Accuracy Tolerant Neural Networks Under Aggressive Power Optimization

Xiang-Xiu Wu*, Yi-Wen Hung*, Yung-Chih Chen†, Shih-Chieh Chang*‡

*Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan

†Department of Computer Science & Engineering, Yuan Ze University, Taoyuan, Taiwan

‡Electronic and Optoelectronic System Research Laboratories, ITRI, Hsinchu, Taiwan

jaubau999@gapp.nthu.edu.tw, ywhung@gapp.nthu.edu.tw, ycchen.cse@saturn.yzu.edu.tw, scchang@cs.nthu.edu.tw

***Abstract*—With the success of deep learning, many neural network models have been proposed and applied to various applications. In several applications, the devices used to implement the complicated models have limited power resources, and thus aggressive optimization techniques are often applied for saving power. However, some optimization techniques, such as voltage scaling and multiple threshold voltages, may increase the probability of error occurrence due to slow signal propagation, which increases the path delay in a circuit and fails some input patterns. Although neural network models are considered to have some error tolerance, the prediction accuracy could be significantly affected, when there are a large number of errors. Thus, in this paper, we propose a scheme to mitigate the errors caused by slow signal propagation. Since the delay of multipliers dominates the critical path, we consider the patterns significantly altered by the slow signal propagation in a multiplier. We propose two methods, weight distribution and error-aware quantization to prevent the patterns from failure. Since we modify a neural network on the software side and it is unnecessary to re-design the hardware structure. The experimental results show that the proposed scheme is effective for several neural network models. It can improve the network accuracy by up to 27% under the consideration of slow signal propagation.**

***Index Terms*—Deep neural networks, timing violation, error mitigation, model optimization**

I. Introduction

The breakthrough of deep learning with deep neural networks (DNNs) brings about a trend in machine learning in recent years. Various types of neural networks have been proposed with the success of deep learning, like convolutional neural networks (CNNs) [1]–[3] and recurrent neural networks [4], [5] (RNNs). They are widely applied to different application domains, such as image classification [1]–[3], [6], natural language processing [4], [7], [8], object detection [9], [10], etc. However, most of these models are power-consuming due to a large number of multiply-accumulate (MAC) operations and memory access. It is challenging to deploy the models on an edge device which usually has limited power resources.

Thus, in the past years, many efforts are devoted to solving the issue [11]–[14]. For example, on the software side, several works propose to compress the neural networks for reducing the computational complexity and/or memory access without significant accuracy degradation, such as quantization [14] and pruning [11], [13]. In general, the former quantizes network parameters and the latter prunes redundant or non-sensitive neurons/filters to reduce the network size. These techniques are crucial for deploying neural networks on resource-limited devices.

On the hardware side, many optimization techniques are proposed to pursuit a good trade-off between performance and power consumption. However, some techniques for saving power, like supply voltage scaling and multiple threshold voltages, may slower the signal propagation and lead to timing violations, so that wrong signal values could be fetched by flip-flops, resulting in unexpected computation errors.

Although neural networks are considered to be able to tolerate faults/errors, the abilities are actually limited. There have been several works that proposed some methods to improve the fault/error tolerance of a neural network. Basically, they can be classified into three categories [15]: (1) explicitly augmenting redundancy [16], [17], (2) modifying training strategy [18], [19], and (3) optimization under constraints [20]–[22]. The primary error types these methods tend to deal with are stuck-at faults and random bit flips, which usually occur in memory or buffer cells. Additionally, some other works, like the Razor flip-flop [23], solve the errors due to timing violations by modifying the hardware structure.

In this paper, we tackle the timing violation errors due to slow signal propagation in the hardware device implementing a pre-trained neural network. We propose methods to mitigate the errors for recovering/improving the network accuracy without re-designing the hardware structure.

In general, in a CNN model, the dominant computation is the multiply-accumulate operation. When the signals become slower due to aggressive power optimization (e.g., voltage scaling), timing violation errors would more likely occur in a multiplier, since it usually has longer delay paths. Furthermore, whether an error occurs depends not only on the current state of a circuit but also on the previous state. Thus, we conclude that, during the computation of a CNN, the occurrence of an error highly depends on the current input and the previous input, and is almost in the multiplication operation.

As a result, we first identify the input patterns which more likely lead to timing violation errors in a multiplier, and build an error table based on simulation under the consideration of slow signal propagation. With the error table, we then propose two methods, weight distribution and error-aware quantization,

978-3-9819263-4-7/DATE20/©2020 EDAA

to adjust the weights and the architecture of a neural network to mitigate the timing violation errors. As a result, the network accuracy can be improved.

In the experiments, we consider two neural network models, LeNet-5 [6] and ResNet-20 [3], and two datasets, MNIST [24] and CIFAR-10 [25]. However, for the ResNet-20 model, we only train it with the CIFAR-10 dataset, since the MNIST dataset is too easy to obtain high accuracy. Thus, there are a total of three trained models. The experimental results show that the proposed methods can improve the accuracy of these models, when we consider the timing violation errors due to slow signal propagation. The improvement is up to 27%.

The main contributions of this work are as follows:

- **We propose a scheme to mitigate the errors due to slow signal propagation in the hardware device that implements a neural network without re-designing the hardware structure.**
- **We propose a weight distribution method and an error-aware quantization method to reduce the computation errors of multiplication in a neural network.**
- **We conduct experiments to demonstrate that the proposed scheme effectively improves the accuracy of three neural network models under the consideration of slow signal propagation.**

The remainder of this paper is organized as follows: Section II reviews some background and related works. Section III presents the proposed methods. Section IV shows the experimental results. Finally, the conclusion is presented in Section V.

II. Background & Related works

A. CNNs

A CNN is a multi-stage model. In general, a stage is composed of a convolutional layer, an activation layer and a pooling layer. The convolutional layer applies convolution operations to the input feature maps. The following activation and pooling layers rectify the feature maps and reduce their scale, and pass the outputs to the next stage. Additionally, one or multiple fully connected layers are applied to the last few layers of a CNN.

Convolutional layers are the core of a CNN. A convolutional layer is made up of a set of learnable filters and a filter is composed of weights and biases. The function of a filter at the lth layer can be mathematically formulated as (1), where, K, W, and H are the depth, width, and height of the filter, respectively. $z_{p,q,k}^{l-1}$ is an input from the $l-1^{th}$ layer and $w_{p,q,k}$ is the corresponding weight in the filter. Additionally, b is the bias. There are a total of $H \times W \times K + 1$ MAC operations for the filter to generate an output value.

$$u = \sum_{k=0}^{K-1} \sum_{q=0}^{W-1} \sum_{p=0}^{H-1} z_{p,q,k}^{l-1} w_{p,q,k} + b \quad (1)$$

Basically, the MAC operations dominate the computation efforts of a CNN. Additionally, since multiplication is the most complicated operation, timing violation errors will more likely to occur in a multiplier due to slow signal propagation. Thus, in this paper, we consider the errors in the multiplier and propose methods to mitigate the errors.

B. Error Tolerance in Neural Networks

Many techniques have been proposed to enhance the error tolerance of a neural network. The first category of methods [16], [17] creates redundancies in a neural network to recover or alleviate errors. The second category of methods [18], [19] proposes to enhance the training scheme. It trains a neural network with the injected errors and adds regularization terms to make the neural network more tolerant to errors. The last category of methods [20], [21] increases the error tolerance of a neural network by solving the optimization problem, such as finding the weights which can minimize the influence of errors. Recently, there are also some works [22] which apply the similar concept to advanced models for making them more robust to bit-flip errors. Moreover, the primary fault models of these methods are the stuck-at fault model and the random bit flip model. The errors they tend to solve are orthogonal to the errors that we would like to deal with in this paper.

C. Error-detection & Error-correction Technique

Reference [23] proposed an approach, called Razor, to mitigate the timing violation errors in dynamic voltage scaling. The key idea is to deploy additional Razor flip-flops with a delayed clock in a circuit to detect whether a timing error occurs or not. When an error is detected by Razor, it can correct it with penalties. Recently, Razor (or similar method) has been applied to neural network accelerators to increase their timing error resilience [26], [27]. Although the method is effective, it needs to re-design the hardware to deploy additional flip-flops and a delayed clock.

The proposed scheme in this paper could work together with Razor, because we do not need to re-design the hardware. We can reduce the number of timing violation errors, so that fewer penalties are required by Razor for detecting and correcting errors.

III. Proposed scheme

In this section, we first present how we analyze the timing violation errors in a multiplier and construct an error table. Next, we present the proposed two methods for mitigating the errors by adjusting weights in a neural network. Then, we discuss how we compute the sensitivity of a weight to perturbation. We determine which proposed method should be applied to a weight according to its sensitivity. Finally, we present the overall flow of the proposed scheme.

A. Error Analysis and Error Table Construction

Slow signal propagation is the main reason that causes timing violation errors. To analyze the errors in a multiplier, we simulate a multiplier with a shortened clock period, so that the critical paths would violate timing constraints. Furthermore, since the occurrence of an error depends not only on the current input, but also on the previous input, we apply all

TABLE I
EXAMPLE OF ERROR TABLE WITH PARTIAL INPUT COMBINATIONS.

Previous Input		Current Input		Output
A_{t-1}	B_{t-1}	A_t	B_t	O
00	00	10	11	0010
00	01	10	11	0010
00	10	10	11	0010
00	11	10	11	0010
01	00	10	11	0010
01	01	10	11	0010
01	10	10	11	0010
01	11	10	11	**1010**
10	00	10	11	0010
10	01	10	11	0010
10	10	10	11	0010
10	11	10	11	0010
11	00	10	11	0010
11	01	10	11	0010
11	10	10	11	0010
11	11	10	11	0010

TABLE II
EXAMPLE OF COMPRESSED ERROR TABLE.

Current Input		Probability	Output
A_t	B_t	P	O
⋮			
10	00	1	0000
10	01	1	1110
10	10	1	0100
10	11	15/16	0010
10	11	1/16	**1010**
⋮			

the possible current and previous input combinations in the simulation, and collect the outputs to build an error table. The error table reveals which input combinations more likely result in erroneous outputs and the error magnitude.

Table I shows an error table example of a 2-bit 2's complement multiplier. Only partial input combinations are presented for simplification. A and B denote the inputs of the multiplier. In a neural network, A and B can be seen as a weight and an input to a neuron/filter, respectively. For the current input vector $(A_t, B_t) = (10, 11)$, only the previous input vector $(A_{t-1}, B_{t-1}) = (01, 11)$ results in the wrong output 1010, as highlighted in red. This confirms that the occurrence of an error is related to the previous input vector as well. Actually, the error magnitude is also related to the previous input vector.

However, it is hard or expensive to obtain the previous input vector during the network inference. Thus, without loss of generality, we assume that the distribution of previous input vectors is uniform and compress the error table. As shown in Table II, the output only depends on the current input with a probability. For example, the current input vector $(A_t, B_t) = (10, 11)$ has a probability of $1/16$ to result in the wrong output 1010 and a probability of $15/16$ to result in 0010.

B. Timing Violation Error Mitigation

According to the error table, we have two observations: (1) Some input vectors have higher probabilities of generating wrong outputs. (2) The error magnitude is often large, since the most significant bit is triggered by the critical path. Thus, the influence of timing violation errors would be severe. To mitigate the errors, we propose two methods to adjust the weights based on the error table: (1) Weight distribution and (2) Error-aware quantization. Additionally, before we apply the two methods, we arrange the error table to show the probability that a weight has of resulting in a wrong output. It also reveals which weight is easier to produce a wrong output. Table III is an example arranged from Table II. Here, we assume that each weight has an even chance to be multiplied by each input number. For example, for $A_t = 10$ in Table II, there are 4 combinations of B_t and only $(A_t, B_t) = (10, 11)$ can produce a wrong output with a probability of $1/16$. Thus, the probability of producing a wrong output for the weight $A_t = 10$ is $1/64$ as shown in Table III. We call that the weight 10 has an *error probability* of $1/64$.

TABLE III
EXAMPLE OF ERROR TABLE WITH ERROR PROBABILITIES OF WEIGHTS.

Weight	Error Probability
A	P
00	0
01	0
10	1/64
11	2/64

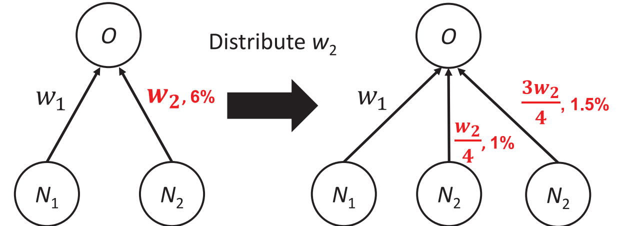
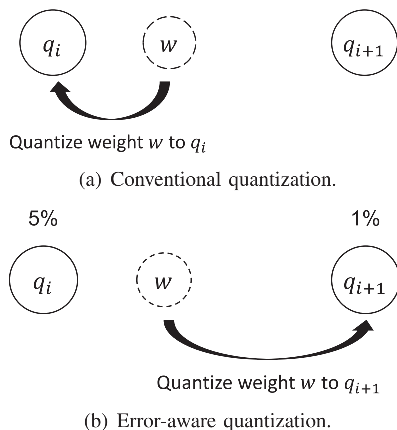


Fig. 1. Example of weight distribution. w_2 is distributed into $w_2/4$ and $3w_2/4$, which have lower error probabilities.

1) Weight Distribution: Weight distribution aims to mitigate errors while preserving the network accuracy. The method is similar to Net2WiderNet, which is a technique used in Net2Net [28] for transferring the knowledge of a previous network to a new deeper or wider network. We distribute one weight w into two weights, such that the sum of the probabilities of producing a wrong output for the two weights is lower than that for w.

Let us use the example in Fig. 1 to illustrate weight distribution. In Fig. 1, suppose that there are two inputs N_1 and N_2 and their corresponding weights are w_1 and w_2, respectively. The function of the output O is $N_1 * w_1 + N_2 * w_2$. Additionally, w_2 has a high error probability of 6%. To reduce the probability of producing erroneous O, we distribute w_2 into two weights, $w_2/4$ and $3w_2/4$, and their corresponding error probabilities are only 1% and 1.5%. Thus, the overall error probability is reduced from 6% to 2.5%. Furthermore, please note that the function of O does not change, which is still $N_1 * w_1 + N_2 * w_2$.

For a weight to distribute, there may exist several valid



(a) Conventional quantization.

(b) Error-aware quantization.

Fig. 2. Different strategies in conventional quantization and our error-aware quantization.

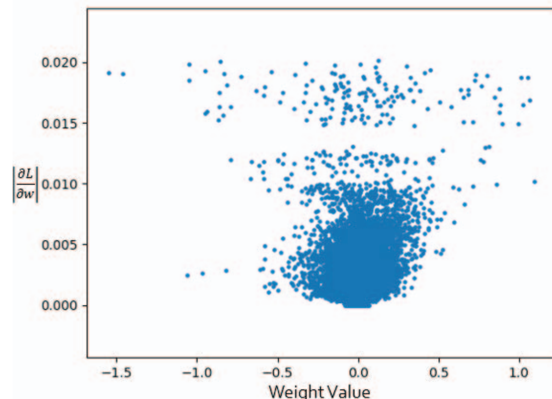


Fig. 3. Sensitivities of weights in LeNet-5.

combinations of two weights to choose. We examine each combination and select the one which has the lowest sum of error probabilities.

Although weight distribution can reduce the probability of error occurrence while preserving the network accuracy, it increases the number of MAC operations. Thus, the second method aims to mitigate errors without increasing computation efforts.

2) Error-Aware Quantization: Quantization [11], [14] has been applied to neural networks for saving storage space and memory access. The key approach is to represent a weight and/or an input with a lower bit-width. In general, since decreasing the bit width would affect the network accuracy due to precision reduction, the approach quantizes a value to its closest quantized value for minimizing the accuracy loss. However, in our method, we consider not only the distance between the weight and its quantized value, but also the error probability of the quantized value.

Basically, when we quantize a weight w, there are two choices of quantized values for w, q_i and q_{i+1}. For conventional quantization, w is quantized to the value which is closest to it, as shown in Fig. 2(a). However, for our error-aware quantization, we choose the quantized value which has a lower error probability, as shown in Fig. 2(b). The chosen value may not be the closest one.

Compared to weight distribution, error-aware quantization can mitigate errors without extra computation efforts, but may cause an accuracy loss when a weight is not quantized to the closest one. Thus, error-aware quantization should be applied to a weight which is less sensitive to perturbation, i.e., perturbing the weight value does not significantly alter the network accuracy. To this end, we present a method to evaluate the sensitivity of a weight to perturbation in the next subsection.

C. Sensitivity Of A Weight

We evaluate whether a weight w is sensitive to perturbation or not by computing its gradient $\frac{\partial L}{\partial w}$ and see the absolute value $\mid \frac{\partial L}{\partial w} \mid$ as the sensitivity of the weight. The reason is that a weight with a smaller $\mid \frac{\partial L}{\partial w} \mid$ indicates that perturbing the weight value has less impact on loss L.

For each weight in a neural network, we compute its sensitivity with the training dataset. Fig. 3 shows the sensitivities of the weights in LeNet-5. As you can see, most weights have a low sensitivity and the sensitivity of a weight is not necessarily directed to the weight value. Thus, we empirically apply weight distribution to the top 1% of sensitive weights based on the error table. A weight is distributed if the error probability can be reduced. Furthermore, we apply error-aware quantization to the remaining 99% of weights, which are less sensitive. Please note that the percentage of sensitive weights can be a user-defined parameter for trading off accuracy and computation effort.

D. Overall flow

Fig. 4 shows the overall flow of the proposed scheme. We first analyze and simulate the target multiplier design to construct an error table. Then, we compute the sensitivities of the weights in the given neural network. Finally, based on the sensitivities and the error table, we apply weight distribution to the top 1% of sensitive weights and error-aware quantization to the remaining 99% of weights, and obtain an improved neural network. In the proposed scheme, if the target multiplier design does not change, we reuse the same error table for different neural networks.

IV. Experimental Results

A. Experimental Setup

We implemented the proposed methods in Python with PyTorch [29] and the experiments were conducted on a workstation with two 2.1GHz Intel Xeon CPUs and two NVIDIA GeForce GTX 1080 Ti GPUs. The multiplier design under consideration is a signed booth encoded radix-4 INT8 multiplier generated by the Synopsys Design Compiler with the FreePDK 45nm cell library. Furthermore, it costs approximately 4 days to construct the error table of the multiplier, which is a one-time effort.

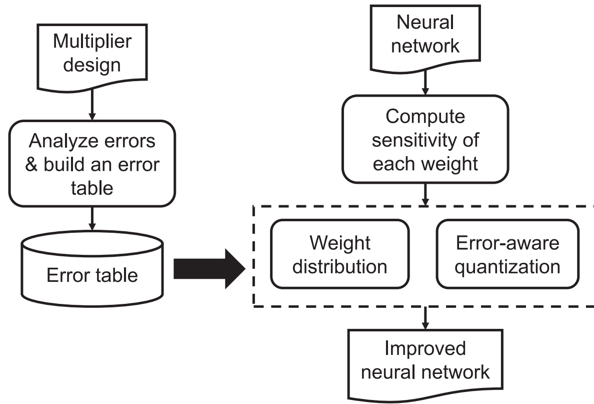


Fig. 4. Overall flow of the proposed scheme.

TABLE IV
NETWORK ACCURACY BEFORE AND AFTER INT8 QUANTIZATION.

Trained model	Accuracy	Accuracy after INT8 quantization
LeNet-5 + MNIST	98.75%	98.31%
LeNet-5 + CIFAR-10	73.21%	73.16%
ResNet-20 + CIFAR-10	92.74%	92.26%

We applied the proposed methods to three trained neural network models: (1) LeNet-5 [6] trained with the MNIST [24] dataset, (2) LeNet-5 trained with the CIFAR-10 [25] dataset, and (3) ResNet-20 [3] trained with the CIFAR-10 dataset. The numbers of labeled images in the two datasets are 70000 and 60000, respectively. The resolution of each image is 32 * 32 pixels and both two datasets have 10 classes. When we trained a model with the CIFAR-10 dataset, we applied some training schemes to improve the network accuracy, such as random cropping, image flipping and so on. For the MNIST dataset, we directly fed the original images to the model without any extra process. Additionally, the models were trained with the SGD optimizer. After training, we further quantized the trained models to the INT8 representation. Table IV shows the accuracy of the three trained models before and after INT8 quantization.

Furthermore, the statistical delay of a path in a circuit is traditionally assumed to be Gaussian [30]. 3σ is a common setting [31] in cell library design for ensuring that critical paths can meet the timing constraints. That is, if the delay of a path exceeds u+3σ (0.135%), it will be determined as timing violation. Here, u is the mean of the path delay. In general, aggressive power optimization would change and shift the delay distribution of a path [32]. To take the change into account, we choose two representative numbers in Statistics, 1.5σ and 2σ, to statistically consider the delay of a path. 1.5σ indicates the percentage of the delay distribution which exceeds u+1.5σ (6.68%), and it will be determined as timing violation.

In the experiments, when we execute the network inference with the timing violation errors, we query the compressed table as shown in Table II to inject errors based on the inputs of the multiplications. For example, for a multiplication $a * b$, we first query the table to obtain the probability of generating a wrong output for the current input $(A_t, B_t) = (a, b)$. Then, to consider the change of path delay distribution, we multiply the probability by 0.0668 (or 0.0227), which is the proportion of delay distribution that exceeds u+1.5σ (or u+2σ). Finally, we inject the error to the multiplication $a * b$ based on the regulated probability.

B. LeNet-5

LeNet-5 has 2 convolutional layers and 3 fully connected layers. We trained two LeNet-5 models separately with the MNIST and the CIFAR-10 datasets. Table V summarizes the experimental results of the two models. Column 1 shows the model names. Column 2 shows the applied values of error regulation. Columns 3 $\sim$ 6 show the results on network accuracy. They are the accuracy of the model after INT8 quantization, denoted as $N_{baseline}$, the accuracy of the model under the consideration of timing violation errors, denoted as N_{TError}, the accuracy of the improved model by using the proposed weight distribution and error-aware quantization (i.e., the complete flow in Fig. 4), denoted as N_{WD+EAQ}, and the accuracy of the improved model by applying error-aware quantization only to all the weights, denoted as N_{EAQ}. The last two columns show the numbers of MAC operations in the models. Since the method of weight distribution leads to extra computation efforts, the required number of MAC operations increases as shown in the last column.

The experimental results show that when we consider the timing violation errors due to slow signal propagation, the network accuracy drops significantly. However, the proposed methods can improve the accuracy with small overheads of MAC operations. The improved accuracy is very near to the baseline. Additionally, when we apply only the method of error-aware quantization, although the quality of improving accuracy slightly decreases, no extra MAC operation is required.

C. ResNet-20

ResNet-20 is more complicated than LeNet-5. It contains 19 convolutional layers and 1 fully connected layer with some shortcut paths. Table VI summarizes the experimental results of the ResNet-20 model trained with the CIFAR-10 dataset.

Basically, Table VI shows similar results to Table V. The proposed methods can effectively improve the network accuracy with small overheads of MAC operations. The largest improvement is approximately 27% ($89.59\% - 62.11\%$). Additionally, applying only the method of error-aware quantization is able to improve the accuracy and no extra MAC operation is required.

In summary, the proposed methods are effective to improve the accuracy of the models under aggressive power optimization which results in slow signal propagation.

V. CONCLUSION

In this paper, we propose a scheme to mitigate the timing violation errors due to slow signal propagation in the hardware

TABLE V
EXPERIMENTAL RESULTS ON LENET-5.

Model	Error rate regulation	Accuracy				#MACs	
		$N_{baseline}$	N_{TError}	N_{WD+EAQ}	N_{EAQ}	$N_{baseline}$ / N_{TError} / N_{EAQ}	N_{WD+EAQ}
LeNet-5 + MNIST	1.5σ	98.31%	94.49%	98.03%	97.86%	0.41M	0.42M
	2σ		97.02%	98.22%	98.02%		
LeNet-5 + CIFAR-10	1.5σ	73.16%	68.02%	72.23%	71.95%	0.65M	0.74M
	2σ		70.56%	72.73%	72.31%		

TABLE VI
EXPERIMENTAL RESULTS ON RESNET-20.

Model	Error rate regulation	Accuracy				#MACs	
		$N_{baseline}$	N_{TError}	N_{WD+EAQ}	N_{EAQ}	$N_{baseline}$ / N_{TError} / N_{EAQ}	N_{WD+EAQ}
ResNet-20 + CIFAR-10	1.5σ	92.26%	**62.11%**	**89.59%**	87.26%	40.55M	44.18M
	2σ		87.01%	91.59%	91.09%		

device that implements a neural network without re-designing the hardware structure. Two key methods of weight adjustment are proposed to reduce the probability of error occurrence. The first method, weight distribution, distributes a weight into two weights that have lower error probabilities. It can preserve the function of a neuron/filter, but will lead to extra computation efforts. Thus, we apply it only to the sensitive weights. Furthermore, the second method, error-aware quantization, quantizes a weight to the value that has a lower error probability. It is applied to all the remaining weights. The experimental results show that the proposed scheme is effective to improve the network accuracy for three neural network models under the consideration of slow signal propagation.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. Hinton, ”ImageNet classification with deep convolutional neural networks,” in *Annual Conference on Neural Information Processing Systems*, 2012, pp. 1097-1105.

[2] K. Simonyan and A. Zisserman, ”Very deep convolutional networks for large-scale image recognition,” in *International Conference on Learning Representations*, 2015.

[3] K. He, X. Zhang, S. Ren and J. Sun, ”Deep residual learning for image recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[4] T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and S. Khudanpur, ”Recurrent neural network based language model,” in *International Speech Communication Association*, 2010.

[5] S. Hochreiter and J. Schmidhuber, ”Long short-term memory,” *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997.

[6] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, ”Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, November 1998.

[7] K. Cho et al., ”Learning phrase representations using RNN encoder-decoder for statistical machine translation,” in *Empirical Methods in Natural Language Processing*, 2014.

[8] D. Bahdanau, K. Cho, and Y. Bengio, ”Neural machine translation by jointly learning to align and translate,” in *Annual Conference on Neural Information Processing Systems*, 2015.

[9] S. Ren, K. He, R. Girshick, and J. Sun, ”Faster R-CNN: Towards real-time object detection with region proposal networks,” in *Annual Conference on Neural Information Processing Systems*, 2015, pp. 91-99.

[10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, ”You only look once: Unified, real-time object detection,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[11] S. Han, H. Mao, and W. J. Dally, ”Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” in *International Conference on Learning Representations*, 2016.

[12] Adriana Romero et al., ”FitNets: Hints for thin deep nets,” in *International Conference on Learning Representations*, 2015.

[13] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, ”Pruning filters for efficient convnets,” in *International Conference on Learning Representations*, 2017.

[14] Y. Guo, A survey on methods and theories of quantized neural networks, CoRR, vol. abs/1808.04752, 2018.

[15] C. Torres-Huitzil and B. Girau, ”Fault and error tolerance in neural networks: A review,” *IEEE Access*, vol. 5, pp. 17322-17341, 2017.

[16] L. C. Chu and B. W. Wah, ”Fault tolerant neural networks with hybrid redundancy,” in *International Joint Conference on Neural Networks*, 1990.

[17] S. Piche, ”Robustness of feedforward neural networks,” in *International Joint Conference on Neural Networks*, 1992.

[18] C. H. Sequin and R. D. Clay, ”Fault tolerance in artificial neural networks,” in *International Joint Conference on Neural Networks*, 1990.

[19] P. J. Edwards and A. F. Murray, ”Penalty terms for fault tolerance,” in *International Conference on Neural Networks*, 1997.

[20] D. Deodhare, M. Vidyasagar and S. Sathiya Keethi, ”Synthesis of fault-tolerant feedforward neural networks using minimax optimization,” *IEEE Transactions on Neural Networks*, vol. 9, no. 5, pp. 891-900, September 1998.

[21] C. Neti, M. H. Schneider and E. D. Young, ”Maximally fault tolerant neural networks,” *IEEE Transactions on Neural Networks*, vol. 3, no. 1, pp. 14-23, January 1992.

[22] C. Schorn, A. Guntoro and G. Ascheid, ”An efficient bit-flip resilience optimization method for deep neural networks,” in *Design, Automation & Test in Europe Conference & Exhibition*, 2019.

[23] D. Ernst et al., ”Razor: A low-power pipeline based on circuit-level timing speculation,” in *Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[24] Y. LeCun and C. Cortes. The mnist database of handwritten digits. [Online]. Available:http://yann.lecun.com/exdb/mnist/

[25] A. Krizhevsky, V. Nair, and G. Hinton. The CIFAR-10 dataset. [Online]. Available:https://www.cs.toronto.edu/ kriz/cifar.html

[26] P. N. Whatmough et al, ”A 28nm SoC with a 1.2GHz 568nJ/prediction sparse deep-neural-network engine with ¿0.1 timing error rate tolerance for IoT applications,” in *IEEE International Solid-State Circuits Conference*, 2017.

[27] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, ”Thundervolt: Enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators,” in *Annual Design Automation Conference*, 2018.

[28] T. Chen, I. Goodfellow, and J. Shlens, ”Net2net: Accelerating learning via knowledge transfer,” in *International Conference on Learning Representations*, 2016.

[29] A. Paszke et al., ”Automatic differentiation in PyTorch,” in *Neural Information Processing Systems Workshop on Autodiff*, 2017.

[30] A. Srivastava, D. Sylvester, and D. Blaauw, Statistical Analysis and Optimization for VLSI: Timing and Power. New York: Springer, 2005.

[31] T. McConaghy, K. Breen, J. Dyck, and A. Gupta, Variation-Aware Design of Custom Integrated Circuits: A Hands-on Field Guide. New York: Springer, 2013.

[32] S. Ghosh and K. Roy, ”Parameter variation tolerance and error resiliency: New design paradigm for the nanoscale era,” *Proceedings of the IEEE*, vol. 98, no. 10, pp. 1718-1751, October 2010.