, we compare the max value of two output, since the output is one-hot encoding. The perturbation(ln.15) is calculated by the formula below:

$$perturbation = \epsilon * (ft\_grad - ff\_grad)$$

Faulty gradient is denoted as ft_grad and fault-free gradient is denoted ad ff_grad. And epsilon decides the effect of perturbation.

D. Compare method

We choose two method to compare with proposed method. The first method, noted as baseline, is training a fault-free model with the original dataset. The second method is training a faulty model with original dataset.

Algorithm 1 $gen\_pattern$

for $batch = 1, 2, \ldots$ **do**
  for $time\_window = 1, 2, \ldots, N$ **do**
    $different\_time = 0$
    for $i = 1, 2, \ldots, check\_time$ **do**
      $fault\_free = good\_Inference$
      $faulty = bad\_Inference$
      if $fault\_free \neq faulty$ **then**
        $different\_time+ = 1$
      end if
    end for
    if $different\_time > K$ **then**
      add new pattern
    else
      $perturbation$ = sum of two output difference
      add $perturbation$ to pattern
    end if
  end for
  if $num\_of\_new\_pattern > M$ **then**
    break
  end if
end for

4. EXPERIMENTAL RESULTS

A. Setup

We use the package in [5] to build the SNN model and train faulty and fault-free model. It adopts rate encoding for spikes. For all models, we use the same topology and adopt Adam optimizer and the same hyperparameters. As for adversarial attack generated pattern, we modify the pattern generating method of [3]. Because SNN models are stochastic, we need to inference more times to assure the faulty model output value and fault-free model output value. If the value of these two models are different more than a specific time out of inference time, this patterns is added to our new data set. For each epoch, we add a hundred patterns into the data set. Thus, there will be total nineteen hundred new patterns for twenty epochs.

Topology	Leaning rate	Epoch	Batch size	Test repetition
784×128×10	0.001	20	64	20

Table 1: Hyperparameters for training

Time window	Inference time	Different time	Pattern number
20	5	4	100

Table 2: Hyperparameters for Generating Patterns

B. Accuracy Results

In this section, we compare three methods under different situation under the same training setup. Baseline is the model trained without fault insertion. Previous one is the model trained with fault insertion, which is proposed in [5]. As for fault models, we choose two neuron fault models, HSF and NASF. For each fault model, we train two fault-tolerant models with different fault injection ratio.

In the fault-free scenario, the accuracy of the proposed models trained with HSF and NASF are close to baseline. However, the accuracy is slightly lower when fault injection ratio is larger, which means that the larger fault injection ratio could lead to lower accuracy in fault-free scenario. The reason is that while training, we inject the fault to trade accuracy in fault-free scenario for better accuracy in faulty scenario. The larger injection ratio would cause the model to have a tendency to faulty scenario.

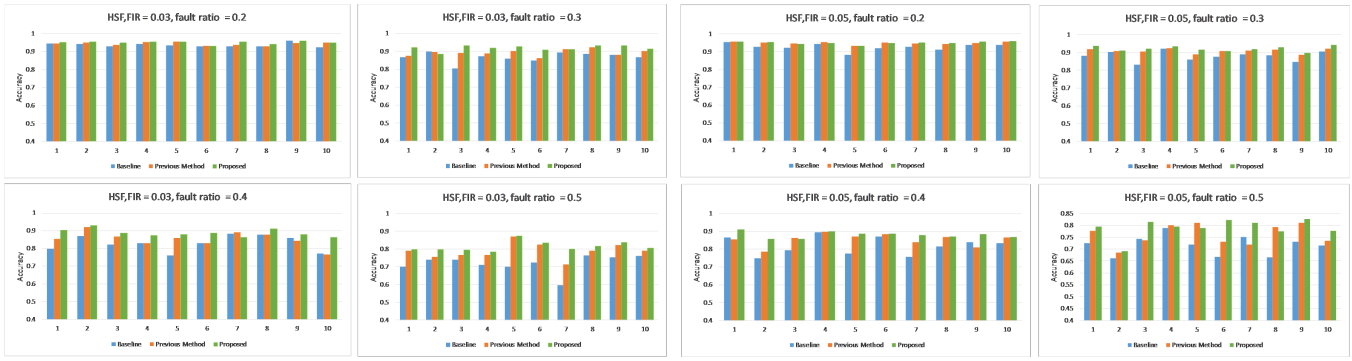
In the faulty scenario, we compare the three methods under different fault model and different fault ratio. Fig. 3 shows the accuracy for each case. For each graph, the vertical represents accuracy, and the horizontal axis represents the ten instances with same fault ratio but different fault locations. According to Fig. 3 (a), the accuracy difference between the proposed technique and previous work is quite small. As the fault ratio gets larger, the difference would grows as well. Fig. 3 (b) shows the accuracy for model trained with NASF. Compared to HSF, the accuracy differences between models of NASF is much larger. With larger fault ratio, our proposed method can reach much better accuracy than previous work, especially for cases with larger fault ratio.

Table 4 shows the mean accuracy for each case. The models trained with our proposed technique outperform the others for the four cases with different fault ratio. Please notice that there is more significant difference between our method and the other when the fault ratio gets larger.

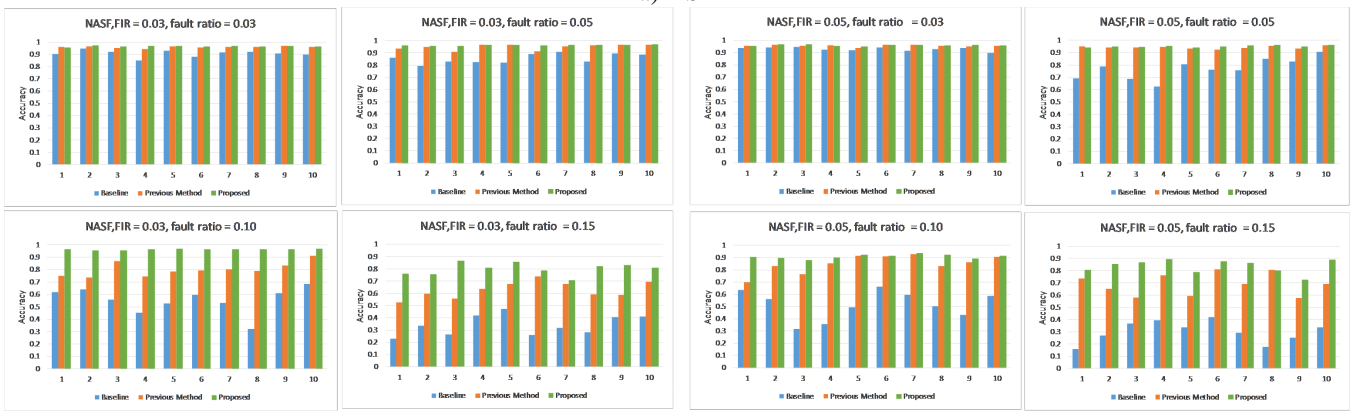
C. Convergence Speed Results

In this section, we shows the time information to see the convergence speed among different methods.While training our proposed method, we store the generated patterns. Then we implement the method in [5] while the data set contains the original data set and the stored patterns.

In Table 5 to Table 8, we shows the cpu run time for the previous method and our proposed method. The proposed method with pattern generation means that it generates new patterns while training. The proposed method without pattern generation means that we train the model with the data set which contains the original data set and the stored patterns. The epoch means that how much epoch do we need for the model without pattern generation to achieve the same accuracy as previous method trained with twenty epochs. We record the mean accuracy for each model in the presence of ten different fault sets with the same fault ratio.



a) HSF



b) NASF

Fig. 3: Accuracy Comparison among Baseline, Previous Work, and Proposed Method

FIR	Basline	Previous w/ HSF	Previous w/NASF	Proposed w/HSF	Proposed w/NASF
0.3	0.971	0.974	0.973	0.974	0.975
0.5		0.973	0.964	0.967	0.968

Table 3: Average Accuracy of Each Model under Fault-free Scenario

HSF					
Fault ratio	Baseline	Previous w/ FIR=0.03	Previous w/ FIR=0.05	Proposed w/ FIR=0.03	Proposed w/ FIR=0.05
0.2	0.9271	0.9435	0.9491	0.9498	0.9510
0.3	0.8800	0.8934	0.9088	0.9181	0.9214
0.4	0.8195	0.8530	0.8542	0.8880	0.8804
0.5	0.7168	0.7889	0.7602	0.8139	0.7897
NASF					
Fault ratio	Baseline	Previous w/ FIR=0.03	Previous w/ FIR=0.05	Proposed w/ FIR=0.03	Proposed w/ FIR=0.05
0.03	0.9079	0.9592	0.9570	0.9646	0.9593
0.05	0.8532	0.9472	0.9427	0.9616	0.9509
0.10	0.5543	0.8016	0.8499	0.8735	0.9079
0.15	0.3403	0.6280	0.6902	0.7987	0.8347

Table 4: Average Accuracy of Each Model under Faulty Scenario

NASF, FIR = 0.03					
	CPU Run Time				
Fault Ratio	Previous	Proposed w/ Ptn Generation	Proposed w/o Ptn Generation	Epoch	Time Reduction Ratio (%)
0.03	1948.02	2280.24	1715.21	16	11.95
0.05	2187.94	2410.26	1699.07	16	22.34
0.10	2127.99	2498.39	585.46	5	72.49
0.15	1765.30	2222.85	191.00	1	89.18

Table 5: CPU time for NASF, FIR = 0.03

NASF, FIR = 0.05					
	CPU Run Time				
Fault Ratio	Previous	Proposed w/ Ptn Generation	Proposed w/o Ptn Generation	Epoch	Time Reduction Ratio (%)
0.03	1708.46	1980.48	1908.48	20	-11.71
0.05	1742.72	2400.38	1948.01	20	-11.78
0.10	1617.79	1919.36	340.86	3	78.93
0.15	1857.12	2166.75	195.66	1	89.46

Table 6: CPU time for NASF, FIR = 0.05

HSF, FIR = 0.03					
	CPU Run Time				
Fault Ratio	Previous	Proposed w/ Ptn Generation	Proposed w/o Ptn Generation	Epoch	Time Reduction Ratio (%)
0.2	1695.65	2930.16	1880.42	20	-10.90
0.3	1766.49	2948.36	1855.86	20	-5.06
0.4	1792.68	2880.10	1862.08	20	-3.87
0.5	1821.97	2876.92	1927.42	20	-5.79

Table 7: CPU time for HSF, FIR = 0.03

HSF, FIR = 0.03					
	CPU Run Time				
Fault Ratio	Previous	Proposed w/ Ptn Generation	Proposed w/o Ptn Generation	Epoch	Time Reduction Ratio (%)
0.2	1856.91	2552.15	1341.46	13	27.76
0.3	1839.59	2557.41	868.36	8	52.80
0.4	1829.27	2590.57	774.41	7	57.67
0.5	1678.04	2350.01	1298.80	14	22.60

Table 8: CPU time for HSF, FIR = 0.05

From the results, we can clearly see that this method can significantly reduce the run time for most cases. However, for HSF, when FIR is 0.03, the convergence speed cannot be improved for these four fault ratio. This situation also happens to NASF when FIR is 0.05 and fault ratio is low. The reason can be observed in Fig. 3. If the accuracy of models with our proposed method are close to the previous works, it could hardly achieve the same accuracy under smaller epochs. In other words, the larger the difference in accuracy is, the less epoch we need to achieve the same performance.

5. DISCUSSION

There are some questions we observe while implementing our method:

1) The number of generated patterns for each epoch is one hundred in our experimental results, which means that there are total 1900 patterns generated after twenty epochs. However, the original training set contains 48000 patterns. We are wondering that how many patterns are needed to generated for training set containing 48000 patterns so that the accuracy and convergence speed could be improved further.

2) According to the experimental results, the run time can be

reduce for most cases. However, the other cases can only achieve lower accuracy compared to the previous work. Is it possible to decide which fault model under certain fault ratio isn't suitable for this training method in advance?

6. CONCLUSION

In this work, we proposed a improved fault-tolerant training technique combined with adversarial attack pattern generation. We consider two neuron fault models, HSF and NASF, with different fault injection ratio and fault ratio. The experimental results indicate that our proposed technique could reduce the accuracy drop more effectively in the presence of faults and maintain the acceptable accuracy in fault-free scenario. Besides, it also achieves better convergence speed if we store the generated pattern in advance. Both results shows the effectiveness of adversarial attack generated patterns for fault-tolerant training.

There are some possible extensions or improvements from this work. Since we focus on two neuron fault, there are still other fault models that should be considered. Choosing a suitable FIR before training models for different fault models could be important. Besides, we only consider one fault model at a time. It is more desirable for a general model to consider multiple fault models. In conclusion, this work improved the overall performance of fault-tolerant models. We look forward that there will be more studies focusing on this promising field.

7. CONTRIBUTION

Our team follows a collaborative work mode where we periodically discuss implementation directions and adapt to find the better solutions. Thus, the work is done with two individuals making an even contribution.

8. REFERENCES

[1]A. Gebregiorgis and M. B. Tahoori, "Testing of Neuromorphic Circuits: Structural vs Functional," 2019 IEEE International Test Conference (ITC), Washington, DC, USA, 2019, pp. 1-10, doi: 10.1109/ITC44170.2019.9000110.

[2]Y. -Z. Hsieh, H. -Y. Tseng, I. -W. Chiu and J. C. M. Li, "Fault Modeling and Testing of Spiking Neural Network Chips," 2021 IEEE International Test Conference in Asia (ITC-Asia), Shanghai, China, 2021, pp. 1-6, doi: 10.1109/ITC-Asia53059.2021.9808431.

[3]H. -Y. Tseng, I. -W. Chiu, M. -T. Wu and J. C. -M. Li, "Machine Learning-Based Test Pattern Generation for Neuromorphic Chips," 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD), Munich, Germany, 2021, pp. 1-7, doi: 10.1109/ICCAD51958.2021.9643459.

[4]I. -W. Chiu, X. -P. Chen, J. S. -I. Hu and J. Chien-Mo Li, "Automatic Test Configuration and Pattern Generation (ATCPG) for Neuromorphic Chips," 2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD), San Diego, CA, USA, 2022, pp. 1-7.

[5]H. -Y. Huang, M. -B. Fan, "Fault-Tolerant Training for Neuromorphic Circuits"

[6]Roy, K., Jaiswal, A., Panda, P., "Towards spike-based machine intelligence with neuromorphic computing." Nature 575, 607–617 (2019).