Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions

Elbruz Ozen and Alex Orailoglu

Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093
elozen@eng.ucsd.edu, alex@cs.ucsd.edu

Abstract— The abundant usage of deep neural networks in safety-critical domains such as autonomous driving raises concerns regarding the impact of hardware-level faults on deep neural network computations. As a failure can prove to be disastrous, low-cost safety mechanisms are needed to check the integrity of the deep neural network computations. We embed safety checksums into deep neural networks by introducing a custom regularization term in the network training. We partition the outputs of each network layer into two groups and guide the network to balance the summation of these groups through an additional penalty term in the cost function. The proposed approach delivers twin benefits. While the embedded checksums deliver low-cost detection of computation errors upon violations of the trained equilibrium during network inference, the regularization term enables the network to generalize better during training by preventing overfitting, thus leading to significantly higher network accuracy.

I. Introduction

DNNs (deep neural networks) are perhaps the most popular AI (artificial intelligence) algorithms in the recent past due to their outstanding performance and broad applicability to many practical, but challenging, AI problems. While the potential application areas of DNNs span problems such as image processing, speech recognition, and natural language processing, a subset of the DNNs, CNNs (convolutional neural networks), particularly specialize in computer vision tasks that involve processing and interpreting complex visual data. Deep neural networks can be observed running on various hardware architectures in a myriad of real-life applications, including safety-critical domains such as autonomous driving [1].

The automotive industry enforces strict safety requirements on its products [2]. Every electronic chip and software algorithm in an automotive application is carefully tested and equipped with rigid safety features to prevent the catastrophic consequences of a potential failure. Electronic system designers usually employ ECC [3] (error correction codes) to prevent silent data corruption (SDC) in the memory elements, and the execution path is protected by replicating the critical components (full duplication, TMR) [4] despite the substantial area and power overhead incurred. These solutions deliver error detection and protection against transient SEUs (single-event upsets), transient timing errors, and permanent hardware defects that might show up after system deployment. Circuit-level hardening methods [5] do exist to detect and mitigate the effects of transient SEUs or timing errors, but their notable area, delay, and power overheads make them a less appealing solution for consumer products.

Deep learning is computationally expensive and usually necessitates dedicated hardware to meet the performance requirements of real-time applications. While GPGPUs and dedicated DNN engines deliver billions of MAC operations every second, they also comprise a significant portion of the system area and power budget. While the deep neural networks in safety-critical systems require safety mechanisms, modular redundancy fails to constitute an appealing solution for DNN hardware due to significant area and power demands.

We utilize the application-specific characteristics of deep neural networks to introduce fault tolerance at the algorithm level, and achieve significant coverage of error cases on off-the-shelf commercial chips, incurring neither hardware nor significant performance overheads. We summarize our contributions as follows:

- We introduce a simple penalty term into the DNN cost function to train a productive error checking invariant and employ this invariant for error detection.
- We implement a comprehensive error injection framework and utilize it to compare and demonstrate the effectiveness of the checksums through exhaustive error injection experiments.
- We draw attention to our observations that indicate that the introduced penalty term delivers a rather useful and perhaps somewhat unexpected ancillary effect. It acts as a regularizer during DNN training and noticeably improves test set accuracy by reducing overfitting and improving generalization.

II. An Introduction to Deep Neural Networks

Deep neural networks are brain-inspired machine learning algorithms commonly used for classification and regression tasks. The central computation unit in a neural network is called a *neuron*, whose responsibility is taking a weighted sum of its inputs and processing the sum with a non-linear activation function. The neurons are organized as a sequence of layers, with the network getting deeper as the number of layers increases. Modern deep neural networks for image processing [6], [7], [8], [9] heavily rely on two types of layers, namely, *convolutional* and *fully-connected*, where the input is initially processed by a series of convolution layers to extract the useful features, with the fully-connected layers subsequently performing the final classification task. Unlike the fully-connected layers, convolutional layer neurons are

978-1-7281-4123-7/20/$31.00 © 2020 IEEE

only locally connected to the previous layer, and the weight values are shared among different neurons. Deep neural networks also employ other types of operations such as non-linear activations, and frequently pooling and normalization layers. The training procedure involves a *cost function*, which measures the distance between the expected and produced DNN outputs. Categorical cross-entropy is a commonly used cost function for the single-label classification problem and can be expressed as follows for a single training example:

$$-\sum_{c=1}^{M} y_{o,c}\ log(p_{o,c}) \tag{1}$$

In Equation (1), M denotes the total number of classes and $y_{o,c}$ represents the true binary label for example o and class c. For single-label classification, $y_{o,c} = 1$ holds only for a single c for a particular example. $p_{o,c}$ denotes the predicted output probability produced by the output layer (with Softmax activation) of the network. Training is carried by calculating the gradient of DNN parameters through the backpropagation algorithm and updating the parameters to reduce the loss at each step until a minimal point is found.

III. Error Checking with Computation Invariants

We aim to embed invariants into deep neural network computations as these invariants could be utilized at inference time to detect discrepancies in the computations. The errors could be caused by various types of hardware-level faults, but independent of their provenance, these errors are detected as long as they violate the introduced invariant. Let us start our discussion by considering the following invariant for a particular DNN layer:

$$\left\|\sum_{i=1}^{S_l} h_i^l\right\|_F = 0 \qquad ,\forall l \tag{2}$$

In Equation (2), h_i^l denotes the output of the i'th computation unit in the l'th layer and S_l represents the total number of computation units in the l'th layer. In the simplest case, Equation (2) can be considered to indicate that the outputs of a fully-connected layer always sum up to zero. As the summation will be a matrix for the convolutional layers, we use the Frobenius norm to generalize Equation (2) for both fully-connected and convolutional layers by reducing the left-hand side to a single number. This invariant is quite useful for error checking since any single output error is always detected, or in general, any error pattern is detected as long as the individual errors do not cancel each other's footprint on the checksum by accumulating to zero.

The reader will realize that the presented invariant requires the outputs of the computation units to span both positive and negative quantities so that they could potentially sum up to zero. While the computation units with a hyperbolic tangent (*tanh*) activation function produce both positive and negative outputs, the range of the very commonly used ReLU and Sigmoid activation functions is restricted to only non-negative quantities, thus precluding the direct utilization of the presented invariant.

We introduce an alternative invariant which allows us to perform a similar check for a set of non-negative quantities. Let us partition the computation units into two groups at each layer, sum each group separately, and set the norm of their difference to zero as in Equation (3). We refer to this invariant as the *balance checksum*. The balance checksum can detect all errors in a single computation unit; errors in multiple computation units are guaranteed to be detected unless they end up in the same balance of the output partitions.

$$\left\|\sum_{i=1}^{\lfloor S_l/2\rfloor} h_i^l - \sum_{i=\lfloor S_l/2\rfloor+1}^{S_l} h_i^l\right\|_F = 0 \qquad ,\forall l \tag{3}$$

Although the balance checksum is useful for detecting errors, the introduction of such a mathematical property to neural network computations involves significant challenges. First, the balance checksum forces the layer outputs to adhere to a linear invariant without considering the norm operator. Linear invariants are straightforward to introduce in linear systems and are the key component for numerous error detection methods such as ABFT (Algorithm-Based Fault Tolerance) [10]. The linearity of fully-connected and convolution layer operations (excepting the non-linear activation functions) enables us to integrate similar invariants through simple weight and input modifications [11]. However, the scope of such an approach is strictly limited to the linear stages of the computation with data being left unprotected in the non-linear stages. In addition, the linear sessions are frequently interrupted by non-linear activation functions at the end of each layer. As the checksum generation and check operations should be performed within the boundaries of each linear stage, such an approach consequently incurs significant overhead. The mathematical introduction of such an invariant involves substantial challenges for non-linear systems, prodding us to follow a rather distinct approach to tackle this problem.

IV. Training Balanced Output Partitions

Deep neural network training enables us to adjust the DNN parameters so that the network learns to perform the desired task correctly. The cost function plays an integral role in the training stage as parameter updates are continuously undertaken to minimize the cost function. One can set various training goals for the network through simple modifications on the cost function. An extra penalty term in the cost function is frequently used in the regularization methods to simplify the classifier complexity by guiding weight and activations to zero. As a simple model is less likely to suffer from overfitting to the training data, regularization methods evince superior network generalization capabilities to future test examples. We similarly integrate the balance checksum into our target DNN model by including it as a penalty term in the DNN cost function and guiding the network to minimize the checksum error while learning to identify the correct labels in the training data. Our overall cost function can be formulated as follows for a single training example:

$$-\sum_{c=1}^{M} y_{o,c}\log(p_{o,c}) + \lambda\sum_{l=1}^{L}\frac{1}{N_l}\left\|\sum_{i=1}^{\lfloor S_l/2\rfloor} h_{o,i}^l - \sum_{i=\lfloor S_l/2\rfloor+1}^{S_l} h_{o,i}^l\right\|_F^2 \tag{4}$$

The first part in Equation (4) is the categorical cross-entropy for the correct output labels. The second part is a mean square error for the group output differences. The name and function of λ is analogous to the coefficient used in the regularization methods. We use the hyperparameter λ to control the impact of the penalty term on the cost function and prevent the penalty term from hindering the main learning task. In Equation (4), L denotes the number of layers and N_l the output size of each computation unit in layer l. $h^l_{o,i}$ corresponds to the output of the i'th computation unit in the l'th layer for input example o after the activation function. An exception needs to be taken for the outputs of the last layer, forcing us to extract $h^L_{o,i}$ before the application of the Softmax activation function. The sum of the Softmax activation function outputs is always normalized to 1 as shown in Equation (5) with the most likely class obtaining a rather elevated probability; therefore, it is not straightforward to integrate the balance checksum to the last layer after the Softmax function. We embed the balance checksum in the last fully-connected layer before the Softmax function, and as the Softmax layer outputs sum to 1, we check this property to protect data during the Softmax computation.

$$\sum_{c=1}^{M} p_{o,c} = 1 \quad , \forall o \tag{5}$$

Finally, we partition the layers according to neuron (filter) indices; however, any equivalent partitioning scheme works as long as the trained scheme is used for error checking since the neurons (filters) are interchangeable before the training.

V. Error Checking at Runtime

Network training minimizes the additional penalty term in the cost function and consequently helps to balance the output partitions. As the network is required to satisfy the main classification task, the differences between groups typically fail to match zero exactly, instead slightly deviating from it. Our observations indicate that as a result of training with the additional penalty term, the maximum deviation between the groups reduces by two orders of magnitude, thus providing us sufficient resolution to detect significant computation errors. We introduce individual threshold values for each layer and check if the difference between the groups deviates more than the determined threshold rather than focusing on a strict check of equality to zero. We learn the threshold values by profiling the training examples. We perform a full run on the training set to determine the maximum balance deviation at each layer and multiply these values with a small margin before setting them as the error thresholds.

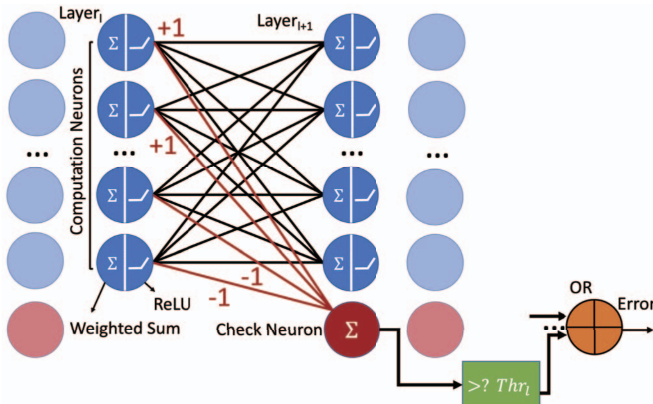

Fig. 1. Checking the balance in fully-connected layers

We introduce some additional modifications on the DNN model as well to detect errors at runtime. The correctness of the computations at each layer is checked by an extra computation unit in the next layer. For fully-connected layers, we employ an additional neuron in the next layer (check neuron) as shown in Figure 1 to check if the balance is satisfied. While the check neuron is connected to the outputs of all the computation units of the previous layer, we set the weight values for the first group connections as 1, and -1 for the other group. The check neuron accumulates both group outputs and calculates the difference at each prediction. We introduce 1×1 convolution filters (check filters) after each convolutional layer to check if the two group outputs are balanced. The channels of the check filter that convolve with the first group outputs are set to 1; the remaining channels are set to -1. In this way, the check filter group-wise accumulates the output channels of the previous convolutional layer and takes the difference. As the generated checksum is a 2D matrix rather than a single value for the convolutional layers, we calculate the maximum value and use it for the threshold calculations and online error checking. The outputs of the check neurons and check filters are forwarded to DNN outputs without being processed with the activation functions. The modified DNN produces a single check value for each layer together with the predictions, and we compare the check values with the thresholds to determine if an error has occurred.

VI. Simulating Hardware-Level Faults on the DNN Graph

We design a comprehensive simulation tool to measure the effect of hardware-level faults on the DNN computation graph. Our simulation tools afford us the ability to inject bit errors into both activation values and weights during DNN execution. A bit error model is commonly employed to model the transient errors caused by SEUs (single event upsets) and also useful for simulating timing errors in sequential circuits. Our simulator injects activation errors via dedicated error injection layers embedded into the target network. Error injection layers produce error patterns dynamically for each prediction. We preprocess the weights to inject errors before runtime. The error-injected weights may affect multiple predictions until the weight values are refreshed by the correct ones. The simulator allows control of the error rate and the data width in the fixed-point format. We utilize our simulation framework to measure the DNN accuracy at different error rates and evaluate the performance of various error detection methods through exhaustive error injection experiments.

VII. Experimental Methods

We utilize a DNN model similar to AlexNet [6] to implement error detection methods and perform error injection experiments. Our target network model includes five convolutional and three fully-connected layers with the network being trained with the SGD (stochastic gradient descent) [12] optimizer (learning rate=10^{-2}, decay=10^{-6}, momentum=0.9, with

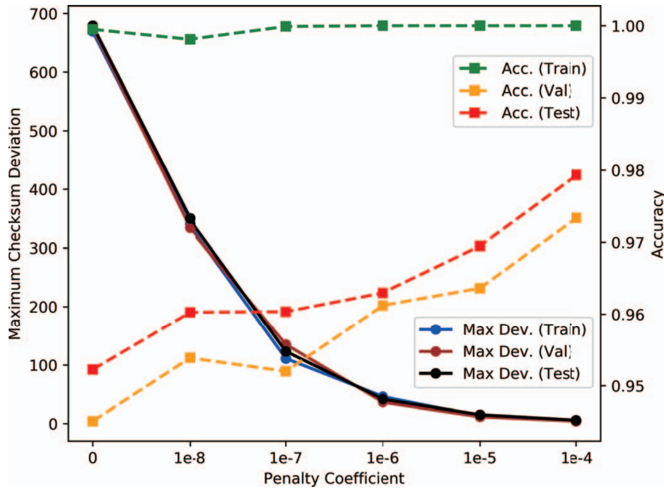

Fig. 2. Penalty coefficient vs. maximum checksum deviation and accuracy

Nesterov momentum) on the German Traffic Sign Recognition Benchmark Dataset (GTSRB) [13]. We up-sample GTSRB images, and adjust the output layer of the AlexNet network for GTSRB classification. We develop the target network, simulation tools, and error detection methods in Keras [14] and TensorFlow [15]. We use an NVIDIA GTX1060 GPU for the experiments and Intel i5-8600K CPU for additional performance measurements.

VIII. Experimental Results & Discussion

We start by training the target model with a variety of penalty coefficients (λ) and observing the maximum difference between the layer output partitions. Figure 2 demonstrates that the penalty term is highly effective in balancing the output partitions through the entire network. The observed maximum deviation decreases more than two orders of magnitude and delivers us sufficient resolution to detect even the small imbalances caused by errors.

We observe during network training another remarkable phenomenon; Figure 2 indicates that network accuracy tends to improve in line with an increase in the penalty coefficient instead of the penalty term limiting the learning efficiency. A 3% accuracy increase in both validation and test data compared to the base case without the penalty term can be observed. As we will explore shortly, the balance checksum acts as a network regularizer and helps to improve network generalization by ameliorating the overfitting problem.

We compare the accuracy increase of our methods to the conventional regularization methods used in practice. We train our model with L1/L2 weight and L1/L2 activation regularization methods and report the highest accuracy that we could obtain through a parameter sweep. We compare these results with the accuracy of the network under a balanced partitions constraint with $\lambda = 10^{-4}$, which delivers the best accuracy that we could obtain for our network. Figure 3 outlines the best accuracy values for training, validation, and test datasets. While almost all regularization methods have a positive impact on network accuracy, training the network with balanced output partitions delivers significantly higher accuracy than the standard regularization methods. If the layer outputs are considered as vectors, the introduced balance penalty term constrains the span of the generated

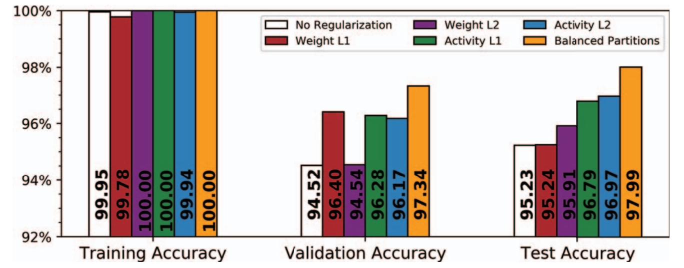

Fig. 3. The comparison of regularization methods

output vectors by forcing them to be orthogonal to the vector $V_l = [1, 1, 1, ..., -1, -1, -1]$ that has a length of s_l and populated by a sequence of $(\lfloor S_l/2 \rfloor)$ 1's and $(s_l - \lfloor S_l/2 \rfloor)$ -1's. As this constraint reduces the number of distinct features that can be extracted by one, we can expect a degradation in accuracy at a first glance.

To address this concern, we first extract the intermediate activations after non-linear functions at each layer for a randomly chosen set of 6400 test examples, then apply PCA [16] (principal component analysis) to find the number of dimensions that account for 99% variance of the activation values at each layer output. The results in Figure 4 indicate that most layers are under-utilized as they extract much fewer features than the layer size (for instance, the first convolutional layer with 96 filters only extracts 11 distinct features). As the output data has much fewer useful features than the potential number of features that the layer can represent, reducing the maximum number of features that can be extracted by one proves to have no negative impact on accuracy.[1] On the contrary, the approach we propose auspiciously improves accuracy by reducing overfitting due to the following reasons. First, it restricts the activation range of the computation units and prevents the units from generating large activation values similar to Dropout [17] or activation regularizers since larger activation values will make it harder to balance the layer partitions. Second, correlating the layer outputs reduces model complexity and makes it harder to overfit to the training data.

We assess the error and error-caused misprediction detection capabilities of our approach in Figure 5. We utilize 4 different evaluation metrics: error detection precision, error detection recall, (error-caused) misprediction precision and (error-caused) misprediction recall. Precision and recall, commonly used measures of detector effectiveness, capture

[1]The last output layer constitutes an exception as it is fully-utilized; we employ an extra neuron specifically for balancing the outputs and omit the additional neuron's output in the network decisions.

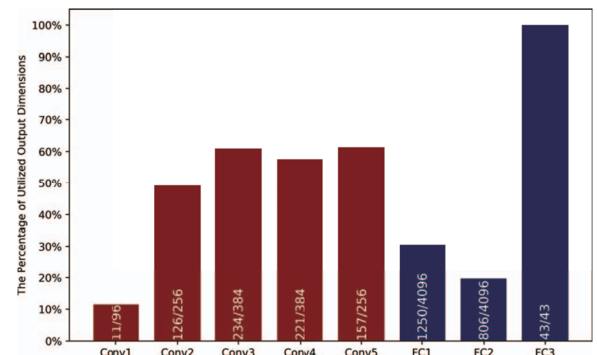

Fig. 4. The percentage of utilized output dimensions at each layer

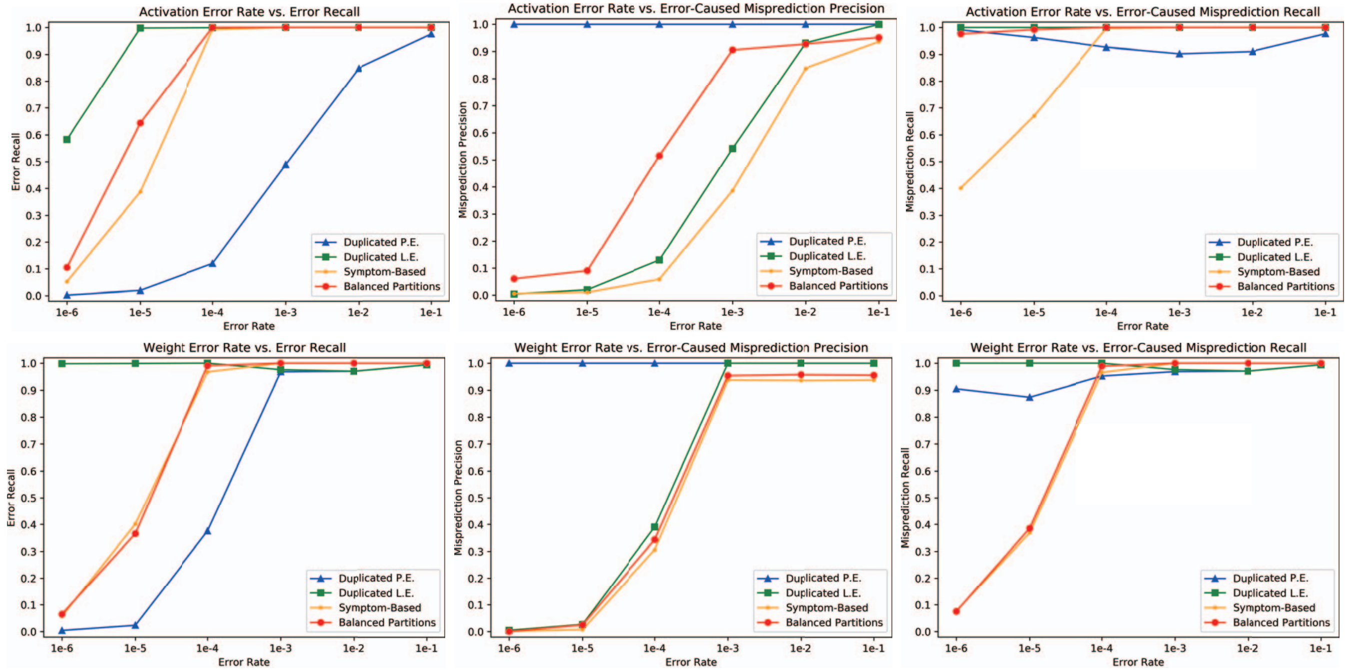

Fig. 5. Error recall, error-caused misprediction precision and error-caused misprediction recall for activation and weight errors

the accuracy of the detector among the positively identified examples and the detector coverage on the actual positive examples, respectively. For instance, while error precision indicates the percentile of the cases the detector is correct among the identified error cases, error recall indicates the percentile of the error cases being detected. A mathematical definition of precision and recall is provided in Equation (6). TP, FP, and FN denote True Positive, False Positive, and False Negative examples, respectively:

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN} \tag{6}$$

We implement three different error detection methods to compare with our approach. We design two duplicated models and employ the replicated DNN graph to check the consistency of the computations. The first method, *Duplicated P.E.* (prediction equality), checks if both replicated networks predict the same class (e.g., same traffic sign) and it reports an error if the predictions differ. The second model, *Duplicated L.E.* (likelihood vector equality), checks the probability vector produced by the last Softmax layer and indicates an error if they are not equal. We also implement a *symptom-based error detector (SED)* outlined in [18] and compare it with our approach. The symptom-based detector profiles the range of the typical activation values, multiplies the range with a margin (e.g., 1.1 as reported), and detects the errors at runtime if an error causes the activation values to lie outside of these thresholds. Similarly, we experimentally observe that multiplying the differences with a margin (e.g., 1.4) before setting as error thresholds prevents the false alarm cases.

We inject bit errors into both weights and activations separately and report the indicated metrics for both activation and weight errors at a variety of error rates to comprehensively assess the delivered safety of the presented error detection methods. We use 16-bit data types for error modeling where 1 bit is the sign, and 11 and 8 bits are allocated to fraction bits in weight and activation values, respectively. We do not plot error precision rates as all error detection methods achieve perfect error precision (1.0). These methods never cause false alarms if no error is present.

The following points in Figure 5 merit further attention. First, our approach delivers consistently higher performance than a symptom-based detector in all metrics for all activation error rates. We observe a more significant advantage in low-error rate regimes. Our method almost always outperforms the symptom-based detector in weight error metrics as well. Duplicated P.E. always delivers perfect error-caused misprediction precision rates since no error is reported until at least one network experiences a misprediction; however, Duplicated P.E. has limited recall rates for both weight and activation errors. Duplicated L.E. detects almost all error cases due to its strict error detection criteria but consequently results in lower precision rates for error-caused mispredictions. Although both Duplicated P.E. and Duplicated L.E. have their advantages, the overhead of duplication restricts in practice its utilization as a safety solution for resource-intensive applications. Finally, we observe that our method is sensitive to the input distribution, as it also detects particular input errors. We aim to measure the system performance under different input transformations and further investigate its applicability to the adversarial machine learning domain [19] in future work.

We compare the memory footprint and performance overhead of the presented methods under fixed hardware architectures in Table I to provide a comprehensive overview. We report the overhead for Duplicated L.E. and Duplicated P.E. under a single heading as duplication dominates their overall overhead. While duplicated models require doubling of the network parameters, the symptom-based detector and

TABLE I
MEMORY AND PERFORMANCE OVERHEAD

	Duplicated	SED	Balanced Part.
Parameters	100.00%	0.00%	0.01%
Perf. (CPU)	100.00%	1.08%	1.00%
Perf. (GPU)	33.32%	2.95%	2.39%

our approach require almost zero memory footprint. SED achieves 93x smaller performance overhead compared to the duplicated models while the overhead of our approach can be 100x lower than the duplicated methods. Even on a GPU, appreciable overhead reductions can be observed by noting that SED is 11x and our method 14x smaller than that of duplication. The available hardware resources ameliorate the overhead of the duplicated models on the GPU, but such opportunities are rarely available on resource-limited edge devices. As the experiments are performed on fixed hardware platforms, we do not report hardware overheads, yet one could expect duplication methods to neutralize the performance overhead at the cost of duplicated hardware resources. The small performance overhead of our method could be further reduced with much smaller overheads than duplication.

IX. Related Work

The effect of hardware-level faults on the DNN behavior has started receiving significant attention in the recent literature. Reagen et al. [20] and Neggaz et al. [21] present tools and methods to simulate hardware-level faults as bit errors on the DNN computation graph. Li et al. [18] present a comprehensive analysis of the error behavior in DNNs as well as practical methods to reduce the FIT (failure in time) rate in DNN accelerators. A symptom-based error detector for DNNs is presented in [18]. Ozen et al. [11] utilize linear checksums to detect the errors in DNN layers. Schorn et al. [22] employ a DNN-based on-line error detection method for deep neural networks. The same authors [23] present a safety aware design by measuring the importance of the neurons and mapping the critical computations and data into hardened hardware. Choi et al. [24] employ a similar approach to assign critical neurons into more robust MAC units to alleviate the effect of timing errors. Thundervolt [25] utilizes Razor flip-flops [26] and recomputes the erroneous value to resolve the timing errors. Balestriero et al. [27] employ penalty terms in the cost function to enforce orthogonality on the computation unit outputs, thus improving the diversity of the extracted features at each layer. Custom training methods are also utilized to detect and provide resilience against adversarial examples [19]. Although the utilization of balance checksums for adversarial example detection is an intriguing idea for future work, the focus of this work is limited to the identification of safety problems.

X. Conclusion

Deep neural networks play an integral role in various safety-critical systems, including ADAS (Advanced Driver-Assistance Systems) and autonomous driving. The impact of hardware-level faults consequently raises questions about the safety of DNN computations. We present an algorithmic fault-tolerance method for deep neural networks by modifying the training process with an additional penalty term. The introduced invariant helps us attain detection rates commensurate with state-of-the-art methods at only a fraction of their cost. Our approach additionally helps to improve DNN accuracy by acting as a regularizer during training. As traditional methods fall short in a practical sense due to their high cost, we deliver an alternative approach for fault-tolerance, thus engendering safe and affordable DNN solutions for the consumer market through algorithm-level modifications.

References

[1] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning affordance for direct perception in autonomous driving," in *IEEE International Conference on Computer Vision*, pp. 2722–2730, 2015.

[2] "ISO 26262-1:2018 road vehicles – functional safety." https://www.iso.org/standard/68383.html, 2018.

[3] H. Shaheen, G. Boschi, G. Harutyunyan, and Y. Zorian, "Advanced ECC solution for automotive SoCs," in *23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 71–73, IEEE, 2017.

[4] M. Baleani *et al.*, "Fault-tolerant platforms for automotive safety-critical applications," in *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 170–177, ACM, 2003.

[5] I. Lee *et al.*, "Survey of error and fault detection mechanisms," tech. rep., University of Texas at Austin, 2012.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

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

[8] C. Szegedy *et al.*, "Going deeper with convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[10] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[11] E. Ozen and A. Orailoglu, "Sanity-Check: Boosting the reliability of safety-critical deep neural network applications," in *Proceedings of IEEE Asian Test Symposium (ATS)*, 2019.

[12] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *COMPSTAT'2010*, pp. 177–186, Springer, 2010.

[13] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, vol. 32, pp. 323–332, 2012.

[14] F. Chollet *et al.*, "Keras." https://keras.io, 2015.

[15] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation*, pp. 265–283, 2016.

[16] J. Shlens, "A tutorial on principal component analysis," *arXiv preprint arXiv:1404.1100*, 2014.

[17] N. Srivastava *et al.*, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[18] G. Li *et al.*, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 8, ACM, 2017.

[19] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE Transactions on Neural Networks and Learning Systems*, 2019.

[20] B. Reagen *et al.*, "Ares: A framework for quantifying the resilience of deep neural networks," in *55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, 2018.

[21] M. A. Neggaz, I. Alouani, P. R. Lorenzo, and S. Niar, "A reliability study on CNNs for critical embedded systems," in *36th International Conference on Computer Design (ICCD)*, pp. 476–479, IEEE, 2018.

[22] C. Schorn, A. Guntoro, and G. Ascheid, "Efficient on-line error detection and mitigation for deep neural network accelerators," in *International Conference on Computer Safety, Reliability, and Security*, pp. 205–219, Springer, 2018.

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

[24] W. Choi, D. Shin, J. Park, and S. Ghosh, "Sensitivity based error resilient techniques for energy efficient deep neural network accelerators," in *56th Annual Design Automation Conference*, ACM, 2019.

[25] 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 *55th Annual Design Automation Conference*, ACM, 2018.

[26] D. Ernst *et al.*, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[27] R. Balestriero *et al.*, "A spline theory of deep networks," in *International Conference on Machine Learning*, pp. 383–392, 2018.