Maintaining Sanity: Algorithm-based Comprehensive Fault Tolerance for CNNs

Jinhyo Jung[1], Hwisoo So[2], Woobin Ko[1], Sumedh Joshi[2], Yebon Kim[1],
Yohan Ko[1], Aviral Shrivastava[2], Kyoungwoo Lee[1]
[1]Yonsei University, [2]Arizona State University
jinhyo.jung@yonsei.ac.kr

ABSTRACT

As the deployment of neural networks in safety-critical applications proliferates, it becomes imperative that they exhibit consistent and dependable performance amidst hardware malfunctions. Several protection schemes have been proposed to protect neural networks, but they suffer from huge overheads or insufficient fault coverage. This paper presents Maintaining Sanity, a comprehensive and efficient protection technique for CNNs. Maintaining Sanity extends the state-of-the-art algorithm-based fault tolerance for CNN, utilizing hamming codes and checkpointing to correct over 99.6% of critical faults with about 72% runtime overhead and minimal memory overhead compared to traditional triple modular redundancy (TMR) techniques.

KEYWORDS

Reliability, Soft Errors, Transient Faults, Fault Injection, ABFT, CNN, Reliable Machine Learning

ACM Reference Format:
Jinhyo Jung, Hwisoo So, Woobin Ko, Sumedh Joshi, Yebon Kim, Yohan Ko, Aviral Shrivastava, Kyoungwoo Lee. 2024. Maintaining Sanity: Algorithm-based Comprehensive Fault Tolerance for CNNs. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3657355

1 INTRODUCTION

The surge of CNNs has led to their adoption in safety-critical applications such as autonomous driving, climate analysis, and disease diagnosis [24, 31]. In such applications, reliability, which ensures correct functionalities despite hardware faults, is the primary challenge. Soft errors, also known as transient faults, are unintended bit flips in the hardware induced by external sources such as cosmic rays, thermal neutrons, or alpha particles [1, 2]. These errors pose a significant challenge as they can lead to application malfunction even when the software and hardware components are flawless, making them a central focus in the design of safety-critical systems.

Neural networks are inherently robust against minor faults due to the distributed and parallel structure of the networks and intrinsic redundancy from over-provisioning [18, 28]. However, the black-box nature of the neural networks does not help to understand the degree of tolerance or vulnerability of individual components to soft errors. Single-bit fault can alter the classification result, for instance, recognizing a truck as a bird [21], which can cause catastrophic results in autonomous driving. Another study on the reliability of neural networks found that unprotected hardware that deployed neural networks failed to meet the strict reliability standards of failure-in-time (FIT) rates [12]. This problem is further aggravated by increased soft error rates due to the drastic shrinking and voltage scaling of transistors [6, 10].

Several protection schemes have been presented to protect neural networks against soft errors, but they suffer from at least one of three significant limitations. Firstly, existing methods for full fault coverage incur severe hardware costs or runtime overheads. Modular redundancy solutions applied to neural networks can detect or correct the effects of soft errors. Still, the high overheads make it hard to meet the strict constraints for safety-critical systems. Secondly, existing schemes aimed at improving efficiency may not provide sufficient fault coverage. These methods may protect parts of neural networks efficiently but may leave the rest of the network exposed to faults. Lastly, existing techniques may require additional components, training, or restrictions. For example, machine learning-based fault mitigation solutions require an additional network or training phase, while range-restriction-based solutions fail if the network is highly quantized.

This paper presents Maintaining Sanity, a comprehensive and efficient fault correction technique for convolutional neural networks inspired by algorithm-based fault tolerance (ABFT). Figure 1 depicts the workflow of Maintaining Sanity. Maintaining Sanity extends the state-of-the-art ABFT for CNN based on the hamming code [9] to provide redundancy for model parameters, correcting faults in weights, biases, and outputs of convolutional and fully connected layers. In addition, Maintaining Sanity duplicates only the original input to recover from any faults in the layer inputs through a checkpointing algorithm. The outputs of convolutional and fully connected layers are used as checkpoints for backward recovery since the extended checksum algorithm corrects any faults that may have occurred. Our fault injection experiments on four popular CNNs show that Maintaining Sanity can correct over 99.6% of faults that would corrupt the final classification results. Comparisons with other techniques show that Maintaining Sanity maintains near-perfect fault coverage with only around 72% the performance cost on the geometric mean of triple modular redundancy methods. It also does so with minimal additional parameters, meaning Maintaining Sanity can be applied even to environments with tight memory constraints.

In summary, this paper makes the following contributions:

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3657355

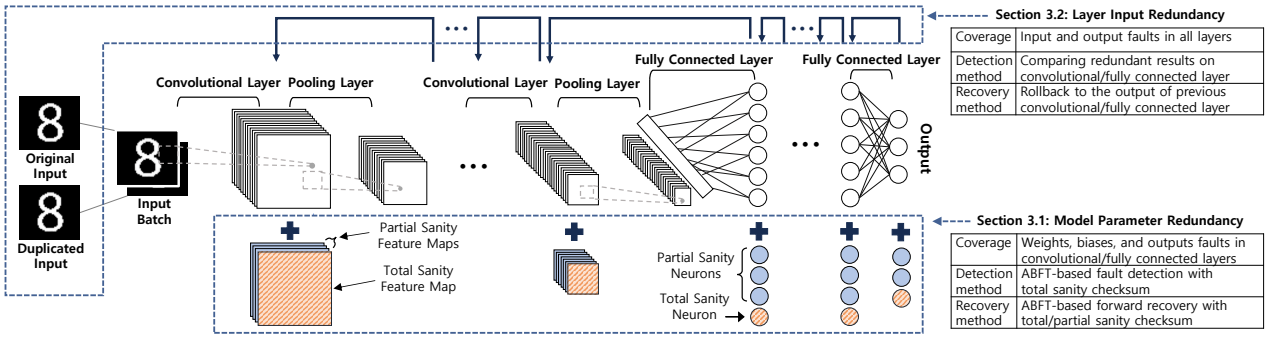

Figure 1: Maintaining Sanity adds extra sanity outputs to correct any fault in model parameters and exploits input redundancy to recover from any fault in the inputs of layers.

- **Comprehensive reliability:** Maintaining Sanity can detect and correct any faults in the entire CNN, regardless of the timing and location of the fault.
- **Efficacy:** Maintaining Sanity incurs about 135% additional overhead, significantly more efficient than traditional triple modular redundancy methods with similar fault coverage.
- **Ease of applicability:** Maintaining Sanity does not have any prerequisites that should be met for the technique to be applied. In addition, its low overheads make it suitable even for resource-constrained environments.

2 RELATED WORKS

2.1 Neural Network Reliability

As the use of CNNs expands to include safety-critical applications, improving the reliability of neural networks becomes increasingly important. Therefore, various techniques for identifying or rectifying erroneous values have been introduced to protect neural networks. A straightforward approach to protecting neural networks is applying dual modular redundancy (DMR) or triple modular redundancy (TMR), a well-known strategy for proving integrity about single component failure. These methods duplicate or triplicate a module at either hardware or software level so that the redundant modules perform the identical procedure. With proper comparison between the modules, a fault can be detected (DMR) or even corrected (TMR). While safety-critical applications in traditional computing systems widely adopted DMR and TMR strategies, the spatial and temporal overhead required for the replication makes it difficult to apply direct modular redundancies to neural networks, which already require extensive computing resources and are highly sensitive to the runtime overheads. [22].

Another approach exploits range-restriction-based strategies to apply lightweight fault correction for deep neural networks (DNNs). Theses studies assume that critical faults that eventually alter the final classification results cause notably large value deviations. Therefore, range-restriction-based solutions restrict the range of activation functions with a global [5] or neuron-wise thresholds [8] to mitigate the amount of deviations a fault can cause. However, since state-of-the-art quantization for the data types, such as INT8, already utilizes most of the range [24], range-restriction-based solutions cannot provide sufficient fault coverage on networks with aggressive quantization.

Other studies attempt to counter the effects of errors using additional neural network training. Cavagnero et al. [4] argue that exposing the target neural network to faults during the training phase can enhance the inherent fault tolerance of the neural network. Another strategy [30] trains a small neural network tailored to identify faults in the feature maps produced by the convolutional layers of the original target network. While these solutions can provide efficient protection, they rely on the black-box nature of neural networks to detect faults, lacking interpretability or coverage. They also have the disadvantage of requiring additional training data with sufficient fault injection trials.

2.2 Sanity-Check

The work most closely related to this paper is – Sanity-Check [25], which proposed a mechanism to detect errors in DNN inference, based on the principles of ABFT [13]. Sanity-Check uses the concept of spatial checksums to protect the weights, biases, and outputs of a fully connected or convolutional layer. For fully connected layers, Sanity-Check adds two redundant neurons: sanity and check neurons. The weights and biases of the sanity neuron are set so that the value of the neuron is equal to the negative sum of all other output neurons, regardless of the input values, in the absence of faults. The check neuron calculates the spatial checksum by adding the value of all output neurons, including the original and sanity neurons. A non-zero spatial checksum would indicate the presence of a fault in the layer's weight, bias, or output.[1] Sanity-Check applies a similar process to convolutional layers, adding an extra sanity filter and bias to detect any fault in the layer parameters.

The spatial checksum alone is inadequate since it cannot detect faults in the layer inputs. Therefore, Sanity-Check introduces temporal checksums, adding sanity inputs within the input batches of convolutional and fully connected layers. Sanity inputs are computed as the negative sum of all other inputs so that the spatial checksum, the sum of all outputs in the layer, equals the product of the batch size and the bias vector/matrix of the corresponding layer. Any deviation from this expected sum allows Sanity-Check to identify the presence of faults in the layer inputs.

While Sanity-Check effectively protects convolutional and fully connected layers, it does not extend its coverage to other layers, such as activation functions and pooling layers. Sanity-Check also necessitates batch inference to calculate the temporal checksums, introducing latency and increased memory pressure, making it suboptimal for time-sensitive applications or memory-limited computing environments [15, 26]. In addition, faults during the sanity input calculation process can propagate to affect both the original and sanity input, making the fault invisible to Sanity-Check.

[1] Due to rounding errors in computation, Sanity-Check compares the checksum against a threshold rather than 0.

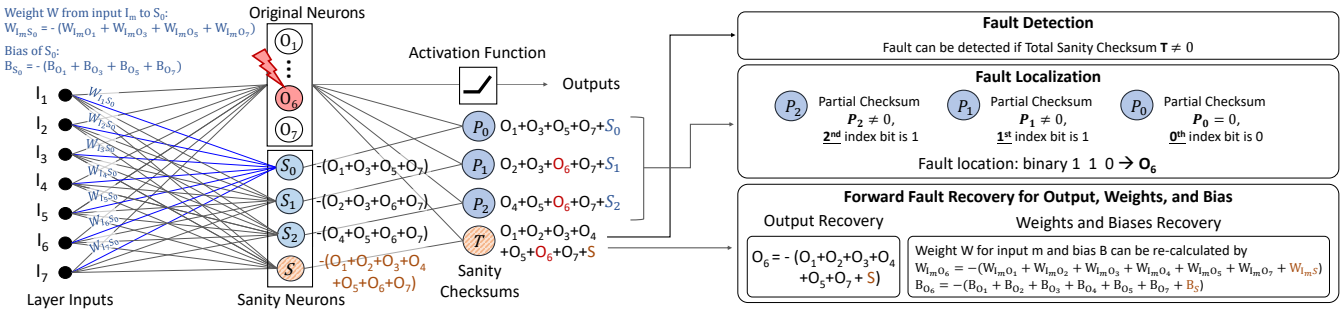

Figure 2: Model parameter redundancy of Maintaining Sanity for fully connected layers can detect, locate, and correct the fault on weights, bias, and the outputs of fully connected layers. This figure shows the process for a fault corrupting output O_6.

3 OUR APPROACH: MAINTAINING SANITY

We present Maintaining Sanity, a comprehensive fault correction technique for convolutional neural networks inspired by ABFT [13]. We enhance the ideas presented by the state-of-the-art ABFT solution, Sanity-Check [25], to correct faults by incorporating the concept of hamming codes [9]. We also refine the idea to extend the fault coverage to the entire network, eliminate any single points of failure, and generalize the technique to apply to any CNNs.

We divide the data to be protected into two categories: model parameters and layer inputs. Model parameters contain weights and biases of convolutional and fully connected layers. These values can be read before processing any input values, allowing the generation of checksum algorithms. We assume that redundant copies of model parameters are unavailable due to the high spatial overhead. Layer inputs contain the intermediate inputs to each layer, including the original input to the neural network. These values cannot be seen before inputting the value to the network, and therefore, sanity inputs cannot be calculated beforehand. The differing features of the two categories require us to devise distinctive techniques to provide appropriate redundancy for each category.

3.1 Model Parameter Redundancy

We introduce a fault correction approach for model parameters, drawing inspiration from algorithm-based fault tolerance methods. This approach incorporates an error detection mechanism, Sanity-Check [25]. Sanity-Check employs spatial checksums to identify faults in the weights, biases, and outputs of convolutional and fully connected layers within neural networks. The core principle involves integrating additional operations within a layer to generate a sanity output. This output is designed such that the aggregate of the original outputs and the sanity output equals zero, irrespective of the input values. However, it is essential to note that the Sanity-Check is limited to fault detection and does not inherently offer fault correction capabilities. Therefore, significant adaptations are required to extend its functionality to include fault correction.

We first explain how our technique is applied in fully connected layers. Assume a fully connected layer with n outputs, namely $O_1 \sim O_n$. Protecting this layer with the spatial checksum algorithm from Sanity-Check gives an additional output O_S, such that $\sum_{x=1}^{n} O_x + O_S = 0$. If any of the layers' weights, biases, or outputs are corrupted, the equation will not hold, and the fault can be detected. We define this crucial extra output O_S as the total sanity neuron. Interestingly, the total sanity neuron O_S can also be used to recover from the fault if we can identify the faulty output. Without loss of generality, assume that a fault corrupts the value of O_k. By modifying the equation above, we can obtain the original value of O_k by calculating $\sum_{x=1,x\neq k}^{n} O_x + O_S = -O_k$. To locate the faulty output, Maintaining Sanity adds extra sanity neurons, defined as partial sanity neurons, in a manner similar to forming hamming codes. Like hamming codes, the number of partial sanity neurons required to protect the layer is equal to the number of bits required to represent the number of output neurons in the layer; therefore, the number of additional neurons for a fully connected layer with n output neurons is $\lceil \log_2(n+1) \rceil$ (partial sanity neuron) + 1 (total sanity neuron). The difference is that the sanity neurons calculate the sum of a subset of outputs instead of parities.

Figure 2 illustrates how Maintaining Sanity adds sanity neurons to protect a fully connected layer with seven output neurons O_1 to O_7. The sanity neurons consist of $\lceil \log_2(7+1) \rceil = 3$ partial sanity neurons S_0 to S_2 and one total sanity neuron S. The total sanity neuron S is formed so that its value is the negative sum of all original output neurons. This can be done by adjusting each weight from the input to the total sanity neuron to be the additive inverse of the sum of the weight from the input to the original output neurons and letting the bias of the total sanity neuron be equal to the negative sum of all biases of original output neurons. A similar process is used to form partial sanity neurons, except that a partial sanity neuron S_i is formed only with a subset of output neurons whose i^{th} bit of its index is 1. For example, the partial sanity neuron S_0 is formed with the output neurons that have 1 as the 0^{th} bit in their index, namely O_1 (binary 001), O_3 (binary 011), O_5 (binary 101), and O_7 (binary 111).

In the absence of a fault, the sum of O_1, O_3, O_5, O_7, and S_0 should be 0. We define the partial sanity checksum P_0 as the sum of the partial sanity neuron S_0 and the partial sum of the outputs corresponding to S_0 (O_1, O_3, O_5, and O_7). More generally, we define partial sanity checksum P_k as the sum of the sanity neuron S_k and its corresponding output neurons, which have 1 as the k^{th} bit of its index. The total sanity checksum is defined as the sum of the total sanity neuron and all the original outputs. Using the original output neurons and the sanity checksums, we can form codes for each error case, where 0 would represent no fault in the location, and 1 would represent a deviation from the expected output. The minimum hamming distance between these codes is 3, meaning that Maintaining Sanity can locate and correct any fault in the model parameters.

The forward recovery process of the model parameter redundancy, including fault detection, localization, and correction, is illustrated on the right side of Figure 2. To reduce the cost required for fault detection, Maintaining Sanity only checks whether the

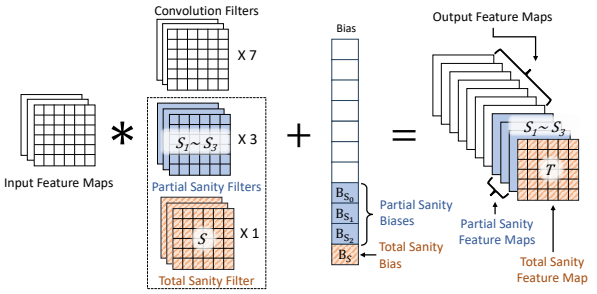

Figure 3: Maintaining Sanity's model parameter redundancy can also be applied to protect convolutional layers.

value of the total sanity checksum T equals zero [2]. In the case of a detected fault, Maintaining Sanity checks the value of the partial sanity checksums, $P0 \sim P2$, to locate the fault. In this case, where output O_6 is corrupted, partial sanity checksums P_2 and P_1 are nonzero while P_0 is 0. The error code for this case would be 110, meaning that Maintaining Sanity succeeded in locating the error in O_6. Finally, Maintaining Sanity can recover the output O_6 and the corresponding weights and bias using the Sanity weights, bias, and checksum. If no partial sanity checksum reports a fault, it means the fault affected only the total sanity checksum. In this case, we recalculate the total sanity neuron's weights and bias.

The same idea can be applied to protect weights, biases, and the outputs of convolutional layers, as shown in Figure 3. Maintaining Sanity adds sanity filters and biases to the network such that the total or partial sum of the original feature maps and the corresponding total or partial sanity feature map is 0. Maintaining Sanity utilizes the total sanity checksum to detect the presence of a fault, finds the fault location using the partial sanity checksums, and corrects the weights, biases, and outputs corresponding to the fault location. Similar to the protection for the fully connected layers, Maintaining Sanity adds $\lceil \log_2(N+1) \rceil$ extra sanity filters and biases as partial sanity feature maps and adds one additional filter and bias for the total sanity feature map. Maintaining Sanity can detect, locate, and correct any fault that corrupts a weight, bias, or output value in a fully connected or convolutional layer with the total and partial sanity checksums.

3.2 Layer Input Redundancy

We have developed an innovative approach to safeguard the input values for each layer of the target network. This was necessitated by the unique nature of these intermediate input values, distinct from model parameters and thus not amenable to the protection method described in Section 3.1. The ABFT-based method would require checksums across inputs and cannot protect the network in single-input environments. Augmenting the input dimensions to add redundancy is also impractical, as it would require changes to the architecture of the network. We, therefore, adopt dual modular redundancy (DMR) for the layer inputs and checkpoint and rollback strategy to recover from input faults. With careful implementation, we can reduce the overheads required for this replication. For example, we can take advantage of the optimization of commonly used matrix operations in CNNs to lower the computation costs, allowing us to protect all network layers, including activation and pooling layers, with tolerable runtime overheads.

[2]Due to precision errors, we check whether the checksum is smaller than a threshold instead of checking if it is 0 in our implementation with 32-bit floating point data type.

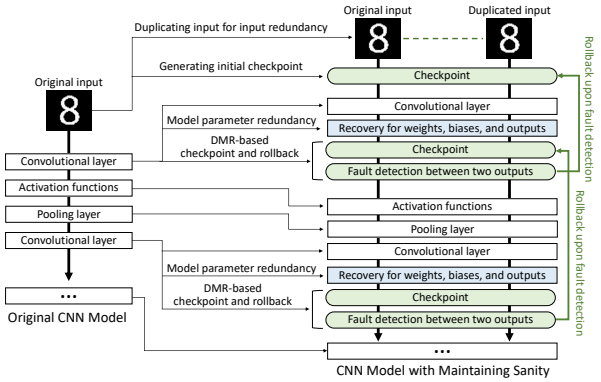

Figure 4: Layer input redundancy of Maintaining Sanity duplicates the original input and applies checkpointing to detect intermediate layer input faults.

Figure 4 illustrates an example of Maintaining Sanity protection for a CNN with model parameter redundancy (discussed in Section 3.1) and layer input redundancy. First off, Maintaining Sanity duplicates the first input to the network. Secondly, after processing a convolutional or fully connected layer and the corresponding forward recovery for weights, biases, and outputs, Maintaining Sanity generates a checkpoint. Since weights and biases can be recovered with the checksums, Maintaining Sanity needs to store only the intermediate outputs for the checkpoint. After generating the checkpoint, Maintaining Sanity compares the redundant outputs to detect a fault. If a fault is detected, Maintaining Sanity roll-backs the process to the checkpoint of the previous convolutional or fully connected layer.

The layer input redundancy of Maintaining Sanity has several noteworthy implementation details. First off, instead of duplicating the input at each layer, Maintaining Sanity replicates only the original input, and utilizes the redundant outputs of a layer as the redundant inputs to the next layer. This not only saves resources to generate redundant input for every layer but also eliminates potential undetectable faults; during the input replication process for each layer, a fault in the original copy may propagate to both copies, allowing the fault to remain undetected. Secondly, since Maintaining Sanity keeps the input redundancy not only for the convolutional and fully connected layers but also for other layers such as activation functions and pooling layers, Maintaining Sanity can cover the faults on all layers. Thirdly, comparing two outputs of the convolutional or fully connected layer can be optimized with the total sanity checksum of the model parameter redundancy in Section 3.1. The forward recovery process in Figure 4 generates the total sanity checksum per each input, which forms a sum of all outputs and the total sanity neuron or feature map. Therefore, instead of comparing the entire output tensors from redundant outputs, Maintaining Sanity can detect a fault by comparing the two total sanity checksums.

A final critical detail is to eliminate any single points of failure that can occur. A naive implementation may fail if faults occur after the checking between two copies, as illustrated in Figure 5 (a). In this example, the fault corrupted the input for the checkpoint before the computation of the next layer. The network then detects the fault and re-executes the layer, only to detect the same fault again. To avoid this problem, Maintaining Sanity stores the input copies before the checking algorithm, as shown in Figure 5 (b). If the copies fail to pass the fault detection process, the layer will be re-executed with the stored inputs. Since the stored inputs have

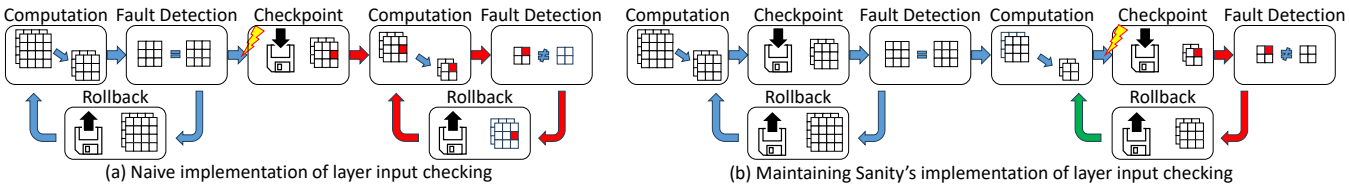

Figure 5: (a) Naive implementation of layer input redundancy may fail to recover from faults occurring after the fault detection and before the checkpoint. (b) To resolve such vulnerability, Maintaining Sanity places the checkpoint before the fault detection.

already passed the checking process in the previous layer, they are guaranteed to be error-free. This interweaving of store and check instructions ensures that all inputs to the next layer are faultless.

In summary, Maintaining Sanity performs forward recovery to correct faults in the model parameters based on ABFT. It performs backward recovery for faults found in the inputs to any layer in the network through a carefully crafted replication algorithm. Maintaining Sanity can also be applied without any preconditions for the target network: it does not require batch inputs [25], functions normally in quantized networks [5, 8], and can be applied without additional training [4, 30].

4 EXPERIMENTAL SETUP

We implemented Maintaining Sanity on AlexNet [17], LeNet5 [20], ResNet18 [11][3], and VGG16 [29], to quantify its fault coverage and runtime overheads. These networks are representative of CNNs and have been used as test benchmarks in various studies on DNN reliability [5, 16, 23, 25]. Maintaining Sanity is implemented in the PyTorch [27] environment, using the 32-bit floating point data type to represent network parameters. We conducted our experiments on an Intel Xeon 2.2 GHz CPU with the ImageNet dataset [7], using the pre-trained model parameters provided by PyTorch, achieving 56.522%, 69.758%, and 71.592% top-1 accuracy for AlexNet, ResNet18, and VGG16, respectively. For LeNet5, we trained the network on the MNIST [19] dataset, achieving 98.71% accuracy.

Baseline Protection Techniques: We implemented existing techniques alongside Maintaining Sanity as the benchmarks for our experiment. We implemented two versions of Sanity-Check. One version (Spa-SC) is implemented with only the spatial checksums since our experimental environment does not use batch inference. The other version (SC) generates the temporal checksum anyway, regarding the single input as the input batch. We also implemented two versions of triple modular redundancy. Input-level TMR (I-TMR) triplicates only the input to the network, while network-level TMR (N-TMR) triplicates both the model parameters and the input. Both versions perform majority voting once before the final output to reduce overheads. We measure the fault coverage, runtime percentage, and application costs of the benchmarks to compare them with those of Maintaining Sanity (MS).

Fault Injection Process: We injected a single fault in the baseline networks during the inference phase of an image from the test dataset. Faults that change the final classification result of the network are classified as critical faults, and we included them in the set of faults to assess the protection techniques. Note that we only injected faults in images initially classified correctly to prevent cases where the injected fault "corrects" the classification. Faults were injected randomly in model parameters and layer inputs until we obtained 1,000 critical model parameter faults and 1,000 critical layer input faults for each baseline network. To speed up the collection process, we injected faults only in the exponent bits of the 32-bit float. This fault model is in line with those used for other DNN reliability studies [3, 32, 33]. We then injected the critical faults into each protection technique and measured the percentage of detected or corrected faults to quantify the fault coverage rates.

[3]The checkpointing of Maintaining Sanity for ResNet also covers input and output of layers inside of residual blocks and parameters of batch normalization layers [14].

5 RESULTS

Figure 6 presents the critical fault coverage and the runtime percentage of the protection techniques in each baseline network. The protection techniques are listed on the X-axis: two versions of Sanity-Check (Spa-SC without temporal checksums and SC with them), input-level TMR (I-TMR), network-level TMR (N-TMR), and Maintaining Sanity (MS), and the Y-axes represent the fault coverage and runtime percentage on the left and right, respectively. The fault coverage, shown as two bars for each technique, is measured by the ratio of detected or corrected faults to the total number of critical faults injected in the model parameters and the layer inputs. While I-TMR shows perfect coverage for layer input faults, it struggles to detect model parameter faults. This is because model parameter faults can propagate to multiple inputs. SC techniques show the opposite behavior, detecting model parameter faults but suffering from layer input faults. Even the one with temporal checksums fails to detect layer input faults injected in layers other than convolutional and fully connected layers. On the other hand, Maintaining Sanity (MS) shows near-perfect fault coverage, correcting over 99.6% of both model parameter and layer input faults. This is more impressive, considering that we do not inject any benign faults. Through detailed analyses, we found that the fault cases that Maintaining Sanity (MS) fails to correct are due to precision errors. If the target network can be quantized, then Maintaining Sanity would no longer suffer from precision errors in the thresholds, achieving 100% fault recovery.

The runtime percentage of the techniques compared to the baseline networks in an error-free environment are also shown as the black lines in Figure 6. Specifically, we measured the runtime of the inference phase of the test dataset of MNIST for LeNet5 and 1,000 images from the ImageNet validation dataset for the other networks. None of the techniques triggered false alarms in this process, indicating that they have no additional performance overheads in fault-free scenarios compared to the baseline model. Maintaining Sanity showed an overall runtime overhead of about 135%, which is only about 72% of the overhead of network-level TMR, which had around 187% overheads. All the values are calculated based on the geometric mean. The other protection techniques displayed similar or lower overheads on average, but it should be noted that their efficiency comes at the cost of incomprehensive fault coverage.

Finally, Table 1 shows the spatial overheads for each technique. The columns represent the five protection techniques, and the rows represent the different baseline networks. Each cell addresses the

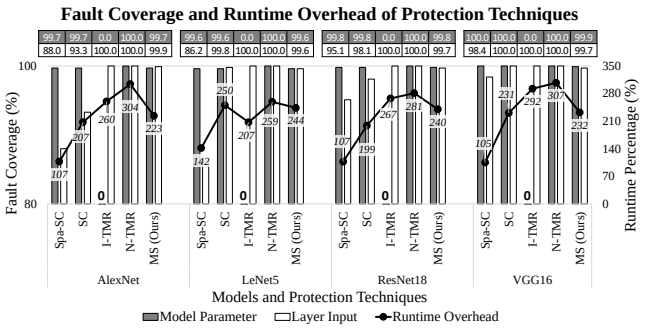

Figure 6: The fault coverage and runtime percentage of the protection techniques. Maintaining Sanity retains near-perfect fault coverage with moderate overheads.

Table 1: The Memory Costs of Protection Techniques

	Spa-SC	SC	I-TMR	N-TMR	MS (Ours)
AlexNet	0.04%	0.04%	0.0%	200.0%	0.53%
LeNet5	1.27%	1.27%	0.0%	200.0%	9.09%
ResNet	0.27%	0.27%	0.0%	200.0%	2.81%
VGG16	0.05%	0.05%	0.0%	200.0%	0.59%

overhead, which refers to the extra number of parameters each technique adds as a percentage compared to the baseline networks. Our findings show a stark contrast in the spatial overhead among these techniques. The Net-TMR technique significantly increases parameters, requiring 200% more than the baseline. This level of overhead is particularly concerning in safety-critical environments where efficiency and resource management are paramount. In contrast, our Maintaining Sanity technique is much more efficient, requiring less than 10% additional parameters for LeNet5 while maintaining high fault coverage. This overhead decreases for larger networks, making Maintaining Sanity even more efficient.

6 CONCLUSION

As neural networks become integral in safety-critical applications due to technological advancements, various methods to shield CNNs from transient faults have emerged. This paper introduces Maintaining Sanity, a strategy designed to efficiently enhance the fault resilience of CNNs through algorithm-based fault tolerance. Our extensive fault injection campaigns reveal that Maintaining Sanity boasts near-perfect critical fault coverage while also speeding up the process by around 22% compared to traditional modular redundancy methods. Future works include further speeding up Maintaining Sanity by optimizing with GPU or DNN accelerator, perfecting the fault coverage through quantization, and broadening the framework by considering options like batch inference or TMR to achieve an ideal balance between performance and dependability.

ACKNOWLEDGMENTS

This research was partially supported by National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. RS-2022-00165225), by Institute of Information & Communications Technology Planning & Evaluation (IITP) Grant funded by the Korean government (MSIT), Artificial Intelligence Graduate School Program, Yonsei University, under Grant 2020-0-01361, by funding from National Science Foundation grants CPS 1645578 and Semiconductor Research Corporation (SRC) project 3154, and by "Regional Innovation Strategy (RIS)" through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (MOE) in 2024(2022RIS-005).

REFERENCES

[1] R. Baumann. 2005. Soft errors in advanced computer systems. *Design and Test* 22, 3 (2005), 258–266.
[2] R. Baumann et al. 1995. Boron as a primary source of radiation in high density DRAMs. In *VLSI Technology Symposium*. IEEE, 81–82.
[3] A. Bosio et al. 2019. A reliability analysis of a deep neural network. In *LATE*. IEEE, 1–6.
[4] N. Cavagnero et al. 2022. Transient-Fault-Aware Design and Training to Enhance DNNs Reliability with Zero-Overhead. In *IOLTS*. 1–7.
[5] Z. Chen et al. 2021. A low-cost fault corrector for deep neural networks through range restriction. In *DSN*. IEEE, 1–13.
[6] V. Degalahal, Lin Li, V. Narayanan, M. Kandemir, and M.J. Irwin. 2005. Soft errors issues in low-power caches. *VLSI Systems* 13, 10 (Oct 2005), 1157–1166.
[7] J. Deng et al. 2009. Imagenet: A large-scale hierarchical image database. In *CVPR*. IEEE, 248–255.
[8] B. Ghavami et al. 2022. FitAct: Error resilient deep neural networks via fine-grained post-trainable activation functions. In *DATE*. IEEE, 1239–1244.
[9] R. W. Hamming. 1950. Error detecting and error correcting codes. *The Bell System Technical Journal* 29, 2 (April 1950), 147–160.
[10] M. Hashimoto, K. Kobayashi, et al. 2019. Characterizing SRAM and FF soft error rates with measurement and simulation. *Integration* 69 (2019), 161–179. https://www.sciencedirect.com/science/article/pii/S0167926018305613
[11] Kaiming He et al. 2015. Deep Residual Learning for Image Recognition.
[12] Yi He, Prasanna Balaprakash, and Yanjing Li. 2020. Fidelity: Efficient resilience analysis framework for deep learning accelerators. In *MICRO*. IEEE, 270–281.
[13] K.H. Huang and J. A. Abraham. 1984. Algorithm-based fault tolerance for matrix operations. *TOC* 100, 6 (1984), 518–528.
[14] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*. pmlr.
[15] S. Khadka et al. 2020. Optimizing memory placement using evolutionary graph reinforcement learning. *arXiv* (2020).
[16] J. Kim and J. Yang. 2019. DRIS-3: Deep neural network reliability improvement scheme in 3D die-stacked memory based on fault analysis. In *DAC*. 1–6.
[17] A. Krizhevsky et al. 2012. Imagenet classification with deep convolutional neural networks. *NeurIPS* 25 (2012).
[18] S. Lawrence et al. 1998. *What size neural network gives optimal generalization? Convergence properties of backpropagation.* Technical Report.
[19] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).
[20] Y. Lecun et al. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[21] G. Li et al. 2017. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In *SC'17*. 1–12.
[22] Z. Liu and X. Yang. 2022. An efficient structure to improve the reliability of deep neural networks on ARMs. *Microelectron. Reliab.* 136 (2022), 114729.
[23] A. Mahmoud et al. 2020. HarDNN: Feature map vulnerability evaluation in cnns. *arXiv* (2020).
[24] A. Mahmoud et al. 2021. Optimizing Selective Protection for CNN Resilience.. In *ISSRE*. 127–138.
[25] E. Ozen and A. Orailoglu. 2019. Sanity-check: Boosting the reliability of safety-critical deep neural network applications. In *ATS*. IEEE, 7–75.
[26] J. Park et al. 2018. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv* (2018).
[27] A. Paszke et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *NeurIPS* 32 (2019).
[28] A. Ruospo and E. Sanchez. 2021. On the reliability assessment of artificial neural networks running on ai-oriented mpsocs. *Applied Sciences* 11, 14 (2021), 6455.
[29] Karen S. and Andrew Z. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition.
[30] C. Schorn et al. 2018. Efficient On-Line Error Detection and Mitigation for Deep Neural Network Accelerators. In *Eds.* Springer International Publishing, 205–219.
[31] S. S. Yadav and S. M. Jadhav. 2019. Deep convolutional neural network based medical image classification for disease diagnosis. *Journal of Big data* 6, 1 (2019), 1–18.
[32] Z. Yan et al. 2020. When single event upset meets deep neural networks: Observations, explorations, and remedies. In *ASP-DAC*. IEEE, 163–168.
[33] Y. Zhang et al. 2022. Estimating vulnerability of all model parameters in dnn with a small number of fault injections. In *DATE*. IEEE, 60–63.

Received 20 November 2023