Vulnerability of Hardware Neural Networks to Dynamic Operation Point Variations

Jeng-Hau Lin[1], Xun Jiao[3], Mulong Luo[4], Zhuowen Tu[2,1], Rajesh K. Gupta[1]
[1]Dept. of Computer Science and Engineering, [2]Dept. of Cognitive Science, UC San Diego
[3]Dept. of Electrical and Computer Engineering, Villanova University
[4]School of Electrical and Computer Engineering, Cornell University
jel252@ucsd.edu, xjiao@villanova.edu, ml2558@cornell.edu, {ztu, rgupta}@ucsd.edu

***Abstract*—Deep Neural Networks have found many practical applications that are beginning to demand their implementation in hardware for a variety of cost, performance reasons. Microelectronic variability, that is, variations in manufactured hardware to variations in operating conditions are a practical reality that has led to design guardbands. These guardbands increasingly undermine the cost/performance advantage of hardware implementations and force us to seek more efficient solutions often spanning the entire system design.**

In this paper, we explore to what extent microelectronic variations affect the quality of results from neural networks and then understand with what opportunities we can improve networks to survive the challenges aroused by physical variations. Given the inherent resilience of neural networks due to the adaptation of their learning parameters, one would expect the quality of results produced by neural networks to be relatively insensitive to the rising timing error rates caused by increased variations. On the contrary, our results of error injection on multiple-layer perceptron (MLP) and convolutional neural network (CNN) show that the two canonical network types cannot withstand typical physical variations with voltage droops as small as 20 mV. Instead, alternative algorithmic implementations provide greater resilience. In particular, we examine here two such alternatives: the binarized neural network (BNN) and local binary pattern network (LBPNet) and present results that show greater sustenance against physical variations.

I. Introduction

Neural network algorithms have found use in a wide range of applications such as medical diagnostics, image classification, speech recognition, and natural language processing. This versatility has led to their implementation on a variety of hardware platforms: GPU, FPGA, and ASIC.

With the continuous scaling of CMOS technology, the underlying transistors in all these implementations are increasingly susceptible to variations in manufacturing and operating conditions. Dynamic variations in microelectronic systems, which are the main focus of this paper, are caused by environmental factors such as supply voltage droops and temperature fluctuations. Voltage droops are caused in response to instantaneous current fluctuations due to activities on the power delivery network. Temperature fluctuation could alter the circuit parameters such as carrier mobility and threshold voltage. Such variations can manifest themselves as timing errors, leading to incorrect computation outputs and system failures. Notwithstanding setting up guardbands is the standard solution to ensure the system's functionality, the incomprehension of NNs' vulnerability can derive over-designed guardbands encumbering the throughput of hardware accelerators or GPUs.

Due to the ability to adapt neural networks' learnable parameters for extracting the abstract common features in data, NNs have an inherent resilience to errors. Thus, one would expect that the quality of results produced by hardware neural networks (HNNs) to be relatively insensitive to the rising timing error rates caused by increased variation, thus opening doors for opportunistic reduction of guardbands to increase the operational efficiency of hardware. There is a need for a quantitative assessment here to explore the extent to which guardbands can be reduced in HNNs. We investigate this question as to whether and how much accuracy of HNNs could be affected by dynamic variations. To do this, we capture and represent variations from low-level hardware, and then expose them to neural network inferences. Unlike logic errors which can be derived through a mathematical formulation[2], variation-induced timing errors can only be obtained using gate-level simulation, making the error injection implementation time-consuming and not scalable.

Approach and Contributions: We propose a cross-layer approach to assess the vulnerability of HNNs to dynamic voltage and temperature variations, in which we extract the timing errors from the hardware layer using gate-level simulations and examine their effects on the software layer using error injections. To evaluate the soundness of this approach, we measure the timing errors using gate-level simulations (GLS) of post-layout circuits in TSMC 45nm technology. We vary the voltage and temperature in a wide range to examine the effects of variations. Then, we represent and inject these timing errors to neural networks during their inference. Finally, we examine the resilience of four types of neural networks, multi-layer perceptron (MLP), convolutional neural network (CNN), binarized convolutional neural network (BCNN) [3], and local binary pattern network (LBPNet) [5], [6], by testing them on MNIST dataset.

Based on our implementation and evaluation, this paper makes the following contributions:

- We extract the circuit level timing errors caused by voltage and temperature variations from twenty different operating conditions using gate-level simulations.

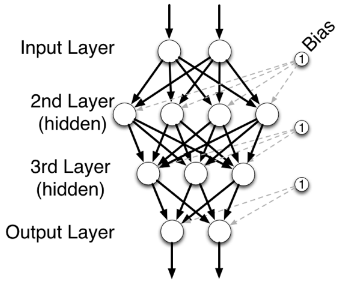

Fig. 1. An example of a 4-layer multi-layer perceptron neural network.

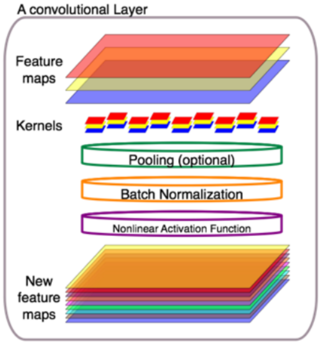

Fig. 2. The processes among a convolutional layer.

- We inject such timing errors back into neural network inference and evaluate the accuracy of MNIST dataset at different conditions.
- Our results quantitatively show that variations can significantly affect the inference accuracy on NNs.
- Among the four subject networks, LBPNet provides the most reliable error immunity that the other three networks cannot be on par.

II. Hardware Neural Networks

Modeled for neural processing, Figure 1 shows a typical neural network, an MLP consisting of an input layer, hidden layers, and an output layer. Except for the input layer, all remaining layers are composed of artificial neurons that represent the basic computation unit. An artificial neuron consists of a linear processing part followed by a nonlinear processing part. The linear part collects the output information, a.k.a activations, from the previous layer, and the collection method is a dot production between weights and activations. The nonlinear part includes regularization like dropout, and activation functions such as logistic sigmoid, hyperbolic tangent, or rectilinear unit (ReLU). The nonlinear activation function enables a neural network to be a universal function approximator. The forward-backward propagation algorithm intelligently applies the chain rule of calculus and gradient descent on neural networks to train the weights and hence minimizes the classification errors.

Since proposed in 1989, CNNs have pushed the performance of neural networks to a new realm. Figure 2 depicts the internal processes in a convolutional layer with nine kernels, each of which consists of three filters. The convolution operation models the hardwired bonding between the neurons on adjacent layers. It uses a sliding filter to perform dot-products of the filter and uses a portion of the input image gradually to generate an output image, namely the feature map. Since the convolution operations are differentiable, the filters can be trained to capture the features of the input images with backward propagation. *Pooling* is used to reduce the size of a feature map and increase the reception area by selecting the maximum pixel strength or averaging several pixel strengths. It benefits the translation invariance because it drops unnecessary minor information and preserves the most dominant features for the overall classification task.

The robustness of neural networks comes from many aspects. From a higher-level point of view, the training process of a neural network model is an ensemble of multiple linear or logistic regressions working in parallel. The regression ignores minor noises of the data and yields a model for the most likely distribution of the given data. Second, the regularization process inside a neural network also contributes to robustness because weights are deterministically penalized if the tensor norms grow too large and the connections can be dropped stochastically to elude a network learning unwanted noises. The weights are, thereby, trained to accommodate the majority of the data with the simplest probable distribution. Moreover, if a re-training process is involved, the convex optimization enforces the learnable parameters in a model to descend on the error surface *again*. Please note that we only assess the inference performance in this work without performing any re-training.

Binarized neural network (BNN) [3] was proposed as an extreme case of network quantization. During the training phase, it maintained two sets of weights: The set of weights contained floating-point numbers to guarantee a smooth gradient descent, and the other set was the binarized one obtained by a hard-hyperbolic tangent function that returned '$+1$' if the input was positive; otherwise, returns '-1'. The forward propagation used the binarized weights to predict network output and calculate loss, and the backward propagation relied on the floating weights to descend the model on a smooth error surface. Whenever the floating-numbered weights got updated, they were binarized and stored in the binarized weights. However, given that the binarized weights cannot carry sufficient information for most classification tasks, a small number of floating number calculations were introduced to compensate for the information loss, i.e., both bias addition and batch normalization were in floating numbers.

Local binary pattern network (LBPNet) [5], [6], as shown in Figure 3, was proposed as an alternative deep learning method to CNN for optical character recognition tasks. Instead of using multiplication-and-accumulation (MAC) operations, LBP operation [9] leveraged sampling, and comparison to efficiently capture features. Gupta *et al.* further introduced LBP to deep learning by stacking the LBP layers together, applying random projection to avoid channel accumulation, and deriving calculus chain rule to develop LBPNet's backward propagation. For OCR tasks, LBPNets delivered near state-of-the-art classification accuracy while reducing the computation demand and model size of convolutional layers by two to three

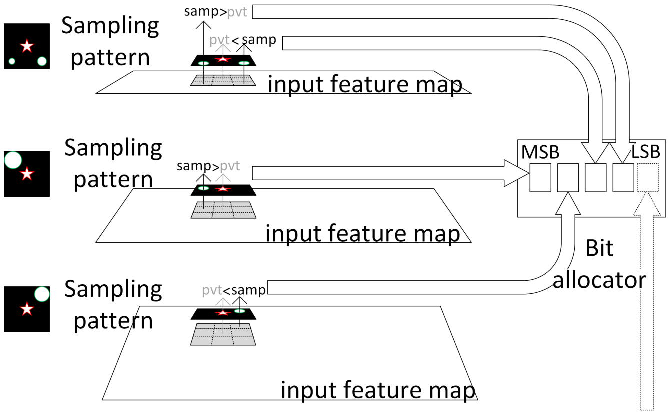

Fig. 3. A detailed illustration of an LBP layer. Three LBP patterns work like masks for sampling through the pivot aperture (pvt) and sampling apertures (samp). The comparison results are allocated to a bit array according to the random projection map.

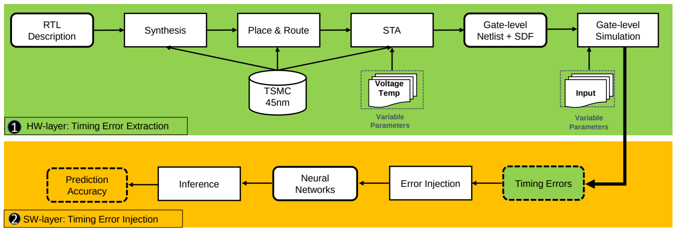

Fig. 4. Cross-layer assessment flow with two stages: a) HW-layer: Timing Error Extraction to extract the timing errors under different operating conditions; b) SW-layer: Timing Error injection into a neural network and perform inference.

orders of magnitudes. In this work, we also binarized the fully-connected layers of an LBPNet for the test of vulnerability.

Hardware variations could impact HNNs through timing errors in both computation logic and control logic. The errors in control logic could lead to catastrophic results, but, fortunately, most critical paths lie in computation logic, which is mainly composed of additions and multiplications, two of the most frequently used operations. Both the forward and backward propagation require intensive additions and multiplications, but most HNNs on ASIC, FPGA, and embedded GPUs do not support on-chip learning. Thus, we mainly focus on the timing errors that occur in addition and multiplication during the inference phase of HNNs.

III. Cross-Layer Vulnerability Assessment

The cross-layer vulnerability assessment is comprised of two phases, as shown in Figure 4: *Timing Error Extraction and Timing Error Injection*. a) The *Timing Error Extraction* phase implements the standard ASIC flow and uses gate-level simulation (GLS) to generate timing errors at each operating condition. b) In the *Timing Error Injection* phase, we inject the timing errors into neural networks and then perform inference. We vary the neural network genres and operating conditions to examine the resulted accuracy. More details about the two phases are illustrated as follows.

A. HW-layer: Timing Error Extraction

We extract the timing errors through *the Timing Error Extraction* module, as illustrated in Figure 4, which is divided into several steps. Note that we focus on dynamic variation-induced timing errors of computation units. We extract timing errors from the adder and the multiplier, which are the two most frequently used computation units in neural network computation. We use FloPoCo [1] to generate the synthesizable VHDL codes of floating-point units. We use *Synopsys Design Compiler* to synthesize the Verilog codes and use *Synopsys IC Compiler* to generate the post-place-and-route netlist in TSMC 45nm technology. Next, we use *Synopsys PrimeTime* to perform static timing analysis, generating Standard Delay Format (SDF) files at different operating conditions. To do this, we use the voltage-temperature scaling features of Synopsys PrimeTime for the composite current source approach of modeling cell behavior. We consider twenty operating conditions, as shown in Figure 8, which could introduce both mild and aggressive timing errors. Then, we use *Mentor Graphics ModelSim* to do SDF back-annotation gate-level simulations under nominal frequency to generate output data at different operating conditions. To extract timing errors, we compare the GLS output $y[t]$ with a pure-RTL simulation result $y\_gold[t]$, which is free from timing errors because there is no delay annotation. If there is a mismatch, then we define it as a timing error.

B. SW-layer: Timing Error Injection

We inject the timing errors extracted by the *Timing Error Extraction* phase to the neural networks by using the second phase *Timing Error Injection*. During the forward propagation in the neural network inference, we inject the errors into the arithmetic computations (addition and multiplication) in the convolutional layer (Conv layer), fully-connected layer (FC layer), average pooling layer (AvgPool layer), batch normalization layer (BatchNrom layer), and local binary pattern layer (LBP layer). There are several noteworthy facts must be highlighted regarding the error injection in the software layer: First, the XNOR operation and pop-count accumulation in BCNN and the comparison operation in an LBP Layer are not implemented in conventional arithmetic and logic units (ALUs) on CPUs or processing elements (PEs) on GPUs. We have to use multiplier and adders to carry out the 1-bit XNOR and the following accumulation in BCNN. For the comparison in an LBP Layer, we use the sign bit of subtraction to produce the comparison result instead. Therefore, the TER from adders and multipliers can affect the outputs of binarized Conv, binarized FC, and LBP layers.

On a circuit, different input could excite different paths, resulting in an input-specific timing error behavior. To mimic this, an exhaustive look-up table containing the entire input space for each bit position of each computation unit under all operating conditions needs to be implemented. Then, the computations need to look up the table to check whether it has a match on any input operands in the input space. This makes the inference process prohibitively slow. To approximate the situation, we inject the timing errors as [10]: let

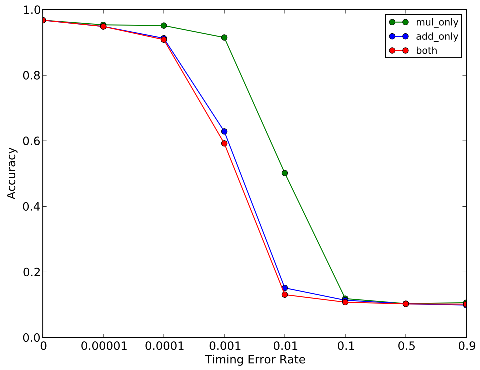

Fig. 5. MLP accuracy as a function of TER.

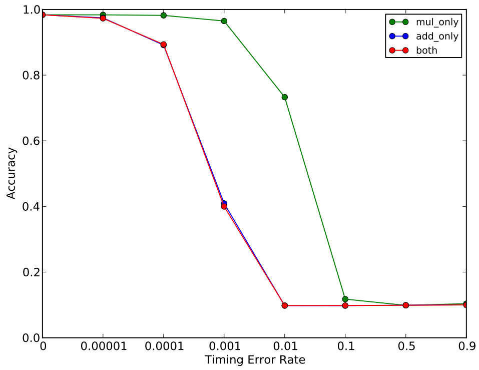

Fig. 6. CNN accuracy as a function of TER.

the computation units return a random value each time they have timing errors. We inject the error into the computation with the pair of adder TER and multiplier TER extracted from the *Timing Error Extraction* phase to mimic the time error behavior. For example, if the adder has a TER at 0.1, we inject errors to 10% of the total additions. This probability is determined by operating conditions and computation logic (addition or multiplication), which can represent the impact of timing errors on computation logic. We vary the error injection probability for each operating condition.

IV. Experiments

In this section, we measure timing errors under twenty operating conditions. Then, we measure HNNs accuracy as a function of varying timing error rates. Finally, we characterize the HNNs accuracy under dynamic variations using MLP, CNN, BCNN, and LBPNet.

A. Experimental Setups

In this work, we use tiny-dnn [8], a header-only, dependency-free deep learning library written in C++, as our deep learning platform for MLP and CNN. This platform is light weighted and is designed for deep learning on the limited computational resource, such as embedded systems and IoT devices. For CNN, we use LeNet-5 like architecture and replace LeNet-5's RBF layer with a fully-connected layer. For MLP, we use 3-layer MLP with a hidden layer of 60 neurons. We adopt the same structure of the BCNN for SVHN in the BNN paper and the LBPNet for MNIST in the LBPNet paper [5]. The synthesizable C codes for BCNN and LBPNet implemented by us for FPGA accelerators are used for the error injection. All the four sets of weights and kernels are pre-trained either from the referred tiny-dnn source or by us on an Nvidia Tesla K40 GPU.

We use MNIST and CIFAR-10 as our datasets to evaluate the neural network accuracy. MNIST (Mixed National Institute of Standards and Technology) of handwritten numbers is a well-known dataset for evaluating the performance of neural network classifiers. MNIST is into training set and test set with $60,000$ and $10,000$ 28×28 images. The features in MNIST are mainly strokes and outlines. Images in CIFAR-10 are daily objects of size 32×32, and the training and test sets include $50,000$ and $10,000$, respectively. CIFAR-10 is considered to be a more challenging datasets because the information and features reside in both outlines and textures. We choose MNIST to conduct a decent evaluation of vulnerability as the first step. Then, we deepen and widen BCNN and LBPNet by adding more layers, kernels, and random projection maps to conduct the second step of experiment on CIFAR-10 to understand the vulnerability of HNN on general object recognition.

For the hardware variations, we vary the voltage from 0.81V to 0.90V with a step at 0.01V and the temperature from $50°$C to $100°$C.

B. Accuracy under the Threat of Timing Errors

Before the error extraction, we assess the performance degeneration as a function of timing error rates. The accuracy is evaluated for both MLP and CNN under the TER at 0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 0.5, and 0.9 at three configurations as shown in Figure 5 and Figure 6; *add_only* means we only inject timing errors to adder, *mul_only* means we only inject timing errors to multiplier and *both* means we inject errors to adder and multiplier at the same time. We observe that for both MLP and CNN, as the TER increases, the accuracy drops monotonically. When the TER is 0.00001, the HNN can still deliver a decent accuracy close to original accuracy. Once the TER of adder reaches 0.0001, the accuracy drops to around 90% and continues dropping to 60% when the TER of adder reaches 0.001. In contrast, the multiplier exhibits a much less significant impact on HNN accuracy: the HNN can still deliver 90% accuracy even when the TER of multiplier reaches 0.001. In fact, for all examined TERs, the *mul_only* resulted accuracy is always higher than that of *add_only*. Moreover, the accuracy under *both* configuration is almost identical to that of *add_only* configuration, suggesting that adders-induced errors contribute to most of the accuracy drop.

One main reason is that the accumulated convolution sum or dot-product sum is fed into a nonlinear activation function and thereby directly affect the activation, while the errors from multipliers will be averaged and diluted. This suggests that more hardware design effort should be made on the adder to ensure its low TER. On the other hand, the worst accuracy of both NN genres is around 10%, when either *add_only* or

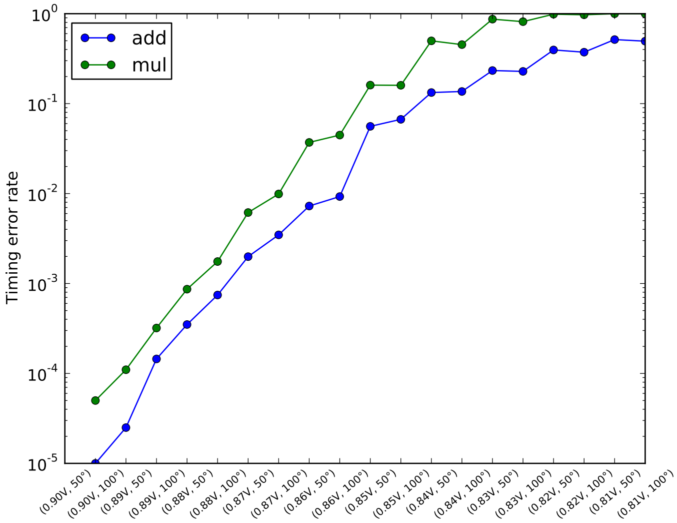

Fig. 7. TER of adder and multiplier under different operating conditions.

mul_only is 0.1. We can observe that such an accuracy drop starts saturating at 0.1 TER, almost identical to a random guess of the 10-class recognition task.

Another important observation is that the accuracy of CNN decreases more drastically than that of MLP, which conflicts our intuition of the higher capability of CNN. The classification accuracy at adder-only 0.001 TER is 61%, which is higher than CNN's 40% accuracy at 0.001 TER. However, when we inspect in detail, the fan-in of a neuron and a convolutional kernel explains the surprising observation. *The fan-in of a convolutional kernel is defined by the spatial size of a filter, which is* 3×3 *and relatively small compared to a neuron' fan-in in the MLP. Therefore, the injected error in MLP gets diluted better.*

In summary, such observations show that even though neural networks have inherent error resilience, the timing errors still can significantly affect neural network accuracy and motivate this work.

C. Vulnerability on MNIST

We then use the real dynamic operating conditions to obtain realistic timing error rates and thereby characterize the vulnerability of HNNs to dynamic variations. Notably, we use the *Timing Error Extraction* described in section III-A to characterize the timing error behavior of 32-bit floating-point adder and multiplier under different operating conditions, as shown in Figure 7. Besides the ideal condition without any error, the selected operating conditions cover a wide range of TERs: at the best condition (0.90V, 50°C) with TERs less than 0.0001; at the worst condition (0.81V, 50°C), 0.5 and 1.0 TER are found in adders and multipliers, respectively. By comparing these two computing units, we find that TER of the multiplier is always higher than the adder under the same condition. This is because the multiplier design has more critical paths than the adder, resulting in more timing violations. The TER of adder reaches 1% when the operating condition is around 0.86V. Based on Figure 5 and Figure 6, the accuracy drop starts to saturate when the TER of adder reaches 0.01; thus we expect to see the worst accuracy starting at around 0.86V.

We then present the accuracy of MLP, CNN, BCNN, and LBPNet under the twenty operating conditions, as shown in Figure 8, where we observe several important facts:

1) The lowest accuracy under worst-case operating conditions is around 10% for all the four networks across multiple conditions from (0.85V, 100°C) to (0.81V, 100°C). For MLP, CNN, and BCNN, this observation is expected as we can see from Figure 5 and Figure 6 where the accuracy drops to 10% when the TER of either unit reaches 0.01.
2) The four curves can be categorized into two groups because the MLP, CNN, and BCNN behaviors similarly, and the LBPNet's accuracy curve demonstrate a high immunity to the TER residing in adders and multipliers.
3) Fig. 8 shows that under the condition between (0.90V, 50°C) and (0.86V, 50°C), where the TER of adder is less than 0.01, the accuracy drop of MLP to its original accuracy is less than that of CNN, indicating MLP might be more resilient than CNN within a certain TER. Part of the reason for this is that given the same TER, the amount of errors in CNN is larger than MLP because CNN has more arithmetic operations, and the percentile of multiplications among all arithmetic computations are higher in CNN.
4) BCNN sustains slightly more timing errors than MLP and CNN. Compared with MLP's curve, BNN's vulnerability is enhanced twice since the classification drops to the same with MLP when the TER is doubled.
5) LBPNet keeps immune against the variation until we impose much harsher conditions. A 10% accuracy deterioration is observed at (0.86V, 50°C), while all the other three models significantly lose classification ability and fall around 10% accuracy. LBPNet totally fails to classify upon (0.85V, 100°C), at with the TERs climb to 0.1 and 0.5 for adders and multipliers, respectively.
6) Last but not least, we find the voltage and temperature both play an important role in determining the inference accuracy. By fixing the temperature at 100°C, reducing the voltage by 0.01V from 0.89V to 0.88V results an accuracy drop of the CNN model from 85.15% to 48.64%; by fixing the voltage as 0.88V, increasing the temperature by 50°C results in an accuracy drop from 70.34% to 48.64%. By comparing the accuracy at (0.90V, 50°C) and (0.86V, 50°C), we find the accuracy drops to worst-case at around 10% from the best case at around 98% by a voltage reduction of 0.04V.

D. Vulnerability on CIFAR-10

Fig. 9 shows the result of CIFAR-10. In the second step, besides deepen and widen the BCNN and LBPNet, we reduced the size of MLP classifiers to two layers of 512 and 10 neurons. Only one batch normalization layer is preserved so that the training process is accelerated and the vulnerability of the binarized convolutional layers and local binary pattern layers can be more prominent. The structure of BCNN is the

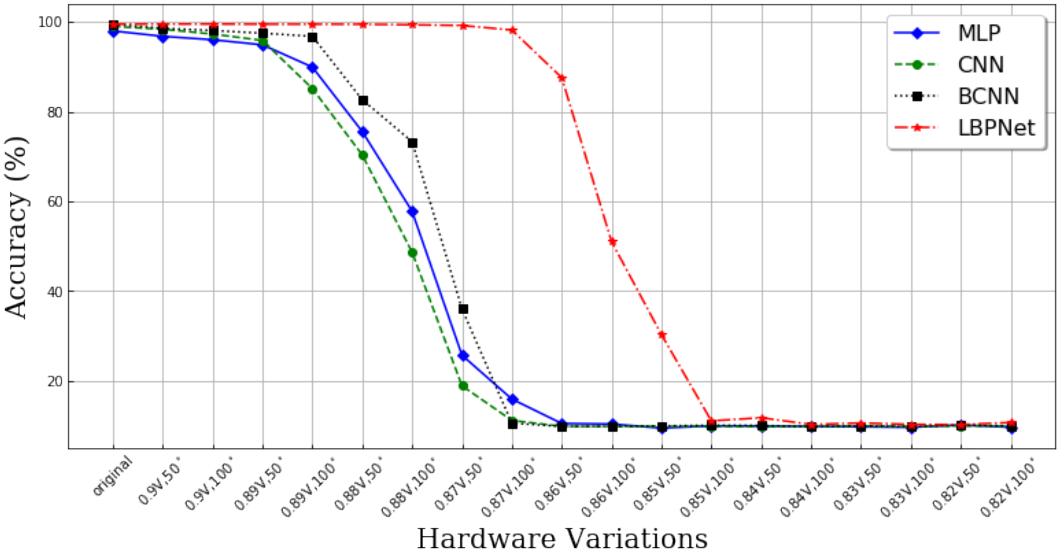

Fig. 8. HNN accuracy as a function of dynamic variations on MNIST.

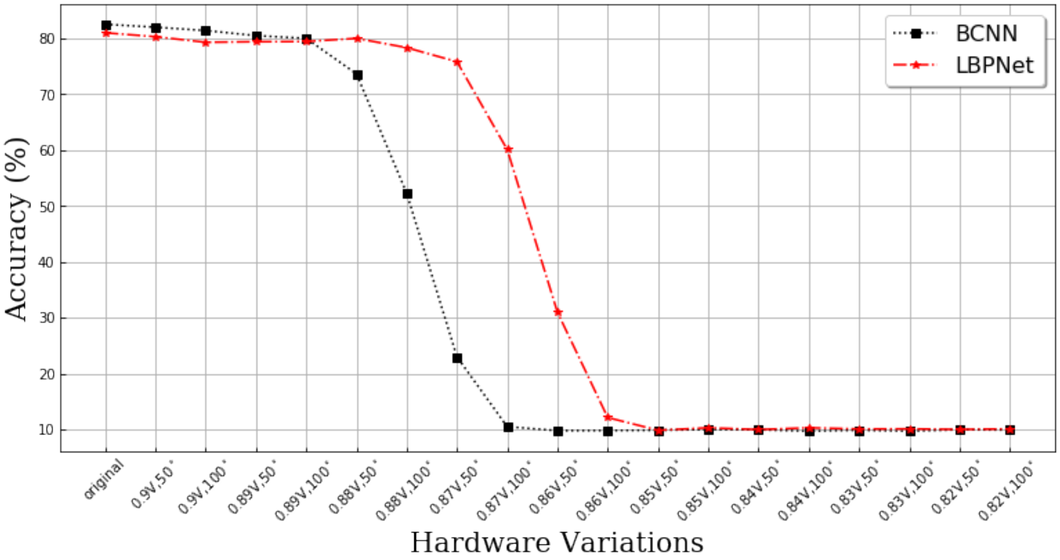

Fig. 9. HNN accuracy as a function of dynamic variations on CIFAR-10.

same with the structure in the original BNN paper except for the simplified fully-connected classifier. We stack the LBPNet to 10 layers and utilized an ensemble of 15 sets of random projection maps to achieve a competent accuracy with the BCNN.

The initial accuracy of BCNN and LBPNet is around 81%. As the hardware variation increases, BCNN's classification ability start to degrade after ($0.89V$, $50°$C), which is not far from the result in the first experiment. However, the immunity of the deeper and wider LBPNet becomes less robust since LBPNet's accuracy starts to fall off the cliff at ($0.88V$, $50°$C). In other words, if overlapping Fig. 8 and Fig. 9, we can see the curves of BCNN and LBPNet recede to the left, and the extent of degradation for LBPNet is more obvious. Although there is still a gap between the two curves, the gap is reduced because the network depth of LBPNet is grown to more than twice of that in the first experiment, but the depth of BCNN remains the same. The classifier in a deeper network would collect more errors than in a shallow network.

E. Discussion of BNN and LBPNet

The binarized values and operations in BCNN rectify a portion of the injected errors and thereby enhances the robustness. Specifically, the 1-bit multiplication and 1-bit accumulation again dilute the impact of the injected errors. Moreover, when the binary activation function converts the erroneous inner-product sum or convolution sum, only the sign inversion changes the activation output. That is, the total of injected errors collected by the activation function must be strong enough to invert the sign; otherwise, the activation output remains the same without the error injection.

The immunity of LBPNet outperforms the other models with a remarkable gap. There are multiple causes contribute to this immunity that can be qualitatively justified through a re-visit of the details in an LBP Layer. The comparison is simulated with the sign bit from the adder's subtraction output. Then, the sign bits corresponding to an LBP kernel are produced by adders in parallel and form a bit sequence to represent an integer on the output feature map. Whenever the adder is stochastically selected for an error injection, the sign bit is flipped randomly according to a uniform distribution. Therefore, an injected error can only affect a single bit rather than an output value as in MLP and CNN. Furthermore, if the selected bit is not the most significant bit (MSB) of the output value, the effect of error injection is scarce. On the other hand, the sign bits are combined with a bit shift and a logic OR operations in parallel, which are relatively less affected by the hardware variations given their circuitry simplicity and not discussed in the scope of this work. The absence of accumulation helps LBPNet to preclude the error accumulation and hinder the propagation of errors.

V. Related Work

Various works study the vulnerability of hardware neural networks under logic errors induced by inexact design [2], [7], [12]. [2] substituted the regular multipliers with inexact multipliers that provide inexact logic but with less hardware cost. [7] further optimized such design with a uniform structure suitable for hardware implementation. [12] provided a framework for hardware neural network designers to choose which parts were suitable for an approximation that led to less impact on accuracy based on a criticality ranking. These works intentionally designed inexact hardware and introduced logic errors in exchange for less hardware cost.

Compared to logic errors, timing errors were less exploited in neural networks because of its unpredictability and uncertainty [4]. Logic errors could be determined once the design is fixed, but timing errors can only be obtained through simulations. A retraining-based method has been proposed to mitigate the timing errors in hardware neural networks [11]. However, these works assumed a fixed timing variation for each gate without considering hardware variations as the root cause, which might be unrealistic.

In summary, there have been no prior works assessing the neural network vulnerability to dynamic operating condition variations. In this work, we do not introduce the errors intentionally but focus on the unintentional timing errors caused by hardware variations. We link the timing errors with low-level hardware variations and characterize them under different operating conditions and present the importance of considering variations when designing hardware neural networks.

VI. Conclusion

Threats to Validity: We mainly focus on variation-induced timing errors in computation logic. However, the timing errors could also occur in control logic, which might lead to more severe accuracy drop or malfunction. Fortunately, it was observed that control logic only contributes a small set of critical paths [11], making it less vulnerable to timing errors.

Promising Solutions: The observations and discussion in previous sections have enlightened us several directions to strengthen the immunity of the voltage and temperature variations.

1) Considering the experiments on the MLP and CNN, we can increase the fan-in of each accumulation to dilute the impact of hardware variations. However, this trend conflicts with pruning, which is a prevailing model reduction method. People need to be aware of the side effects of pruning includes the degradation of network vulnerability.
2) Binarizing the network also dilutes the timing error's impact, and it works for both MLP and BCNN. More generally, quantization of a network not only make the nework hardware-friendly but also increase its vulnerability.
3) While deepening a network structure can usually increase classification accuracy, we need to keep in mind that the increase of depth will reduce the immunity to hardware variations.
4) Another method is to adopt LBPNet because the lack of floating number MAC operations and the high parallelism in the LBP operations have demonstrated that both the error injection and propagation in LBPNets can be limited effectively.

Future Work: In this work, we focus on assessing the effects of hardware variations on neural network performance. The next question is how we can mitigate such timing errors. For future work, we focus on integrating the timing errors as a vector for backpropagation to enable an adaptive training method. Moreover, we plan to design a reconfigurable architecture that can automatically select suitable weights for a given voltage and temperature from a set of pre-stored weights.

References

[1] Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 4:18–27, 2011.

[2] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna V Palem, Olivier Temam, and Chengyong Wu. Leveraging the error resilience of neural networks for designing highly energy efficient accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1223–1235, 2015.

[3] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[4] Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh K Gupta. Slot: A supervised learning model to predict dynamic timing errors of functional units. In *the Conference on Design, Automation, and Test in Europe (DATE)*, pages 1183–1188. IEEE, 2017.

[5] Jeng-Hau Lin, Justin Lazarow, Yunnan Yang, Dezhi Hong, Rajesh Gupta, and Zhuowen Tu. Local binary pattern networks. *the IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2020.

[6] Jeng-Hau Lin, Atieh Lotfi, Vahideh Akhlaghi, Zhuowen Tu, and Rajesh K Gupta. Accelerating local binary pattern networks with software-programmable fpgas. In *the Conference on Design, Automation, and Test in Europe (DATE)*, pages 1112–1117. IEEE, 2019.

[7] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 7, 2016.

[8] Taiga Nomi. tiny-dnn. https://github.com/nyanp/tiny-cnn, 2016.

[9] Timo Ojala, Matti Pietikäinen, and David Harwood. A comparative study of texture measures with classification based on featured distributions. *Pattern Recognition*, 29(1):51–59, 1996.

[10] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[11] Ying Wang, Jiachao Deng, Yuntan Fang, Huawei Li, and Xiaowei Li. Resilience-aware frequency tuning for neural-network-based approximate computing chips. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[12] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. Approxann: an approximate computing framework for artificial neural network. In *the Conference on Design, Automation, and Test in Europe (DATE)*, pages 701–706, 2015.