Soft Error Resilient Deep Learning Systems Using Neuron Gradient Statistics

Chandramouli Amarnath, Mohamed Mejri, Kwondo Ma and Abhijit Chatterjee
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332–0250
Email: chandamarnath@gatech.edu, mmejri3@gatech.edu, kma64@gatech.edu, abhijit.chatterjee@ece.gatech.edu

***Abstract*—Deep learning techniques have been widely adopted in daily life with applications ranging from face recognition to recommender systems. The substantial overhead of conventional error tolerance techniques precludes their widespread use, while approaches involving median filtering and invariant generation rely on alterations to DNN training that may be difficult to achieve for larger networks on larger datasets. To address this issue, this paper presents a novel approach taking advantage of the statistics of neuron output gradients to identify and suppress erroneous neuron values. By using the statistics of neurons' gradients with respect to their neighbors, tighter statistical thresholds are obtained compared to the use of neuron output values alone. This approach is modular and is combined with accurate, low-overhead error detection methods to ensure it is used only when needed, further reducing its cost. Deep learning models can be trained using standard methods and our error correction module is fit to a trained DNN, achieving comparable or superior performance compared to baseline error correction methods while incurring comparable hardware overhead *without* needing to modify DNN training or utilize specialized hardware architectures.**

***Index Terms*—Neural Networks, Fault Tolerance, Resilience, Soft Errors**

I. Introduction and Prior Work

The increasing use of Deep Neural Networks (DNNs) in safety critical applications such as autonomous driving [1] has drawn attention to their vulnerability to soft errors. To achieve high accuracy in these tasks, DNNs rely on a large number of Multiply-Accumulate (MAC) and activation operations for each input [2]. For safety critical systems such as autonomous driving, these operations are required to be performed at a high level of accuracy in the presence of computation (soft) errors [3]. To achieve this, low-overhead, low-impact online methods for error correction must be built to provide resilience against errors that may impact DNN accuracy [4].

Soft error detection methods in the dot product ensemble and activations of DNNs drawing on prior Algorithm-Based Fault Tolerance (ABFT) [5] techniques were first explored in [6], [7]. Related later work in [8] used a 'sanity neuron' to detect soft errors in the dot-product ensemble of DNNs. This work has been extended in [9] to detect errors across DNN layer computation units including activations. However, network retraining with extra hyperparameters is needed in this case. The use of predictor-based checks in DNNs for statistical detection of errors and security threats was explored in [10].

Robustness to parameter variation in DNN execution has been examined in [11], using dynamic fixed-point representations and device variability aware training to enhance DNN resilience. Hessian-based sensitivity metrics have been formulated in prior work [12] to allow prioritization and protection of sensitive DNN parameters and computations, enhancing resilience to parameter variation. However, [11] and [12] have not been tested for compute errors. There has also been work on algorithmic noise tolerance (ANT) [13], [14] for DNNs. Low precision redundant computations are used to both detect and correct errors in a range of digital signal processing algorithms. The focus of ANT is on errors induced by voltage overscaling for ultra low power operation as opposed to radiation or noise-induced soft errors. Faults in DNN systolic array structures have been rapidly detected using small test sets ($\sim$0.1% of the test dataset) to verify the functional safety of the neural network [15]. However, unlike our proposed scheme, this approach is applicable only to DNN faults rather than on-line soft error resilience. Ranger [16] uses selective range restriction of DNN layer computations with restriction ranges derived from DNN performance on training data. This reduces the severity of critical faults that affect high-order bits in DNNs with low overhead. However, Ranger incurs a tradeoff between resilience and error-free accuracy that systems involving error detection and correction (such as our proposed approach) do not.

Soft error resilience in the activations of a DNN is addressed in [17]; erroneous activations are identified using correlations in activities of neighboring neurons and suppressed (by setting to zero), requiring modification of the DNN training process and specialized hardware for on-line error resilience. Resilience-aware computation scheduling, exploiting fault-tolerant device parts for sensitive computations has been explored in [18], allowing a flexible tradeoff between reliability and efficiency. However, this assumes a subset of processing units can be hardened against single-event upsets and may fail under high error rates. The use of median filtering (median feature selection), involving training with median filtering layers after each DNN layer computation block has been explored in [19]. This provides high resilience to soft errors and incurs low computation overhead, but requires alteration of

978-1-6654-7355-2/22/$31.00 ©2022 IEEE

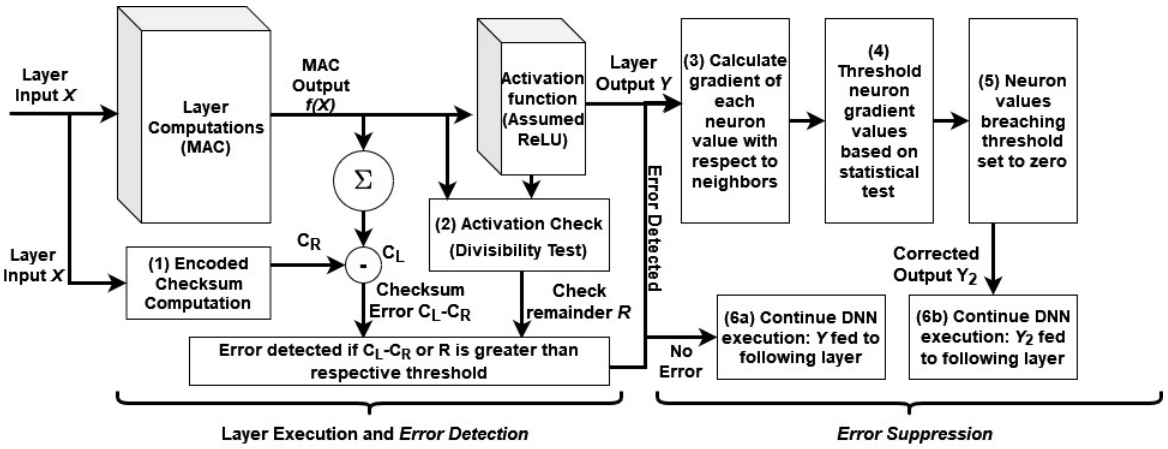

Fig. 1: Proposed error resilience framework: Concurrent error detection is performed using encoded checks on DNN operation, and the presence of an error triggers the suppression system. Neurons with anomalous gradients (determined by a statistical test based on trained DNN outputs on training data) have their values set to zero before DNN computation continues.

DNN training to ensure that inferencing accuracy is unaffected by the median filter layers. Recent work using an opportunistic parity bit achieves high coverage detection and suppression of bit errors in DNN weights with zero storage overhead, but can be masked and does not cover activation errors [20].

In contrast to prior work, this paper presents a flexible approach to DNN error resilience, using statistical analysis of neuron outputs with respect to their neighbors (neuron gradients with respect to their neighbors) to localize and suppress errors. The proposed approach achieves *comparable or superior performance* to median filtering [19] on benchmark DNNs with comparable hardware overhead. Unlike prior work [17], [19], the proposed approach *does not require modification of DNN training or hardware* and can be combined with error detection mechanisms [10] to reduce the effective computational overhead, invoking error correction only when a soft error in DNN computation is detected. As with prior work, we consider error resilience in image classification DNNs.

Section II presents an overview of the proposed approach, and Section III presents the detailed error resilience flow including implemented error detection methods. Section IV discusses experimental results on benchmark DNNs and FPGA hardware, and we conclude in Section V.

II. Overview

The presented error resilience framework (shown in Fig. 1) is composed of two steps: **(1)** *Error detection*, which is used to trigger **(2)** *Error Suppression* when an error is detected. This reduces the number of memory accesses and the compute overhead compared with an "always-on" error resilience approach. This section begins with a brief overview of the error detection setup followed by an overview of the error suppression system that forms the main contribution of this research. In Fig. 1, the DNN MAC operations (dense or convolutional) produce an output $f(X)$ from the layer input X and the activations (assumed ReLU) produce a final output Y. These are used for concurrent error *detection* and *suppression*.

Error detection in DNN layer computations is performed in two phases, illustrated in Blocks 1 and 2 of Fig. 1. *First*, for the MAC computations of a DNN layer (dense or convolutoinal), encoding methods similar to [7] and [8] (Block 1 of Fig. 1) are used for error detection in linear weight-bias computations. These methods produce a checksum value (C_R in Fig. 1) from an encoding of the DNN weight matrix [7]. For convolutional layers, an additional convolutional neuron is used to generate C_R as in [8], with kernel weights equal to the sum of the weights of the convolutional layer kernels. This is compared against the weighted sum of all DNN layer outputs (C_L), which by design is equal to C_R under error-free operation. If the absolute value of $C_L - C_R$ is greater than a threshold, an error is assumed to have occurred in DNN MAC computations (weight-bias multiplication or convolution). These self-checking codes allow rapid, low-overhead error detection. The *second* step (Block-2) involves error detection in DNN activations. In this work, error detection is presented for ReLU activations. The ReLU formulation is $y = max(0, x)$, so that under ideal conditions the ReLU output y is perfectly divisible by x. If y is not perfectly divisible by the activation input x (leaves a remainder $R \neq 0$ greater than a threshold) an error is flagged in DNN activation, since the random nature of soft errors makes then highly unlikely to produce such an outcome.

Error suppression is invoked if the error detection systems flag an error present and forms the *core contribution* of this work. In Fig. 1 this is the block where $C_L - C_R$ and R are compared with their respective thresholds to check for errors. If no error is present, the DNN computations continue by passing the layer output Y to the following layer (Block 6a). If an error is present, error suppression begins by processing Y in Block-3. Here the gradient of each neuron's output value (each value in the output Y) with respect to the corresponding value of neighboring neurons is computed. *Neighboring neurons* of a given neuron are defined as adjacent neurons in a circularly ordered list of neurons in the same layer defined in the model's code. For dense layers, the gradient is the difference between neighboring neuron outputs. For convolutional layers, it is the difference between corresponding pixel values in the

convolutional kernel output. This gives a set of gradient values of the same dimension as the DNN layer output, which are sent to Block-4 for diagnosing which neuron output values exhibit errors. The gradient values are compared with predetermined thresholds for statistically determining (via Student's t test [21]) which of these gradients is anomalous (different from expected, beyond statistical thresholds). Anomalous gradients that are deemed to take place from compute errors in the corresponding neuron are corrected by setting that neuron value (or pixel value, for convolution outputs) to zero in Block-5, producing a modified layer output Y_2 that is sent on for further DNN computations in the following layers in Block 6b. The statistical thresholds used here are determined from the mean and standard deviation of the neuron gradients across the training dataset. The statistical thresholding incurs memory overhead for comparison with upper and lower thresholds for each element in the layer output due to the need to access the upper and lower threshold values, each of equal size to the layer output for a single input. The above error resilience technique can be applied to all but the last DNN layer, similar to [19]. *The use of gradient values allows more flexible thresholds that accommodate a larger range of values of neuron outputs than the use of just neuron values.* This approach is illustrated in further detail below, beginning with an overview of the DNN computations involved.

III. Approach Details

A. Deep Neural Network Layer Computations

We consider both dense and convolutional layers of DNNs. The layer input is denoted by X, the output of its MAC computations is denoted by $f(X)$ and the final activation output is denoted by Y. MAC computations in a *dense* layer consist of weight-bias matrix multiplication and addition. The weights W are multiplied with the input X and added to the bias vector b such that $f(X) = WX + b$.

For a *convolutional* layer, each input and output in the batch is in the form of a 'stack' of 2-D images (3-D image tensor). The layer is assumed to consist of C_{out} neurons, the input X is assumed to have a shape (C_{in}, I_h, I_w) and MAC computation output $f(X)$ has a shape (C_{out}, O_h, O_w), where C_{in} is the number of input channels, C_{out} the number of output channels (convolutional neurons), I_h and I_w the input image height and width respectively, and O_h and O_w the output image height and width respectively. Each convolutional neuron is associated with one kernel W for each input channel and a bias b. The output for the ith neuron MAC computation for a single input is thus $f_i(X) = b_i + \sum_{k=0}^{C_{in}-1} W_{i,k} * X_k$ where $*$ is the convolution operator, giving a 2-D output for the 'stack' of 2-D inputs to the neuron.

DNN activation follows the MAC computations and is assumed to use the ReLU operation $y = max(0, x)$, applied elementwise to $f(X)$ to give the layer output Y.

B. Error Detection Via Encoded Checksums

In a dense layer, the matrix multiplication MAC operation is encoded by adding an extra row to the weight matrix [7]. The additional row of the weight matrix is given by αW and the additional bias element by βb with α and β called the encoding weight vectors. For a layer with N neurons and M inputs, the weight matrix is of dimension $N \times M$. The encoding is thus generated as $C_L = \alpha W X + \beta b$. Here, $\alpha = \beta = [1, 1, ..., 1] \in \mathbf{R}^{1 \times N}$, giving $C_L = \sum f_i(X)$, or the sum of all elements of $f(X)$. In the absence of an error, $C_L - C_R \approx 0$ to within the margin of error of digital quantization. $C_L - C_R$ is thresholded at an empirically determined value (here 0.5) to detect an error.

In the DNN convolutional layer, the output $f(X)$ is a 3-D image tensor of shape (C_{out}, O_h, O_w). Error detection is implemented based on [8]. The encoding is calculated as a summation of all values in the 3-D tensor, with weights given to each neuron, so that $C_R = \sum_{i=0}^{C_{out}} \alpha_i \sum_{j=0}^{O_h} \sum_{k=0}^{O_w} f_{i,j,k}(X)$, where α_i is a weight assigned to the ith neuron and $f_{i,j,k}(X)$ denotes the indice (i, j, k) of the MAC output tensor, with the MAC operation discussed for convolutional layers in Section III-A. C_R is thus the weighted summation of all elements in the MAC output tensor $f(X)$. This reduces to: $C_R = \sum_{j=0}^{O_h} \sum_{k=0}^{O_w} ((\sum_{i=0}^{C_{out}} \alpha_i W) * X + \sum_{i=0}^{C_{out}} b_i)$ which under the condition $\alpha_i = 1 \forall 1 \leq i \leq C_{out}$ (used in this work) is simply a convolutional neuron operation using a kernel $W_c = \sum_{i=0}^{C_{out}} W_i$, whose kernel weight is the sum of all other kernels' weights and whose bias b_c is the sum of all other kernels' biases, $b_c = \sum_{i=0}^{C_{out}} b_i$. This additional convolution generates $C_L = W_c * X + b_c$. In the absence of error $C_L - C_R \approx 0$ to within a margin of error for digital quantization. This $C_L - C_R$ is thresholded at an empirically determined value (in this work 0.5) to detect an error.

For activation errors we use the divisibility of the output of ReLU activation for error detection. The ReLU operation is applied elementwise to each element $f(x)$ of the MAC output $f(X)$ to produce each element $y = max(0, f(x))$ of the output Y. Each y is thus divisible by $f(x)$ so that $y \% f(x) \approx 0$, where $\%$ denotes the modulo operation. An error is detected if the summation $\sum(y \% f(x))$ exceeds a predetermined threshold (here set as 0.5).

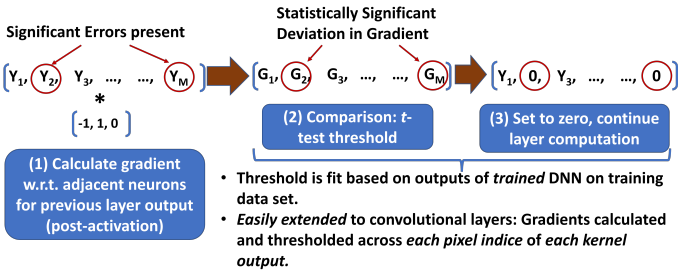

Fig. 2: Overview of DNN neuron gradient driven error localization and suppression applied to dense layers: Convolutional layers necessitate this operation for each pixel indice across all layer kernels (along the channel-dimension).

C. Neuron Gradient Driven Error Suppression

Error localization and suppression is performed upon error detection in layer computations. Fig. 2 shows the proposed error suppression framework applied to a dense layer, using the layer outputs $Y = \{Y_i\}_{i=1}^{M}$ as inputs. First (in step (1) of Fig.

2), the system calculates the gradient of each value of Y with respect to the corresponding value in one of the neighboring neurons (the difference between the two values is taken as the gradient). For a dense layer (output is a vector), the gradient is calculated as the linear convolution $\Delta Y = [-1, 1, 0] * Y$, giving the gradient as the difference between the neuron output value and the output value to its 'left' (left gradient). This implicitly assumes that two adjacent neurons will not show significant errors that do not significantly affect the gradient between them. In Fig. 2 this is the gradient vector $[G_1, G_2, ..., G_M]$ in Step 2. This gradient computation is performed using circular padding (hence the length-3 kernel) to ensure that ΔY and Y have the same dimensions and also to capture gradient information for neurons at the layer edges with respect to one another rather than a constant (which would remove the advantages of gradients as opposed to absolute values). For a convolutional layer, the gradient is given by $\Delta_{i,j} Y = [-1, 1, 0] * Y_{:,i,j}$, where 1-D convolution is carried out between adjacent neuron values of the same pixel indice for all pixel indices (i,j). The number of filter groups is set to 1. This is equivalent to a Conv1d layer with number of groups=1 applied to each $Y[i, j, k]$ where (i, j) are image pixel indices and k denotes the channel index. The vector of gradients ΔY is used to localize errors.

To do this (Step (2) of Fig. 2), each value of ΔY is thresholded using Student's t test [21]. The thresholds are determined using the mean and variance of ΔY across clean training data in the absence of errors, and are set as $\mu \pm k_\Delta \sigma$, where μ is the mean of the relevant value in ΔY and σ is its standard deviation. k_Δ is a user defined tuning parameter. Values that breach the thresholds are deemed *potentially erroneous*. Due to the nature of the convolutional kernel $[-1, 1, 0]$, potentially erroneous values will occur in pairs for each error. One element of the pair will be the gradient of the erroneous value with respect to a clean neighbor. The other adjacent element is the gradient of a clean neighbor with respect to the erroneous neuron. The shape of the left gradient ensures that the second (latter) element of every pair is not erroneous, and the first element of each of these pairs is erroneous. This gives a set of indices for erroneous values in Y that are then set to zero (Step (3) of Fig. 2). This modified vector Y_2 is sent on to the next layer to continue DNN inference.

Fig. 3 shows an example of soft error injection into two neurons in the final hidden layer of LeNet-5 [22] trained on the MNIST [23] dataset. The 1st, 6th and 25th bits are flipped in the 3rd and 20th neurons of the error free outputs (top plot) of the final hidden layer prior to DNN activation, causing misclassification to class '4' (correct class '7') due to the resultant erroneous layer output (bottom plot). The plots display neuron values (y-axis) for each neuron indice (x-axis), showing the deviation caused by soft error injection.

Fig. 4 shows the effect of error suppression. Soft errors are injected into nominal outputs (Plot (1)) identically to Fig. 3, giving the plot (2) (errors highlighted in red). Neuron gradients are found via convolution to get Plot (3) and compared against the upper and lower statistical thresholds (dotted lines in magnified inset) to flag potentially erroneous neurons. Erroneous neurons are set to zero to get Plot (4) of error-suppressed outputs, giving the correct classification ('7') (step (5)).

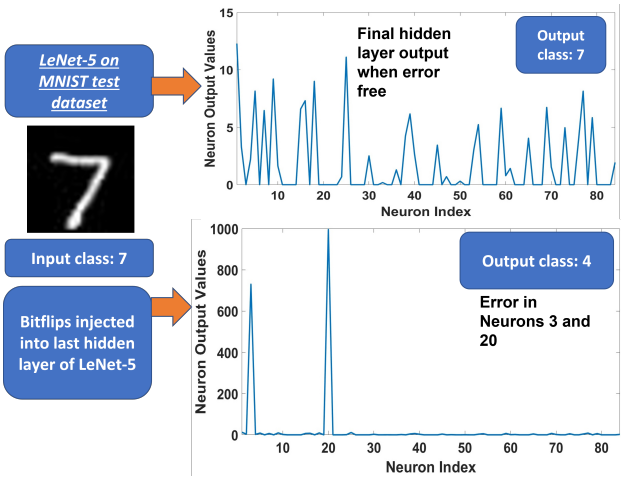

Fig. 3: Sample error injection into LeNet-5 hidden layers, causing a misclassification when the input is of class '7', forcing the DNN to classify to class '4'. Bits were flipped in the 3rd and 20th neuron out of the 84 in LeNet-5's final hidden layer.

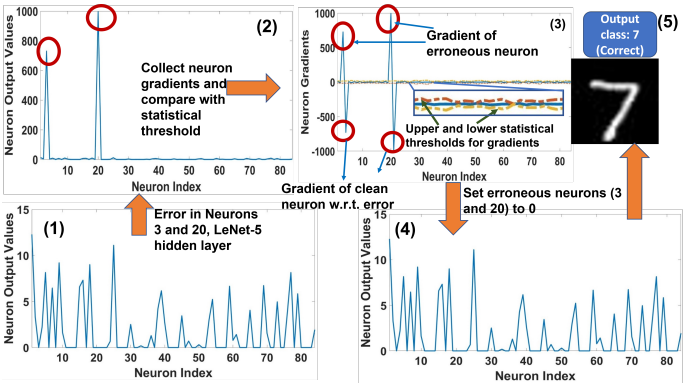

Fig. 4: Sample error suppression in LeNet-5 hidden layers, applied to the error injected in Fig. 3. The neuron gradients are potentially erroneous at four locations, corresponding to the two errors in neurons 3 and 20. The neurons 3 and 20 are then set to zero, allowing correct classification to class '7'.

IV. Experimental Results

The proposed error resilience framework has been tested using error injection experiments in hidden layer operations for PyTorch [24] operations in CUDA as well as error injection in DNNs running on FPGA hardware. Testing used two DNNs: LeNet-5 [22] trained on the MNIST dataset [23], and ResNet-18 [25] trained on the CIFAR-10 dataset [26]. The proposed framework is compared against median filtering approaches [19] and a network with no error correction mechanisms.

Error resilience was measured using the test accuracy of the DNN under soft errors. Each neuron in the DNN has a given probability of error for each inference. This probability is referred to as the *error rate*. Identical errors are injected using median filtering, using the proposed framework and using no error correction method. These soft errors are modeled as bitflips in DNN computations. For PyTorch, this is 32-bit

floating point arithmetic. For hardware validation, soft errors are injected into 16-bit fixed-point arithmetic. Soft errors are injected as *bit* errors, where a single bit flips in a random position, or as *word* errors, where a group of neighboring bits centered around a random position flip together.

Error injection in PyTorch is performed using the PyTorchFI tool for runtime error injection in DNN inference on CUDA [27]. *Bit* and *Word* errors were injected into the output values of neuron MAC computations with error probability for each output value determined by the error rate. Word errors are injected such that every bit adjacent to an erroneous bit has a high probability (here 75%) to flip and thus be erroneous.

Hardware Validation: Hardware validation was performed on a Xilinx Zynq UltraScale+ MPSoC ZCU104 FPGA , running a multilayer perceptron network across MNIST using gradient based error suppression, median filtering (as a baseline) and no correction method. The network has three hidden layers with 64, 32, and 16 neurons respectively. All computations were encoded in 16-bit fixed point (8 integer bits and 8 decimal bits). Test accuracy under error was recorded running the DNN on the FPGA. Overhead figures (resource utilization, latency, power and energy) were recorded using Xilinx Vivado simulation. For hardware validation, error detection was not implemented. Hardware validation primarily evaluated the performance and overhead of error resilience. Using error detection would lower effective overhead by ensuring error suppression was not invoked under nominal conditions.

A. Error Injection in LeNet-5

Fig. 5 shows results for bit error injection experiments on LeNet-5 hidden layers. The DNN is trained on the MNIST dataset and has a error-free test accuracy of 99.1% for the error free and gradient based suppression cases. The gradient based error suppression system is fit to the trained DNN with $k_\Delta = 4$ for dense layers and $k_\Delta = 6$ for convolutional layers. The median filtering enabled DNN [19] is presented for comparison, with error-free test accuracy of 98.8%. Error rates varied from 0.001% to 0.1%. The DNN showed significant loss in accuracy for higher error rates such as 0.1%. Gradient error suppression achieved comparable results to median filtering (within 1-3%), without any modification to DNN training.

Fig. 6 shows results for word error injection experiments on LeNet-5 hidden layers. Error rates again varied from 0.001% to 0.1%. The DNN showed significant loss in accuracy earlier than for bit errors, dropping as low as 72% under high error rates like 0.1%. The use of gradient error suppression for DNN resilience achieved comparable results to median filtering here as well, staying within 1-2% of the DNN test accuracy using median filtering. More severe error rates in both bit and word error cases show a degradation in performance for gradient-based error suppression due to the higher number of neurons suppressed to zero, impacting inference.

B. Error Injection in ResNet-18

Fig. 7 shows results for bit error injection on ResNet-18. The DNN is trained on CIFAR-10 and has an error-free test accuracy of 85%. Here $k_\Delta = 4$. The gradient based error suppression system is fit to this trained DNN and has identical error-free accuracy. The median filtering enabled DNN [19] is presented for comparison (error-free test accuracy of 85.4%). For ResNet-18, *the median filtering module was placed after each residual block* (consisting of two convolutional layers and input addition), due to the module impacting error-free accuracy if placed after each convolutional layer. Error rates varied from 0.001% to 0.1%. The DNN showed high loss in accuracy without correction, whereas gradient based error suppression keeps DNN accuracy above 75% for all test cases.

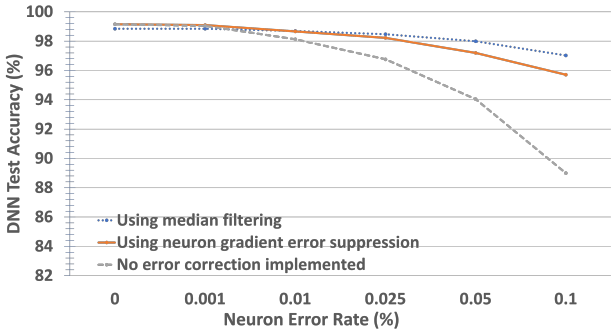

Fig. 5: Bit error injection on LeNet-5 (MNIST): Bit errors were injected at varying error rates (probabilities of neurons showing error) on LeNet-5 hidden layer computations. The proposed approach was tested against median filtering [19] as well as the baseline DNN (no error correction). Comparable performance (within 1-3%) to median filtering is achieved *with no requirement to modify DNN training*.

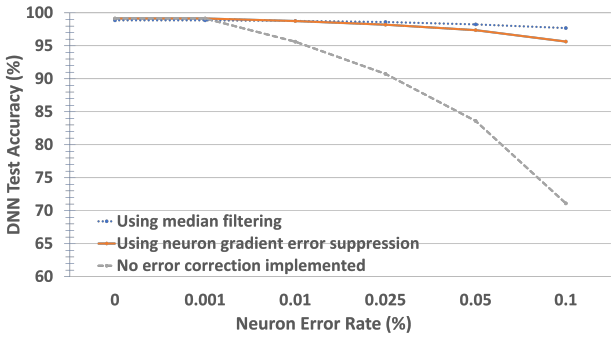

Fig. 6: Word error injection on LeNet-5 (MNIST): Word errors were injected at varying error rates into LeNet-5 hidden layer computations. Comparable performance (within 1-3%) to median filtering [19] is achieved *with no requirement to modify DNN training*.

Fig. 8 shows results for word error injection experiments on ResNet-18 hidden layers. Error rates varied from 0.001% to 0.1%. The DNN showed more severe accuracy loss here than for bit errors. Gradient error suppression achieved superior results to median filtering and similar accuracy to the bit error case. Under high error rates, test accuracy drops to just above 60% for gradient based error suppression, unlike the collapse in accuracy for the network without error correction.

This superior performance to median filtering in comparison to the case in Section IV-A is partly due to the placement of filter modules after every pair of convolutional layers (each residual block), and in part due to the greater number of neurons in ResNet-18. Placement of median filter modules after

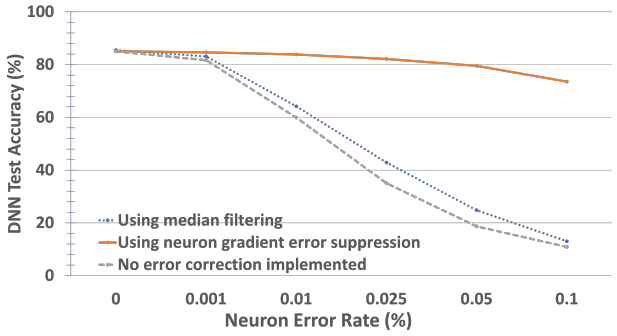

Fig. 7: Bit error injection into ResNet-18 (CIFAR-10): Bit errors were injected at varying error rates (probabilities of neurons showing bit error) on ResNet-18 hidden layer computations. The proposed approach was tested against median filtering (implemented after each residual block) [19] as well as the baseline (no correction method used) network. Superior performance to median filtering is achieved *with no requirement to modify DNN training*.

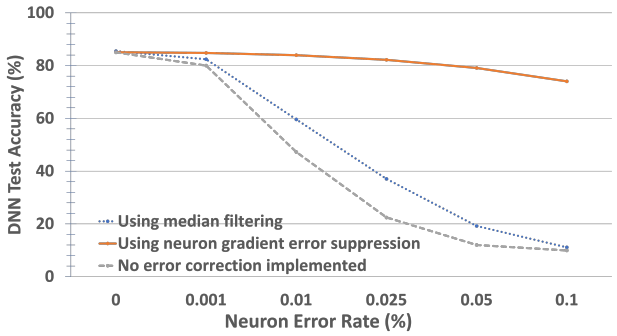

Fig. 8: Word error injection on ResNet-18 (CIFAR-10): Word errors were injected at varying error rates into ResNet-18 hidden layer computations. Superior performance to median filtering [19] is achieved *with no requirement to modify DNN training*.

each layer of ResNet-18 led to a significant drop in accuracy during training, thus leading to use of median filter modules after each residual block. The greater number of neurons allows suppression (zeroing) to be done more aggressively under high error rates without high impact on accuracy.

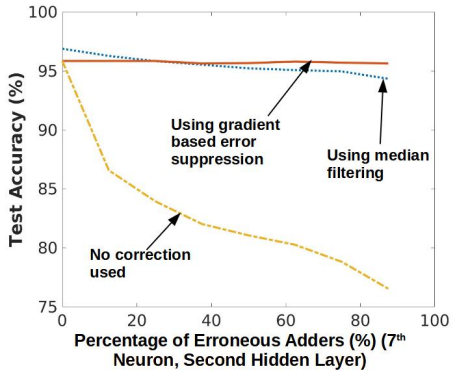

Fig. 9: Neural network performance on FPGA when using gradient based suppression, median filtering and no correction when varying the percentage of adders in the 7th neuron of the second hidden layer that show errors. The gradient approach achieves slightly better performance over median filtering.

Method	Configurable Logic Blocks (CLB)	Latency (ms)	Power Consumption (W)	Energy (mJ)
Gradient-based Approach	8511	5.869	4.219	24.761
Median Filtering	8501	5.866	4.277	25.089
No Correction	8379	5.865	4.198	24.621

TABLE I: Hardware overhead: Calculated for the FPGA implementations against the baseline (median filtering [19]).

C. Hardware-Based Error Injection

Fig. 9 shows the performance of the DNN on FPGA under error injection using median filtering, using the proposed approach (with $k_\Delta = 4$) and using no error correction. The percentage of erroneous adders in a chosen neuron of the second hidden layer was varied from 0 to 87.5%. Bit errors were injected in these erroneous adders. The target neuron was chosen randomly and fixed for the entire testing process (Here that was the 7th neuron of the second hidden layer). The faulty adders' positions varied from 0 to 64 and are chosen randomly each time. The 2nd MSB bit in the integer part was chosen for flipping to insert the bit error. It is seen from Fig. 9 that the proposed gradient based approach performs slightly better than median filtering as the number of faulty adders increases. The proposed approach thus performs slightly better than state of the art methods on hardware for small networks. As seen in Section IV-B, larger networks can lead to superior performance from the gradient based approach due to allowing more aggressive zeroing of erroneous neuron values.

Table I shows the overhead (collected in Xilinx Vivado simulation) for the proposed resilience system measured against the overhead of median filtering on FPGA. The overhead for the network with no error correction is shown for reference. Resource utilization is calculated as the number of configurable logic blocks (CLBs) used by the Xilinx Vivado simulation. The gradient based approach shows lower energy consumption, marginally higher CLB usage and lower power consumption than median filtering. Latencies for all three scenarios are almost identical. Table I shows that the proposed approach has slightly lower overhead than median filtering *when operating in always-on mode*. The use of error detection systems in conjunction with this (as in Fig. 2) can further lower effective overhead and memory access requirements.

V. Conclusion

This work presents a DNN error resilience approach that achieves comparable or superior performance to state of the art methods *without requiring* specialized hardware or modification of DNN training. The method is validated on benchmark DNNs using published error injection methods as well as on FPGA hardware. Future work envisions testing this approach against fault aware training [11] and permanent faults.

Acknowledgment

This research was supported by the Semiconductor Research Corporation under Auto Task 2892.001 and in part by the U.S. National Science Foundation under Grant No. 2128149 and Grant S&AS:1723997.

REFERENCES

[1] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *The IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[3] ISO, "Road vehicles – Functional safety," 2011.

[4] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018, pp. 1–6.

[5] V. Nair and J. Abraham, "Real-number codes for fault-tolerant matrix operations on processor arrays," *Computers, IEEE Transactions on*, vol. 39, no. 4, pp. 426–435, Apr 1990.

[6] S. Pandey, S. Banerjee, and A. Chatterjee, "Reinn: Efficient error resilience in artificial neural networks using encoded consistency checks," in *2018 IEEE 23rd European Test Symposium (ETS)*, 2018, pp. 1–2.

[7] ——, "Error resilient neuromorphic networks using checker neurons," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, 2018, pp. 135–138.

[8] E. Ozen and A. Orailoglu, "Sanity-check: Boosting the reliability of safety-critical deep neural network applications," in *28th IEEE Asian Test Symposium, ATS 2019, Kolkata, India, December 10-13, 2019*. IEEE, 2019, pp. 7–12. [Online]. Available: https://doi.org/10.1109/ATS47505.2019.000-8

[9] ——, "Concurrent monitoring of operational health in neural networks through balanced output partitions," in *25th Asia and South Pacific Design Automation Conference, ASP-DAC 2020, Beijing, China, January 13-16, 2020*. IEEE, 2020, pp. 169–174. [Online]. Available: https://doi.org/10.1109/ASP-DAC47756.2020.9045662

[10] C. Amarnath, M. I. Momtaz, and A. Chatterjee, "Addressing soft error and security threats in dnns using learning driven algorithmic checks," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021, pp. 1–4.

[11] Y. Long, X. She, and S. Mukhopadhyay, "Design of reliable DNN accelerator with un-reliable reram," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*, J. Teich and F. Fummi, Eds. IEEE, 2019, pp. 1769–1774. [Online]. Available: https://doi.org/10.23919/DATE.2019.8715178

[12] S. Dash, Y. Luo, A. Lu, S. Yu, and S. Mukhopadhyay, "Robust processing-in-memory with multibit reram using hessian-driven mixed-precision computation," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 4, pp. 1006–1019, 2022. [Online]. Available: https://doi.org/10.1109/TCAD.2021.3078408

[13] S. Zhang and N. R. Shanbhag, "Embedded algorithmic noise-tolerance for signal processing and machine learning systems via data path decomposition," *IEEE Transactions on Signal Processing*, vol. 64, no. 13, pp. 3338–3350, 2016.

[14] A. Mahmoud, S. K. S. Hari, C. W. Fletcher, S. V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Hardnn: Feature map vulnerability evaluation in cnns," 2020.

[15] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021.

[16] Z. Chen, G. Li, and K. Pattabiraman, "A low-cost fault corrector for deep neural networks through range restriction," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 1–13.

[17] E. Ozen and A. Orailoglu, "Just say zero: Containing critical bit-error propagation in deep neural networks with anomalous feature suppression," in *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020*. IEEE, 2020, pp. 75:1–75:9. [Online]. Available: https://doi.org/10.1145/3400302.3415680

[18] C. Schorn, A. Guntoro, and G. Ascheid, "Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators," *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 979–984, 2018.

[19] E. Ozen and A. Orailoglu, "Boosting bit-error resilience of dnn accelerators through median feature selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, pp. 1–1, 11 2020.

[20] S. Burel, A. Evans, and L. Anghel, "Zero-overhead protection for cnn weights," in *2021 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2021, pp. 1–6.

[21] E. L. Lehmann and J. P. Romano, *Testing statistical hypotheses*, 3rd ed., ser. Springer Texts in Statistics. New York: Springer, 2005.

[22] 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, 1998.

[23] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[24] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[26] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[27] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "Pytorchfi: A runtime perturbation tool for dnns," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, 2020, pp. 25–31.