Comparing Application-Level Hardening Techniques for Neural Networks on GPUs

electronics

MDPI

Article

Comparing Application-Level Hardening Techniques for Neural Networks on GPUs

Giuseppe Esposito *,†, Juan-David Guerrero-Balaguera †, Josie E. Rodriguez Condia † and Matteo Sonza Reorda †

Department of Control and Computer Engineering, Politecnico di Torino, 10129 Turin, Italy; juan.guerrero@polito.it (J.-D.G.-B.); josie.rodriguez@polito.it (J.E.R.C.); matteo.sonzareorda@polito.it (M.S.R.)
* Correspondence: giuseppe.esposito@polito.it

Abstract: Neural networks (NNs) are essential in advancing modern safety-critical systems. Lightweight NN architectures are deployed on resource-constrained devices using hardware accelerators like Graphics Processing Units (GPUs) for fast responses. However, the latest semiconductor technologies may be affected by physical faults that can jeopardize the NN computations, making fault mitigation crucial for safety-critical domains. The recent studies propose software-based Hardening Techniques (HTs) to address these faults. However, the proposed fault countermeasures are evaluated through different hardware-agnostic error models neglecting the effort required for their implementation and different test benches. Comparing application-level HTs across different studies is challenging, leaving it unclear (i) their effectiveness against hardware-aware error models on any NN and (ii) which HTs provide the best trade-off between reliability enhancement and implementation cost. In this study, application-level HTs are evaluated homogeneously and independently by performing a study on the feasibility of implementation and a reliability assessment under two hardware-aware error models: (i) weight single bit-flips and (ii) neuron bit error rate. Our results indicate that not all HTs suit every NN architecture, and their effectiveness varies depending on the evaluated error model. Techniques based on the range restriction of activation function consistently outperform others, achieving up to 58.23% greater mitigation effectiveness while keeping the introduced overhead at inference time low while requiring a contained effort in their implementation.

Keywords: reliability evaluation; software-based application-level hardening; graphic processing units; neural networks; fault tolerance

Academic Editor: Valeri Mladenov

Received: 23 January 2025
Revised: 27 February 2025
Accepted: 4 March 2025
Published:

Citation: Esposito, G.; Guerrero Balaguera, J.D.; Rodriguez Condia, J.E.; Sonza Reorda, M. Comparing Application-Level Hardening Techniques for Neural Networks on GPUs. *Electronics* **2025**, *1*, 0. https://doi.org/

1. Introduction

Artificial intelligence (AI) is increasingly utilized in various fields, particularly in consumer electronics and safety-critical systems, often employing neural networks (NNs). In autonomous vehicles, NNs process large amounts of sensor data in real time for decision-making and action. Hence, the reliability of NNs is vital for system performance. Edge AI allows for the implementation of NN algorithms in Internet of Things (IoT) applications and devices with limited resources [1–3]. Techniques like split computing [4] help integrate complex neural networks into systems using *commercial off-the-shelf (COTS)* devices, such as drones. Recent advancements in semiconductor technology have also made it possible to include AI accelerators, like GPUs, in embedded IoT devices [5].

Unfortunately, in small technology scaling (7 nm and below), semiconductor devices are more susceptible to permanent hardware faults caused by different phenomena (e.g., premature degradation, aging, or manufacturing defects) [6–8]. In fact, such faults

can silently propagate as errors through the NN execution, jeopardizing the application reliability (e.g., faults inducing wrong predictions that cause drone accidents) [9–12].

Fault effects can be reduced in AI hardware design by adding specialized structures for detection and mitigation. Nonetheless, these solutions may increase costs and power consumption and may not be available in COTS devices. Alternatively, software-based hardening techniques (HTs) seek to mitigate the effect of hardware faults in NNs by adopting NN architectural-level modification, including sometimes additional NN retraining. Although such NN architectural-level hardening solutions are flexible and cheaper than hardware-based solutions, there are still two main concerns about these solutions, (i) the assessment of NN hardening solutions may lead to optimistic results when using hardware-agnostic fault injections (FIs) [13–17], and (ii) to the best of our knowledge, every NN hardening solution is evaluated under different conditions (i.e., NN models, and error models), making it hard to determine which solution fits better in a different context.

In the first case, most of the reported works in the literature assess the effectiveness of NN hardening strategies against transient faults by resorting to random error corruptions claiming to describe hardware fault effects in different hardware elements [13–17]. Nonetheless, none of such works consider the interaction of such faulty hardware with the application execution (e.g., tiling matrix multiplication in GPUs).

For example, the evaluations in [13] focus on data path errors (due to transient faults) with the corruption of weights and neurons according to an error rate. However, the error rates are arbitrarily chosen within the range $[10^{-6}, 10^{-1}]$ and with random patterns that do not consider how the accelerator propagates the fault as error across the layer computation. In ref. [18], the authors resorted to a bit-masks approach (describing the deviations of weights and neurons due to the energy spent in memory access) proposed in [19] as error modeling mimicking the effect of the fault occurring in the SRAM cells with error rates that range in $[10^{-3}, 10^{-1}]$. Moreover, the authors of [14,15] injected multiple random bit-flips in neurons' output to mimic the effect of transient faults due to a particle that strikes a flip-flop during MAC operations or a memory unit.

These evaluation approaches often overlook propagation errors across NN layers, particularly where convolutional or linear layers are implemented, as in tiling matrix multiplications on embedded GPUs. A faulty GPU core affects only certain matrix multiplication tiles, leading to unique error effects at the NN layer that may influence the effectiveness of hardening techniques differently than standard evaluation methods do [20,21].

While instruction-level FIs can provide a more realistic reliability assessment of NNs regarding transient [22–24] or permanent faults [25–27], their application is limited to specific hardware devices, making it challenging to adopt such strategies on COTS embedded GPUs when evaluating hardening approaches. Furthermore, executing instruction-level FIs through an ad hoc framework [28–30] significantly increases simulation time.

Alternatively, hardware-aware evaluations at the application level offer a good balance between accuracy and evaluation time for assessing NN reliability and hardening strategies. For instance, refs. [31,32] use weight single bit-flip (WSBF) to evaluate permanent faults in hardware memory, resorting to the stuck-at fault model. Additionally, the authors of [21] explored the effect of faults in MAC units during NN layer execution through the neuron bit error rate (NBER) error model inspired by the results gathered in [20]. In particular their findings emphasize the necessity for error models that account for faulty matrix multiplication in Edge AI applications.

On the other hand, it is important to note that when assessing HTs, most research activities use varying test benches (e.g., NN architecture, datasets, target components, and other HTs). Consequently, results from previous studies are not applicable for selecting a specific hardening technique for a given application. For example, the authors of [14,15]

evaluated the proposed HT on 7 NNs on five datasets, while the authors of [16] evaluated their HT on three NNs and one dataset different from the previous work. Hence, the achieved results cannot be generalized for the selection of a specific HT for different test bench configurations. Moreover, as the training process strongly depends on the NN architecture along with the selected datasets, selecting a different test bench implies a different and non generalizable training setup when the HT requires a training step. This specificity hinders the generalizability of the HT implementation descriptions provided in the respective studies. In addition, to the best of our knowledge, there is no standard procedure for establishing a comprehensive ranking of HTs for edge computing scenarios.

Therefore, it is still unclear (i) whether the application-level hardening techniques are effective in terms of reliability enhancement when they are evaluated with hardware-aware error models and when they are implemented on any neural network and (ii) which HT shows the best trade-off between the cost for its implementation on the normal functioning of the NNs and the capabilities of the HT in enhancing NN reliability.

Based on the studied literature, we expect that not all the application-level HTs enhance the reliability of the NNs when they are subject to both WSBF and NBER error models considering different NN architectures and datasets.

Furthermore, performing an impact analysis on the implementation of the HTs can yield valuable insights regarding their feasibility of implementation on COTS constrained devices with limited computational capabilities employed in Edge AI domains, prior to evaluating their effectiveness in terms of reliability enhancement.

To prove our hypotheses , this work proposes a systematic evaluation methodology that combines two main aspects of the NN HTs: (i) the features of the HT when incorporated in a given NN and (ii) the fault mitigation capabilities. In the first aspect, we proposed a combination of qualitative (e.g., effort of implementation) and quantitative metrics (e.g., inference time overhead) to evaluate the cost for the HT implementation. In the second aspect, we used hardware-aware error models WSBF and NBER to assess the effectiveness of a HT to mitigate hardware faults in terms of accuracy degradation. The proposed evaluation methodology enables a direct comparison of the HTs in terms of the feasibility implementation and effectiveness as a fault countermeasure.

The key contributions of this article are as follows:

- The proposal of a systematic evaluation of different application-level software-based HTs applied to a representative set of NNs from Edge AI domains. The evaluations have been conducted through hardware-aware FI campaigns.
- The proposal, for the first time of qualitative and quantitative evaluation metrics specially developed to compare objectively different HTs for NNs, considering the engineering effort, the execution time overhead, and the impact on the accuracy of the NN.
- The proposal of a general training strategy enabling the adaptation of HT across different NN architectures. Such training is crucial in cases where the HT needs either fine-tuning or retaining stages.
- The proposed evaluation indicates that each HT is not always suitable for all the NN architectures considering both the perspectives of the HT implementation impact evaluation and their capabilities of counteracting faults.

We applied the evaluation strategy using five HTs (i.e., adaptive clipper [33], Ranger [14], Swap ReLU6 [34], median filter [13] and fine-grain TMR [13]) implemented on five NN architectures (Lenet5 [35], Mnasnet [36], Mobilenet V2 [37], Resnet18 [38], and Squeezenet [39]) evaluated on three datasets (i.e., MNIST [40], Cifar10 [41] and STL10 [42]).

The results show that reliability enhancement depends on the error model and test bench used in the evaluations. For example, Swap ReLU6 on Resnet18 experiences a

maximum accuracy degradation of 13% in WSBF FI campaigns compared to a fault-free scenario, while the same technique leads to over 35% degradation during NBER FI on the same NN architecture. In NBER FI campaigns, implementing Swap ReLU6 on Mobilenet V2 results in a 16% accuracy decrease. At the same time, when it is implemented on Resnet18, it achieves only 1.5% of degradation. In general, range-restriction-based HTs, like Adaptive Clipper and Ranger, offer better trade-offs, with Adaptive Clipper scoring up to 6/10 in effort, and only a 4.5% accuracy degradation. Inference time overhead is modest for Adaptive Clipper (up to 3.52%), while for Ranger, it reaches 7.54%. Notably, these HTs can effectively mask fault effects in up to 58.23% of cases during fault injections.

Section 2 covers backgrounds on application-implemented HTs, error propagation modeling and related works on the HT effectiveness assessment. Then, Section 3 describes the proposed hardware-aware application-level evaluation strategy. The experimental setup is presented in Section 4, followed by the experimental results in Section 5. Subsequently our results are summarized and compared with the state of the art in Section 6. Finally, Section 7 outlines conclusions and future remarks for our work.

2. Background

2.1. Neural Networks

In Edge artificial intelligence (AI) applications, convolutional neural networks are typically employed to solve tasks, such as image classification, object detection, or real-time signal processing, using convolutional operations as basic functional blocks. Every network design consists of a specific arrangement and set of layers. As shown in Figure 1, a possible convolutional neural network organization can involve the execution of (i) the convolutional layer, (ii) batch normalization layer, and (iii) activation function. The convolutional layer performs the dot product $\underline{w} \cdot \underline{x} + \underline{b}$ where $\underline{w}$ is the matrix containing the layer weights (i.e., kernel), $\underline{x}$ is the patch of the input whose shape matches the kernel size and, $\underline{b}$ is the bias vector. Batch normalization is used to enhance the stability of the training process by normalizing the inputs to each layer. Lastly, activation functions are generally used to make the convolutional neural networks able to learn/non-linear input/output relationships. For example, the rectified linear unit (ReLU) activation function applies the ramp function $ReLU(\underline{x}) = max(0, \underline{x})$. Building upon this function, several other ReLU-like functions have been developed, e.g., SELU [43], GELU [44], ReLU6 [37], andTanH.

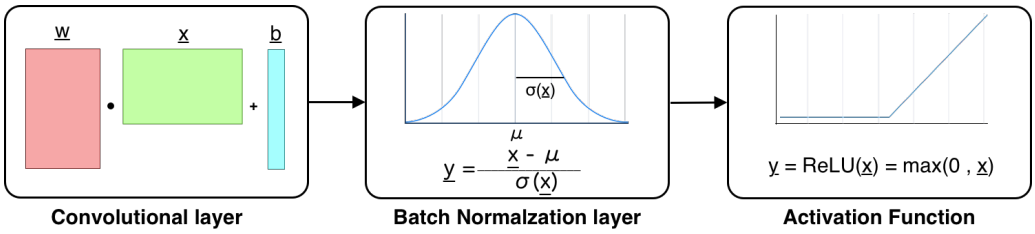


Figure 1. Basic convolutional block.

NNs designed for Edge AI are built upon the convolutional block, focusing on optimizing weight model size for greater efficiency in resource-constrained environments. The following models are five of the most representative NN architectures used in Edge AI applications.

LeNet, as one of the pioneering convolutional NNs, is relatively simple and lightweight, making it suitable for tasks with lower complexity [35]. It includes three convolutional layers, each employing 3×3 kernels with a padding of 1. The number of filters increases from 32 to 128 across these layers. Following each convolutional layer, a

2×2 MaxPooling operation is applied with a stride of 2, progressively reducing the spatial dimensions. The fully connected layers comprise 256, 128, and 10 neurons each.

MobileNet was originally designed for mobile and embedded vision applications [37]. Its architecture is built around inverted residual blocks and linear bottlenecks, which help reduce space complexity. In its topology, MobileNetV2 starts with an initial standard convolutional layer with 32 filters, followed by a series of 19 residual bottleneck layers. The inverted residual design incorporates shortcut connections, also known as skip connections, that bypass the inner bottleneck layers. This design enhances the propagation of gradients and reduces computational overhead. The bottleneck layers utilize an expansion factor, usually set to 6, which increases the number of channels before depthwise convolution is applied. The bottleneck layers allow the model to capture richer feature representations with minimal additional complexity. Additionally, the architecture uses ReLU6 as its activation function and includes a linear bottleneck to maintain low-dimensional features, ensuring high efficiency for mobile applications.

MnasNet is a convolutional NN specifically designed for mobile devices, incorporating real-world latency and accuracy as key objectives, thus making it suitable for Edge AI applications [36]. The architecture features a modular design that includes inverted residual blocks. The process begins with a 3×3 convolution, followed by batch normalization and ReLU activation to process the input image effectively. Within the feature extraction block, multiple inverted residual blocks are stacked together. Each inverted residual block consists of three phases: an expansion phase, which increases the number of features; a depthwise convolution phase, which performs spatial filtering; and a projection phase, which reduces dimensionality.

ResNet offers a good trade-off between depth and complexity [38]. It employs a unique architecture to address the challenges of training deep neural networks, particularly the degradation problem where adding layers reduces performance. Each residual block typically comprises two or more convolutional layers. Depending on the specific version of the NN (such as ResNet18, ResNet34, ResNet50, ResNet101, or ResNet152), the model contains a certain number of residual blocks, enabling it to learn more complex input–output functions.

SqueezeNet is know for its exceptionally small model size, achieving AlexNet-level accuracy with 50 times fewer parameters [39]. It is a compact and efficient neural network. Its architecture begins with a simple convolutional layer, followed by eight Fire modules, which consist of a sequence of squeeze and expand layers that operate along the depth dimension. Each module gradually increases the number of filters. The final layer employs max-pooling with a stride of 2 to further reduce dimensionality in the later stages, allowing the model to maintain accuracy despite its smaller size. A unique aspect of SqueezeNet is its lack of fully connected layers; instead, feature learning is integrated within the transformations between the Fire modules. This design enables SqueezeNet to achieve up to a 50-fold reduction in model size compared to AlexNet while maintaining comparable or even better performance.

The design of each NN, whether through residual connections in ResNet or the Fire modules in SqueezeNet, ensures that gradients can flow efficiently through the layers during back-propagation [45].

Since NNs rely on differentiable operations, it is possible to compute the gradients of the loss function with respect to the network weights. Once the gradients are obtained, an optimizer, such as stochastic gradient descent (SGD) [46] or Adam [45], updates the weights to minimize the loss function. These forward and backward steps are repeated for each batch of training data over several epochs, where an epoch refers to a complete pass through the entire training dataset. During each epoch, the adjustments made in

the multidimensional weight space are scaled by the learning rate (lr). The lr can also be progressively adjusted using a scaling factor. By appropriately configuring the hyper-parameters, it is possible to ensure that the training algorithm converges to a sub-optimal state with a low loss value.

2.2. GPU Organization and Fault to Error Propagation

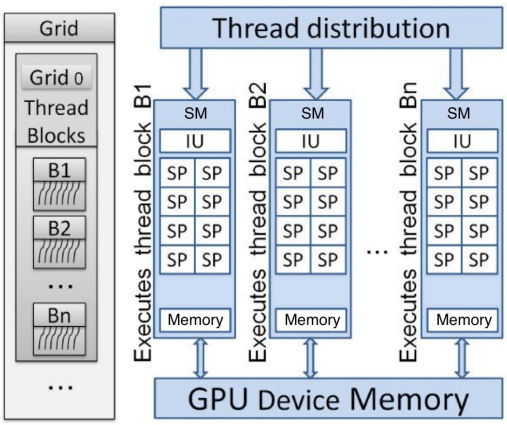


Figure 2. Hierarchical GPU organization [47].

Neural networks are computationally intensive algorithms often characterized by time-consuming and resource-demanding processes; consequently, hardware accelerators, such as GPUs, are generally employed to boost NN operations in parallel, reducing their execution latencies. In particular, the hierarchical structure of the GPU, as illustrated in Figure 2 and presented in [47], improves the execution of parallel tasks efficiently. The left side of Figure 2 represents the single-instruction multiple-thread (SIMT) programming model of the GPU. The program organizes the threads in grids and blocks. The blocks (B0, B1, B3) are collections of threads, and a grid (G0) is a collection of blocks.

On the right hand side of Figure 2, the GPU hardware components are depicted: the GPU scheduler defines the policy for distributing the threads to the physical components. Each streaming multiprocessor (SM) executes to a thread block (TB), while all SMs share the GPU device memory. An SM includes local memories and register files to support the parallel thread execution. Subsequently, the SM schedules several warps into the scalar/streaming processor (SP) of fused multiply-accumulate (FMA) core to perform integer, floating-point, and trigonometric operations. The description of the hierarchical representation provides useful insights to better understand the error propagation induced by a physical defect throughout the GPU architecture.

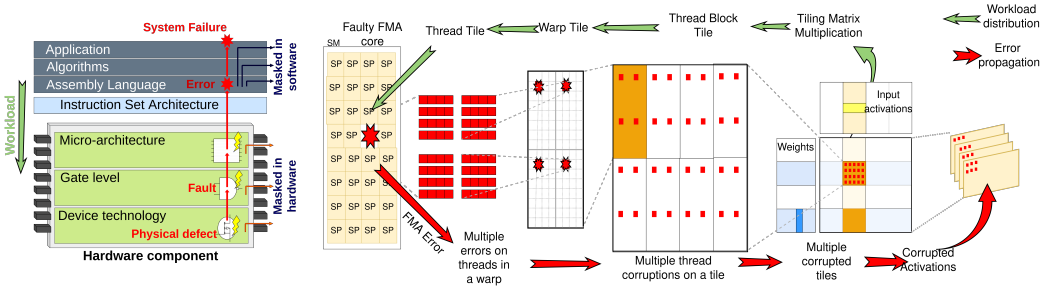


Figure 3. Effect of the fault propagation up to the application level.

Despite the benefits in terms of the performance stemming from the execution of parallel programs, as discussed in Section 1, physical defects in the semiconductors of

GPUs and their propagation are primary sources of faults. On the left hand side of Figure 3, the propagation of the error induced by physical defect to the application level is depicted. It illustrates the varying levels of impact the defect can have, affecting both hardware and software. This supports the intuition behind the NBER error model that we used for our evaluations, which describes the errors occurring in the output of the fused multiply-accumulate cores during the execution of matrix multiplication algorithms.

Physical defects can occur in a system due to manufacturing flaws or external factors that disrupt the GPU's operation, either temporarily or permanently. Specifically, these defects may trigger and propagate as hardware faults (for example, corrupting the logic state on a gate) across the microarchitecture during the execution of an application, leading to system failures or errors.

In more detail, corrupted logic states can significantly impact the execution of assembly instructions, leading to incorrect implementations of algorithms, such as the matrix multiplication algorithm. As shown on the right-hand side of Figure 3, the algorithm divides inputs, feature maps, and weight arrays into smaller submatrices known as tiles, which are then distributed among the GPU's parallel cores. The computation is carried out through several TB computations, which are further divided into smaller tiles at the warp level within the GPU's SMs. This tiling method requires each thread in a warp to compute up to four small matrix multiplications between tiles, each of size 4×4 [48]. When a single SM has a faulty FMA core, errors can propagate to multiple threads within a warp, resulting in data corruption at the output of the tile. Consequently, if more than one tile is processed by the faulty SM, a similar error pattern will likely affect the results of the algorithm.

However, the structural organization of the hardware or the implicit nature of the code in the application may mask some of these effects, meaning they can go unnoticed if they do not produce a visible corruption in the application's outputs or the system itself.

Although software mechanisms can sometimes conceal the effects of faults, the main concern is with faults that propagate silently, resulting in overlooked errors in the outputs of the NN, also known as silent data corruptions (SDCs) [9,10,49]. Therefore, it is crucial to thoroughly evaluate the effectiveness of application-level HTs that instrument the NN topology to tackle such propagation at the application level.

2.3. Software Solutions for Reliability Enhancement of NNs

In the literature, several methodologies have proposed the enhancement of the system's reliability by tacking hardware fault impacts on NNs through the adaptation of the components of the software [14,50,51], hardware [52–54], or a combination of both (e.g., employing software-hardware co-design) [13,55].

Although the approaches that involve hardware design adaptation seek to increase system robustness resorting to well-known approaches [52,56–58] regardless of the application, it has significant drawbacks, such as the need for modifications during the design phase, which can be impractical for system designers. Additionally, these hardware changes can increase power consumption and costs, limiting their use in embedded systems.

In contrast, software-based reliability modifications are more efficient, cost-effective, and easier for system designers to implement, due to their inherent flexibility and scalability.

HTs that are software-based at the application level mainly involve the modification of the NN topology [13,34] or the modification of the training process [13,33] while avoiding area and hardware implementation cost overhead. The state of the art includes the HTs based on the range restriction [14,33,59] where activation functions, such as ReLU, Swish, or Tanh functions, included in the most common NN architectures, are bounded within a specific interval. In addition, a few studies have explored the possibility of performing a neural architectural search based on reliability-oriented metrics for evaluating NN topol-

ogy [50]. Finally, taking reliability into account during the training process to make the model more resilient at inference time has been addressed as well [34,60]. These techniques fall under the category of fault-aware training HTs, which aim at minimizing the error that a fault can induce at the application level during the training phase to improve the robustness of the NN.

The following subsections describe the implementation detail of the state-of-the-art HTs (i.e., Adaptive Clipper, Ranger, Swap ReLU6, median filter and fine-grain TMR).

2.3.1. Adaptive Clipper

This HT was proposed in [33] and replaces the *ReLU* activation functions with the HardTanH activation function that allows us to set the lower and upper bounds so that the new architecture is ready for the additional training step. The new neural network is trained using an identical hyperparameter configuration and training strategy as the original architecture. This includes the same learning rate, batch size, and number of epochs, ensuring that both networks undergo a comparable training process.

2.3.2. Ranger

Ranger places the HardTanH after each NN layer, assigning proper lower and upper bounds to the activation functions of all layers in the NN [14]. The HardTanH function is an activation function that allows an entry-wise check on the input feature map to be performed. While performing inference on 20% of the training dataset, the upper and lower bounds are computed as the minimum and maximum entries recorded from the activation functions that belong to the original NN architecutre. After applying Ranger, if the input value exceeds the upper or lower bound, it will be set to the upper or lower limit accordingly. If it lies within the acceptable range, the value remains unchanged.

2.3.3. Swap ReLU6

Swap ReLu6 changes the order of batch normalization and *ReLU* activation function positions [34]. The ReLU activation functions are bounded to 6 (i.e., *ReLU6*). Afterwards, the new architecture is fine-tuned with a new hyperparameter configuration. The convolutional layer is the most computationally intensive component of a neural network, which makes it the most susceptible to faults. Consequently, applying batch normalization to corrupted feature maps can alter the distribution of those feature maps. This alteration may lead to faster and more significant error propagation throughout the entire batch normalization output. Consequently, the authors hypothesize and prove that placing an upper bounded activation function right after the convolutional layer allows them to counteract the effect of faults.

2.3.4. Median Filter

Median filter is an HT developed to mitigate hardware faults on AL accelerators calculating the median operation to the outputs of convolutional and linear layers [13]. The typical hardware implementation of the median operation generally involves 2 main steps: sorting the array subject to median computation and selecting the second array element. However, sorting an array of length L (Sort-L) hinders the parallelization of the workload through CUDA-implemented functions. For this reason, as suggested by the authors of [13] taking inspiration from the work [61], a parallelizable, thus efficient, implementation of the median operation is implemented as the composition of L steps of Sort-2 algorithms. The median operation is applied to the input of the convolutional and linear layer feature maps (FMs). However, including the median operation requires the NN to be trained from scratch or fine-tuned so that the new layer can also process the gradients computed during the training process.

2.3.5. Fine-Grain TMR

This HT executes three times every neuron computation (i.e., dot product operation) at the software level [13]. In addition, The results from these three computations undergo a majority voting process, where the most common output is selected as a fault-free result. The main targets of this HT are the convolutional and linear layers. It is worth noting that this strategy does not involve the NN retraining process.

3. Proposed Evaluation Methodology

This section introduces the evaluation methodology used in this work to assess a set of application-level HTs implemented on a representative collection of NNs from the edge AI domain. Figure 4 depicts the three main steps involved during the evaluation process: (i) HT implementation, (ii) reliability assessment through hardware-aware fault injection campaigns, and (iii) data analysis.

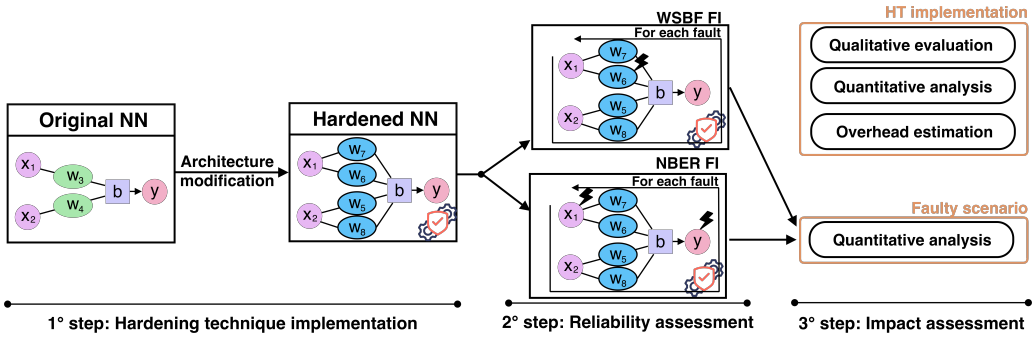


Figure 4. Hardening technique evaluation procedure.

The first step involves the implementation of the HTs on the NN architectures. For this purpose, a set of NN architectures are modified according to the HTs (e.g., changing the order of the layers or including an additional operation in the original NN architecture).

This stage involves the implementation of the HT on a set of target NN architectures. To make such HT successful, it is necessary to consider two main aspects. The first one corresponds to the required changes or degree of modifications required to implement a given HT on an NN model. For example, an HT can involve replacing a ReLU activation function with a ReLU6 activation function. This modification is straightforward and can be implemented simply by substituting one layer for another. However, some strategies are more complex and require deeper modifications to the NN, such as adding layers that include custom operations, (e.g., a median filter or TMR operations). On the other hand, the HTs may involve only fine-tuning the model by using the pre-trained weights of the original architecture. In contrast, another HT implementation might involve training the NN using a hyperparameter configuration that significantly differs from the one used to train the original NN (e.g., the required number of epochs can increase with respect to the original NN training process).

The second step consists of an experimental stage that conducts the reliability assessment to assess the implemented HT's reliability enhancement capabilities. Specifically, FI campaigns are conducted at the application level resorting to 2 hardware-aware error models: (i) WSBF and (ii) NBER.

The last step in our evaluation procedure involves an evaluation that targets (i) the HT implementation impact and (ii) the effectiveness of the HTs as fault countermeasures. Remarkably, the implementations of the considered HTs require modifications in the NN architecture, which can increase inference time. Additionally, this new architecture may require an extra training phase, and there is a possibility that the new neural network may not reach the same level of convergence as the original architecture. For this reason,

during the HT implementation impact evaluation, we evaluated the impact on the NN accuracy (quantitative assessment), the required effort to make HTs successful (qualitative evaluation) and the introduced overhead (overhead estimation), whereas in the faulty scenarios (during WSBF FI and NBER FI), we evaluated the NN accuracy degradation.

Concerning the HT effectiveness study, we have conducted a quantitative assessment on the NN accuracy degradation recorded after the WSBF and NBER FI campaigns.

3.1. Error Models

In this work we adopted 2 hardware-aware error models: (i) WSBF to mimic the effect of permanent faults in the GPU memory units, resorting to a stuck-at fault model [31,32], and (ii) NBER, which models the error propagation during the execution of tiling matrix multiplication algorithms [21]. The validation step was performed with a modified version of PytorchFI, a tool that is generally used to perform FIs at the application level [62–64]. We used PytorchFi to implement WSBF FI [65] and NBER FI [20,21].

3.1.1. Weight Single Bit-Flip

This error model is widely used to validate the effectiveness of SW HTs [14] by simulating faults in the memory shared among different threads of the same SM. As pointed out in [66], WSBFs FI can provide preliminary hints for the reliability assessment in front of measures, such as error coverage and error resilience. The number of faults to be injected is computed by resorting to the statistical method introduced by [67] and detailed in Equation (1).

$$n = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times p \times (1-p)}} \tag{1}$$

Starting from an initial population size N, the resulting total number of faults n satisfies an error margin e, the cut-off point t corresponding to a confidence level, and the probability p of randomly picking a specific value of the features describing the fault. When weights are corrupted, the characteristics of a fault are: *kernel K*, *channel Ch*, *row R*, *column Col*, and *bitmask b*, which collectively define the exact position of the bit to flip in the parameters tensor to perturb.

(*K*,*Ch*,*R*,*Col*,*b*) is a vector of integers, and one of the assumptions behind Equation (1) requires that

- K $\sim$ Uniform[0,batch_size],
- Ch $\sim$ Uniform[0,tensor_depth],
- R$\sim$ Uniform[0,tensor_width],
- Col $\sim$ Uniform[0,tensor_height],
- b $\sim$ Uniform[LSB,MSB].

Data (i.e., weights and entries of the intermediate FMs) are represented in *FP*32 format and, according to the IEEE Standard for Floating-Point Arithmetic (IEEE 754), a generic FP32 encoded number x, in binary scientific notation, can be written as shown in Equation (2).

$$x = (-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times (1.\text{fraction}) \tag{2}$$

where $sign = 1$ if x is negative, and $sign = 0$ if x is positive, then $exponent$ is an 8-bit unsigned integer with a bias of 127, and in the end, the fraction represents the mantissa of the x binary encoding. As we can see from the equation, a bit-flip in the mantissa does not affect the real representation of x, while a bit-flip performed on the exponent would induce a high error. To support this statement, several works [32,68] have evaluated the effect of bit-flips on the binary encoding of the weights/neurons of the NN exploring the bit-wise impact on error percentage as well. Consequently, due to the negligible induced

error by performing a bit-flip on the mantissa bits, as for b, we set the least significant bit (LSB) to 19 and the most significant bit (MSB) to 31, so as to consider the sign bit, the exponent bits, and the mantissa bits from 23 to 19. However, for what concerns the upper bounds of the ranges from which the components of (*K*,*Ch*,*R*,*Col*) are extracted, *batch_size*, *tensor_depth*, *tensor_width*, and *tensor_height* represent the dimensions of the parameter tensors to corrupt in case the injection occurs in a convolutional layer.

After extracting a sufficient number of faults and generating a fault list, we can use PytorchFI to perturb the weight parameters at run-time. At the end of each FI campaign, a specific evaluation process is performed, classifying faults into different categories according to their effects.

3.1.2. Neuron Bit Error Rate

Typically, the most prevalent fault injection method for evaluating software-based HTs involves corrupting multiple neurons neglecting any error propagation pattern. We instead resort to a hardware-aware error model that simulates the faults in the FMA cores on a single SM that executes parallel tiling matrix multiplication algorithms on a graphical processing unit (GPU) device [21] . While the NN's weights are fixed, the neuron's output changes depending on the input FM to the target layer. Before the parameter corruption stage, the framework generates the fault list, which contains the layer, the tile size (TS), the block error rate (BlER), the neuron error rate (NER), and the bit location. All these fault features define the output FMs (i.e., intermediate NN output) portion to corrupt along with the specific bit locations to mimic the effect of permanent faults affecting the FMA cores in the GPU device [20,21]. According to the TS, each channel of the intermediate feature maps is divided into several possibly overlapping blocks. Consequently, the rate of corrupted blocks is defined as $BlER = \frac{\#target\_blocks}{\#total\_blocks}$. Within the block, the rate of corrupted neurons is defined as $NER = \frac{\#target\_neurons}{\#total\_neurons}$. In the end, a bit-flip is performed on the target bit location, considering the binary encoding of the selected entries. Due to the fixed bit location for each experiment, the rate of parameters over all the FM entries represents the bit error rate that can be obtained as $BER = NER \times BlER$.

3.2. Data Analysis

3.2.1. HTs Implementation Impact Evaluation

To conduct the study on the impact of the HT implementation, we introduced the effort of implementation (EoI) that contributes to better grade in an objective and qualitative way, the best HT to be implemented considering 2 main features: complexity and training adaptations. As shown in Table 1, we assigned an EoI score ranging from 0 to 10, which is based on the composition of two features. The first feature is called *complexity*, and it can range from 0 to 5. A higher score indicates that more topological modifications are required for the implemented HT design. We considered assigning a complexity from score 1 to 3 when the layers of the original NN topology are replaced with another one, while complexity is assigned a score of 4 or 5 when an additional layer is introduced within the original NN topology. *Training adaptations* also can range from 0 to 5. This feature measures the extent of differences between the training process of the original NN architecture and the training process needed for the HT implementation, with higher values assigned for more significant modifications. A lower degree of training adaptations (scores of 1 or 2) indicates that the new neural network (NN) architecture only needs fine-tuning. This fine-tuning can either follow the original hyperparameter configuration (score = 2) or make the training process lighter than the original setup (score = 1). A score of 3 or higher is assigned when the new architecture requires retraining with a hyperparameter setup that may be similar (score = 3) or different (score > 3) from the original configuration.

Table 1. Scoring system (out of 5) based on the effort required for implementation, considering both the complexity of NN modifications and the necessary adjustments to the training process.

Score (Out of 5)	***Effort of Implementation (Complexity + Training Adaptations)***	
	Complexity	***Training Adaptations***
0	No architecture modifications	No need of a training step
1	# of replaced layers $\leq$3	Need fine-tuning with new # epochs $\ll$ original # epochs
2	3 < # of replaced layers $\leq$ 10	Need fine-tuning with new # epochs $\approx$ original # epochs
3	# of replaced layers $\geq$ 10	Need re-training original setup = new setup
4	The NN topology is modified (e.g., changed layer ordering)	Need re-training with the same setup but new # epochs > original # epoch
5	Additional operations are introduced in normal NN functioning	Need re-training with a different setup

After setting up the new NN topology with the HT implementation, we evaluate the accuracy with Equation (3) according to [45].

$$Acc = \frac{number\_of\_correct\_predictions}{total\_number\_of\_predictions} \tag{3}$$

where $number\_of\_correct\_predictions$ indicates the number of elements that the NN correctly classified, while the $total\_number\_of\_predictions$ represents the overall samples on which the NN made inferences.

During our impact assessment on the HT implementation, we considered the computational overhead introduced in terms of increased algorithm execution time. Given the necessity for a near real-time response from the algorithm running on the device in focus for this work, it is crucial to quantify the computational overhead introduced by the HT implementation. Consequently, we evaluate the inference time overhead (ITO) with Equation (4).

$$ITO = \frac{IT_{hard} - IT_{base}}{IT_{base}} \times 100 \tag{4}$$

where IT_{hard} and IT_{base} are the execution times of the NN inference when the HT is implemented and when it is not implemented, respectively.

Since the execution time depends on several factors (e.g., software libraries or the underlying hardware executing the NN inference), we created a technology-independent metric that estimates the inference time (IT), resorting to the count of the number of basic operations (i.e., comparison, addition, subtraction, multiplication, and division) that each HT adds, taking as reference the application level operations to the original NN workload. However, from the hardware perspective, each operation involves a different complexity in terms of bit-wise operations. For this reason, we evaluated the IT as a weighted average of the executed operations according to Equation (5).

$$IT = \frac{(w_{comp} \cdot \#comps) + (w_{add} \cdot \#adds) + (w_{sub} \cdot \#subs) + (w_{mul} \cdot \#muls) + (w_{div} \cdot \#divs)}{w_{comp} + w_{add} + w_{sub} + w_{mul} + w_{div}} \tag{5}$$

where #comps, #adds, #subs, #muls, and #divs represent the number of executed comparison, addition, subtraction, multiplication, and division operations, respectively, while w_{comp}, w_{add}, w_{sub}, w_{mul}, w_{div} represent the arbitrary weights corresponding to each operation.

EoI and ITO are crucial to determine the feasibility of implementation of a given HT during the design stages of an NN architecture, even before it can be deployed on an edge device. In addition, our overhead estimation can give hints on the power consumption trend because the higher the ITO, the higher the required computational effort, thus, the higher the power consumption.

3.2.2. HT Effectiveness Evaluation

The adopted error models enable us to provide two distinct assessments (WSBF and NBER), allowing us to compare the HTs. Additionally, the adopted metrics (i.e., relative accuracy degradation (RAD), NN accuracy) are collected for all hardening strategies, subjecting each one to the same comprehensive evaluation framework. For WSBF FI campaigns, injected faults are classified into different categories depending on the severity of their impact on the application performance during inference. Inference is performed on a set of images, and in the end, an accuracy metric is used to evaluate the performance of the NN. The accuracy metric is defined as the percentage of correctly classified images over all the tested images. Consequently, during the fault classification stage, the framework uses the RAD to quantify the NN performance degradation. Mathematically, RAD is defined in Equation (6) [69].

$$RAD = \frac{ACC_{fault-free} - ACC_{faulty}}{ACC_{fault-free}} \tag{6}$$

where $ACC_{fault-free}$ represents the golden accuracy, and ACC_{faulty} is the accuracy of the corrupted DNN. Therefore, after the inference of the corrupted model, the fault is labeled as *critical silent data corruption (critical SDC)*. If the NN prediction is different than the fault-free one, the corresponding predictions are compared with the ones generated in the fault-free scenario. The fault classified assigns the label *masked* if they match (because the injected error does not change the inference output) or *safe silent data corruption (Safe SDC)* (when the fault impacts the outputs but does not change the prediction). In case the fault has produced an invalid output (e.g., the prediction output is composed of only NaN values), it is classified as a *detectable unrecoverable error (DUE)*. On the other hand, for NBER FI, the corruption of multiple components simultaneously does not allow a fine-grained fault classification depending on the RAD.

After running the WSBF FI campaign, the best outcome that we can obtain by implementing an HT on an NN is the decrease in critical SDCs, as well as the increase in masked labels (i.e., lower impact in NN accuracy). At the same time, after both WSBF and NBER FI campaigns, we also can see a reliability enhancement effect from the NN accuracy trend to have a more detailed and quantitative evaluation of HT effects with respect to fault class distribution (i.e., critical SDC, Safe SDC, masked and DUE).

4. Experimental Setup

We conducted a series of hardware-aware FI campaigns (i.e., WSBF and NBER), examining the 5 SW-implemented HTs introduced in Section 2.3. The effectiveness of the different HTs was evaluated on five of the most representative NN architectures: MnasNet, MobileNet V2, ResNet18, LeNet5, and SqueezeNet. The first three NNs were trained and tested on the Cifar10 [41] dataset, LeNet5 on the MNIST [40] dataset, and SqueezeNet on the STL10 [42] dataset. The choice of the NN–dataset pairings was guided by the original articles that introduced the tested NN architectures, with the exception of Resnet18.

MobileNet V2, MnasNet, and ResNet18 all utilize skip connections, but they employ them in different ways. MobileNet V2 and MnasNet incorporate skip connections within an inverted residual block, while ResNet integrates them in a residual block. Testing these three neural network architectures on the same dataset allows us to evaluate the impact of faults when HTs are implemented across these different architectures.

As stated in Section 2.3, implementing Adaptive Clipper, Swap ReLU6, and median filter HTs requires an additional step of retraining or fine-tuning according to the implementation methodology. The original HT implementation, as it is described by the corresponding articles, shows a hyperparameter configuration that is suitable only for the NN architectures under test. For this reason, we adopted a trial-and-error approach to the find the hyperparameter configuration that allows the model to converge to an acceptable level of accuracy (at most 10% of degradation with respect to the NN version without the hardening). required by the baseline neural network architecture; the hyperparameters are selected through a trial-and-error approach The description of the original implementation of HTs does not explicitly specify, when training is required, whether the new NN architecture needs to undergo fine-tuning or a complete training step. For this reason, we initially attempted to fine-tune the model using a number of epochs ranging $[0, original\_number\_of\_epochs]$, where $original\_number\_of\_epochs$ indicates the number of epochs originally needed to train the NN without applying the HT. In case the model does not successfully converge, a complete training step is required, and the number of epochs ranges $[original\_number\_of\_epochs, 2 \times original\_number\_of\_epochs]$.

During the hyperparameter tuning stage, (i) the number of epochs required for training convergence is increased until the model does not reach an acceptable validation accuracy score, (ii) the NN weight optimization algorithm always refers to the one used to train the original NN architecture, (iii) the initial lr ranges within $[\frac{original\_lr}{2}, original\_lr]$ when fine-tuning is required and within $[original\_lr, 2 \times original\_lr]$ when re-training is required. It is reduced if the training process ends with a steady training loss and an accuracy score below 74%, (iv) the scaling factor for each epoch multiplies the current lr, and it is kept constant. Finally, we reported the best hyperparameter configuration in Table 2 for each architecture under test, along with details on whether the neural network architecture requires retraining from scratch. Table 2 illustrates that although Swap ReLU6 does not need retraining when deployed on Lenet5, Mnasnet, and Mobilenet V2, the number of epochs remains consistently close to the number of epochs utilized for baseline convergence (i.e., 16, 100, 100, respectively). Conversely, as the Squeezenet architecture lacks a batch normalization layer, it is sufficient to fine-tune the baseline architecture for 2 epochs. In contrast, Adaptive Clipper demands retraining, regardless of the chosen NN architecture and with the identical baseline training configuration. In the end, despite the median filter's advantageous impact on fault tolerance, it necessitates many epochs for model convergence. For example, Resnet18 requires retraining and 198 epochs to be trained.

For the FI campaigns, we applied an extensive evaluation of every error model used in this work, as follows:

- WSBF FI: each statistical FI campaign is run such that a confidence level (CL) of 98%, error margin (e) of 1%, and an instance sampling probability (p) of 50% are satisfied. Moreover, since the weights are $FP32$ encoded, LSB = 19 and MSB = 31 due to the lower impact of bit-flip on the previous bits.
- NBER FI: the injection of errors on the Neurons' output used the TS, which is fixed to 16, corresponding to the typical TS managed by an FMA core; and a NER per tile, which takes values within the range [0.4%, 4%]. Similarly, the BlER takes 10 values within the same range as the NER parameter. The values that have been chosen for NER and BlER are based on the result that the authors of [70] obtained during more accurate FI

campaigns at a lower level of abstraction (i.e., the instruction set architecture level). Consequently, each fault represents a specific combination of the vector (TS, BlER, NER, *bit_location*) where bit location is chosen within the range [LSB, MSB], and LSB = 19 and MSB = 31.

Table 2. Hyper-parameter configuration used for training each NN version that allows us to reach the highest validation accuracy. Red colored cells correspond to the best hyperparameter configuration found to train NN versions when the validation accuracy does not reach 74%. In this table, the HTs names are abbreviated as follows: baseline (BL), Adaptive Clipper (AC), Swap ReLU6 (SR) and median filter (MF).

NN Architecture	*HT*	*Epochs*	*Optimizer*	*Initial Lr*	*Lr Scaling Factor (Per Epoch)*	*Require Retraining? (Y/N)*
Lenet5	**BL**	16	SGD	0.001	0.9	N
	AC	16	SGD	0.001	0.9	Y
	SR	9	SGD	0.001	0.9	Y
	MF	20	SGD	0.0005	0.9	Y
Mnasnet	**BL**	100	Weighted Adam	0.01	0.9	N
	AC	100	Weighted Adam	0.01	0.9	Y
	SR	200	Weighted Adam	0.02	0.9	Y
	MF	200	Weighted Adam	0.02	0.9	Y
Mobilenet V2	**BL**	100	SGD	0.02	0.9	N
	AC	100	SGD	0.002	0.9	Y
	SR	96	SGD	0.002	0.9	N
	MF	200	SGD	0.04	0.9	Y
Resnet18	**BL**	100	SGD	0.001	0.9	N
	AC	100	SGD	0.001	0.9	Y
	SR	74	SGD	0.0025	0.9	N
	MF	198	SGD	0.005	0.9	Y
Squeezenet	**BL**	118	SGD	0.001	0.9	N
	AC	118	SGD	0.001	0.9	Y
	SR	2	SGD	0.001	0.9	N
	MF	236	SGD	0.001	0.9	Y

The Weights FI campaigns were performed on the Leonardo HPC hpc@cineca (https://www.hpc.cineca.it/) with a booster partition that contains 3456 nodes with a single socket Intel Ice Lake CPU 32 cores and an Intel Xeon Platinum 8358, 2.60 GHz TDP 250W. Each node is equipped with 4× NVIDIA Ampere GPUs, 64 GB HBM2e NVLink 3.0 (200 GB/s). The experiment workload was parallelized among the cluster nodes, allowing the injection of a sufficient number of faults to satisfy CE, e, and p. In particular, 17,630 errors are injected in Lenet5 architecture, 108,392 errors in Mnasnet, 112,760 errors in Mobilenet V2, 72,954 errors in Resnet18, and 81,059 errors in Squeezenet. Similarly, NBER FI campaigns were deployed on the hpc@polito (http://www.hpc.polito.it) 6 node cluster with 2 Intel Xeon Scalable Processors Gold 6130 2.10 GHz 16 cores and equipped with 6 NVIDIA Tesla V100 SXM2, 32 GB, and 5120 CUDA cores able to inject all the possible combinations of the vector components (TS, BlER, NER, *bit_location*) and repeating each experiment 10 times. In total, 6500 errors are injected for each NN architecture.

Finally, during the statistical evaluation step, the ITO is calculated using the assigned weights for each operation. Inspired by the hardware implementation of the different oper-

ations and fixing the representation of the entire subject to the reported binary operators, we assigned the weights reported in Table 3.

Table 3. Assigned weights for each MAC operation.

Operation	*Weight*
Comparison	1
Addition	2
Subtraction	2
Multiplication	3
Division	4

5. Experimental Results

This section presents a comprehensive analysis of the effectiveness of the five different HTs implemented on five representative edge AI neural network architectures, as described in Section 3.

The following subsection will provide an overview of the obtained results from two different perspectives: (i) HT implementation impact assessment and (ii) HT effectiveness in terms of fault countermeasure when subject to a hardware-aware reliability assessment.

5.1. HT Impact Assessment

5.1.1. Trading Off EoI and Accuracy

Table 4 shows the results of the qualitative and quantitative studies conducted on the HTs when implemented on the NN architectures under test. This study provides a detailed view of the impact of HTs on NN architectures immediately after their implementation, focusing on the effort involved and the accuracy of the resulting model. In particular, Table 4 illustrates the EoI as the composition of complexity and training adaptations, the accuracy of the resulting model and the accuracy of the model without any HT implemented.

As an initial result, it is important to note that not all hardening techniques, when their implementation includes a training step, allow for the convergence of the training algorithm.

For example, median filter requires a significant effort to be adapted. In fact, it does not guarantee training algorithm convergence for all NN architectures at an acceptable level of test accuracy. Specifically, the training of Mnasnet, Mobilenet, and Squeezenet converged to 35.40%, 54.50%, and 62.34% accuracy, respectively. Nowadays, most NNs used for Edge AI applications (e.g., Mnasnet, Mobilenet, and Squeezenet) involve an optimized version of the convolution algorithm, which resorts to the grouping strategy. This strategy allows the optimization of the RAM footprint and the inference time simultaneously by convolving only a group of channels with a corresponding group of kernels, thus avoiding the computation of redundant operations. A drawback of the grouping strategy is that the processed tensors are shaped according to the optimized convolution algorithm, which makes the implementation of median filter at the application level (requiring a window computing median and sliding along the tensor channel dimension) almost impractical.

Similarly, Mnasnet's training algorithm converged to the accuracy of 35.40% when Swap ReLU was implemented. By observing the distribution of the Mnasnet intermediate feature, we have noticed a higher variance than the intermediate feature of other NN architectures (up to 10%). Consequently, the strict activation functions upper bounding to 6 makes the convergence of the Mnasnet training algorithm impractical.

Table 4. Qualitative assessment on the EoI of each hardening technique for each NN architecture. The EoI scores are reported in terms of *complexity* and *training adaptations*, along with the test accuracy score reached after the HT implementation. The red colored cells highlight the accuracy score whose drop with respect to the baseline is non-negligible.

NN Architecture	*HT*	*Effort of Implementation*		*Accuracy*	*Baseline Accuracy*
		Complexity	*Training Adaptations*		
Lenet5	**Adaptive Clipper**	1/5	3/5	98.00%	98.00%
	Ranger	1/5	0/5	98.00%	
	Swap ReLU6	4/5	3/5	97.00%	
	Median Filter	5/5	5/5	96.00%	
	Fine-grain TMR	5/5	0/5	98.00%	
Mnasnet	**Adaptive Clipper**	3/5	3/5	83.00%	83.00%
	Ranger	3/5	0/5	83.00%	
	Swap ReLU6	4/5	2/5	70.3%	
	Median Filter	5/5	5/5	35.4%	
	Fine-grain TMR	5/5	0/5	83.00%	
Mobilenet V2	**Adaptive Clipper**	3/5	3/5	74.46%	78.99%
	Ranger	3/5	0/5	78.99%	
	Swap ReLU6	4/5	2/5	84.36%	
	Median Filter	5/5	5/5	54.5%	
	Fine-grain TMR	5/5	0/5	78.99%	
Resnet18	**Adaptive Clipper**	2/5	3/5	82.59%	85.72%
	Ranger	2/5	0/5	85.72%	
	Swap ReLU6	4/5	2/5	77.00%	
	Median Filter	5/5	5/5	76.00%	
	Fine-grain TMR	5/5	0/5	85.72%	
Squeezenet	**Adaptive Clipper**	3/5	3/5	82.50%	77.50%
	Ranger	3/5	0/5	77.50%	
	Swap ReLU6	2/5	1/5	76.25%	
	Median Filter	5/5	5/5	62.34%	
	Fine-grain TMR	5/5	0/5	77.50%	

In Lenet5, the Ranger method achieves an EoI score of 1/10. This low score is attributed to the use of only three activation functions, which contributes to a complexity rating of 1/5. Additionally, Ranger does not require retraining and maintains the baseline accuracy of 98%. Similarly, the Adaptive Clipper method also does not experience any drop in accuracy when applied to Lenet5. It has a relatively low EOI score of 4/5, reflecting a complexity of

1/5 and a training adaptation degree of 3/5. However, it is important to note that Adaptive Clipper requires retraining using the same setup as the baseline model. Fine-grain TMR also maintains its accuracy with no drop. Even though it can be implemented without additional training, the introduction of new layers results in a complexity rating of 5/5. In contrast, both Swap ReLU6 and median filter methods have higher EOI scores, with 7/10 and 10/10, respectively. These methods involve substantial changes to the neural network (NN) architecture, such as altering the order of sequential layers and adding new custom operations. Consequently, they are assigned complexities of 4/5 and 5/5, respectively. Regarding training adaptations, Swap ReLU6 requires only fine-tuning of the NN with fewer epochs than the original training, while the median filter necessitates retraining with a different hyperparameter configuration. Moreover, both Swap ReLU6 and median filter incur a slight accuracy reduction compared to the baseline model, with drops of 1% and 2%, respectively. The increased effort involved in implementing these methods must also be considered.

Regarding Mnasnet, Swap ReLU6 and median filters failed to converge at an acceptable accuracy level (i.e., the best-reached accuracy is 70.30% and 35.40%). At the same time, Adaptive Clipper and Ranger do not face any accuracy drop with respect to the baseline model. Furthermore, when Ranger and Adaptive Clipper are implemented, the EoI scores increase in complexity. Adaptive Clipper is the only HT requiring a training step that does not face any accuracy drop. In particular, as the original Mnasnet architecture involves 35 activation functions, Adaptive Clipper has increased complexity in the implementation to 3/5. From the training adaptation standpoint, the assigned score is 3/5 due to the same hyperparameter setup as the baseline model required during the training stage. Despite the EoI scores of 6/10 and 3/5, Adaptive Clipper and Ranger do not cause an accuracy drop, maintaining it at 83.00%. On the other hand, despite fine-grain TMR not facing an accuracy drop and not needing training adaptations, the implementation complexity is 5/5.

In MobileNet V2, the median filter reaches an EoI score of 10/10 due to the need for a custom operation and the extensive training process involved. In addition, this results in an accuracy of 54.50%.The limitation stems from the grouped convolutions inherent in the original MobileNet V2 architecture, which highlights challenges in causing the training algorithm to converge when integrating the median filter within the current architecture. On the other hand, both the Ranger and Adaptive Clipper methods achieved the same EoI scores as MnasNet. This can be attributed to the inclusion of 38 activation functions and a training process that closely resembles that of the baseline Mobilenet V2 during the implementation of Adaptive Clipper. After completing the training for Adaptive Clipper, the accuracy of MobileNet V2 only decreased from a baseline of 78.99% to 74.46%. Interestingly, despite a medium EoI score of 6/10, it still resulted in an accuracy increase of 5.37%. Additionally, fine-grain TMR does not lead to a drop in accuracy for MobileNet V2, and it received a complexity score of 5/5.

When Resnet18 architecture is considered, all HTs achieve an acceptable level of accuracy. The original architecture uses only eight activation functions, which leads to complexity scores of 2/5 for both Adaptive Clipper and Ranger. Although Adaptive Clipper requires retraining, it has a training adaptation score of 3/5 due to the similarities of the new training process to one of the original NN. Furthermore, Adaptive Clipper experiences a significant accuracy drop of 3.13%, while Ranger does not require any additional training steps, allowing it to maintain a stable accuracy of 85.72%. In contrast, when Swap ReLU6 is implemented on ResNet18, the neural network achieves an EoI score of 6/10 due to the interchange of batch normalization and ReLU layers. In addition to this high EoI score, the network experiences a considerable accuracy drop from 85.72% to 77.00%. Additionally,

implementing a median filter on ResNet18 results in an accuracy drop of 9.72%, along with an EOI score of 10/10.

Finally, all HTs impact the accuracy of the neural NN when implemented on SqueezeNet. Specifically, the accuracy increases by 5.00% compared to the baseline NN when using the Adaptive Clipper with a medium score of 6/10 for EoI. In contrast, the Ranger implementation avoids an additional training step and achieves the same accuracy as the baseline model, which is 77.50%. Similarly to Adaptive Clipper, the final EOI score for Ranger is 3/10. This score is attributed to the unchanged NN architecture after implementing Swap ReLU6, the limitation of the ReLU activation functions to a value of 6 (indicating a complexity of 2), and the short fine-tuning step (resulting in a training adaptation score of 1). Regarding the median filter, it demands significant effort to implement on SqueezeNet, leading to an EOI score of 10/10, but it only achieves an accuracy of 62.34%. Moreover, when fine-grain TMR is applied to SqueezeNet, the accuracy remains at 77.50%, consistent with the baseline architecture. This implementation has a complexity rating of 5/5 and a training adaptation degree of 0/5 since no additional training step is required.

5.1.2. HT Inference Time Overhead

While some hardening techniques demonstrate a certain degree of EoI and accuracy degradation, they may also result in an increased computation overhead, manifested as an increased inference execution time with respect to the NN architecture without HT incorporated. Thus, to extend the initial comparison of HTs during the study of the HT implementation impact, in Table 5, we report an estimation of the ITO based on the basic operations (i.e., comparison, addition, subtraction, multiplication, and division) that each HT requires when implemented on a specific NN architecture. In the first column, a weighted average of the absolute number of operations for the Baseline model is reported (i.e., the NN architecture without HT incorporated). The weighted average number of operations was estimated using equation Equation (5) considering the equivalent number of basic operations for every NN. Subsequently, the overhead with respect to the NN without HT implemented is computed using Equation (4). We can notice that Adaptive Clipper introduces a negligible overhead at inference time (i.e., less than 1%) for Lenet5, Resnet18, and Squeezenet, while for Mnasnet and Mobilenet V2, the overhead increases to 3.52% and 2.24%, respectively. Since Adaptive Clipper bounds the activation functions of the original NN architecture, the introduced overhead depends on the number of activation functions included in the original NN.

On the other hand, Ranger shows a higher overhead than Adaptive Clipper because it includes additional checks after every functional block. In fact, regardless of the NN, the overhead of Ranger increases by at least 2× with respect to the overhead introduced by Adaptive Clipper.

Table 5. Inference time overhead estimate based on the number of the additional MAC operations when the HT is implemented. In this table, the HT names are abbreviated as follows: Baseline (BL), Adaptive Clipper (AC), Swap ReLU6 (SR) median filter (MF).

NN Architecture	*Weighted Average Number of Operations*	*Inference Time Overhead*				
		AC	*R*	*SR*	*MF*	*Fine-Grain TMR*
Lenet5	97,709.45	0.29%	2.29%	0.29%	2.29%	200%
Mnasnet	1,007,184	3.52%	7.54%	-	-	200%
Mobilenet V2	1,787,803.62	2.24%	1.93%	2.24%	-	200%
Resnet18	6,992,621	0.30%	0.65%	0.30%	0.65%	200%
Squeezenet	1,776,322.87	0.88%	2.16%	0.88%	-	200%

In the case of Swap ReLU6, swapping the positions of batch normalization and ReLU activation functions does not add any overhead. However, the bounding introduced by the activation function results in an overhead similar to that caused by Adaptive Clipper.

Furthermore, the main idea behind implementing the median filter involves sorting a sliding window of length n. For all implementations of median filter, n is always set to 3, and the sorting operation is performed for each convolutional and linear layer. Consequently, the expected median filter is $\approx 9\times$ larger than Ranger HT overhead. The optimized implementation of the median operation, as proposed by the authors of [13], achieves a high degree of parallelization that reduces the overhead to approximately $3\times$ that of Ranger HT.

Finally, fine-grain TMR reaches a significant overhead (i.e., $\approx 200\%$) since it replicates each operation 3 times (neglecting the overhead due to the majority voting step as it would have introduced a negligible overhead with respect to the triple computation of the same operation).

More in general, our study indicates that range-restriction-based HTs, specifically Ranger and Adaptive Clipper, exhibit the best balance between implementation effort and NN accuracy. Regardless the NN where Ranger and Adaptive Clipper are implemented, they achieve scores of 2.4/10 and 5.4/10 in terms of EoI, with an average accuracy degradation of 1.48% and 0.00%, respectively. Notably, Swap ReLU also has a relatively low average EoI of 5.6/10. However, its implementation significantly impacts NN accuracy, resulting in an average drop of 6.66%. Nonetheless, the average overhead introduced by the implementation of Swap ReLU is 0.93%, whereas the overhead for Ranger and Adaptive Clipper is 1.44% and 2.91%, respectively.

5.2. Fault Mitigation with respect to WSBFs

Figure 5 allows us to compare the effectiveness of different HTs with respect to the WSBF error model. Figure 5 reports the classification of the WSBF according to their impact on the NN outcomes. The results indicate that due to the redundancy introduced by the fine-grain TMR (independent NN kernel copies), this strategy masks the impact of the injected WSBF. Therefore, we opted not to report the results of the corresponding FI campaigns for this case. In general, the experimental results show that according to the NN architecture on which the HT is implemented, the effectiveness of the hardening mechanism can quantitatively differ.

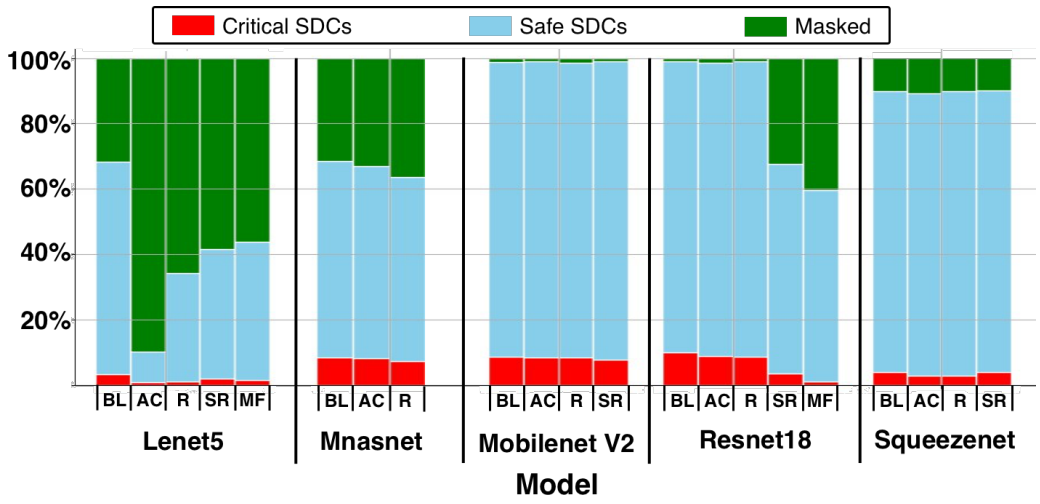


Figure 5. Fault class distribution. In this figure, the HT names are abbreviated as follows: Baseline (BL), Adaptive Clipper (AC), Swap ReLU6 (SR) and median filter (MF).

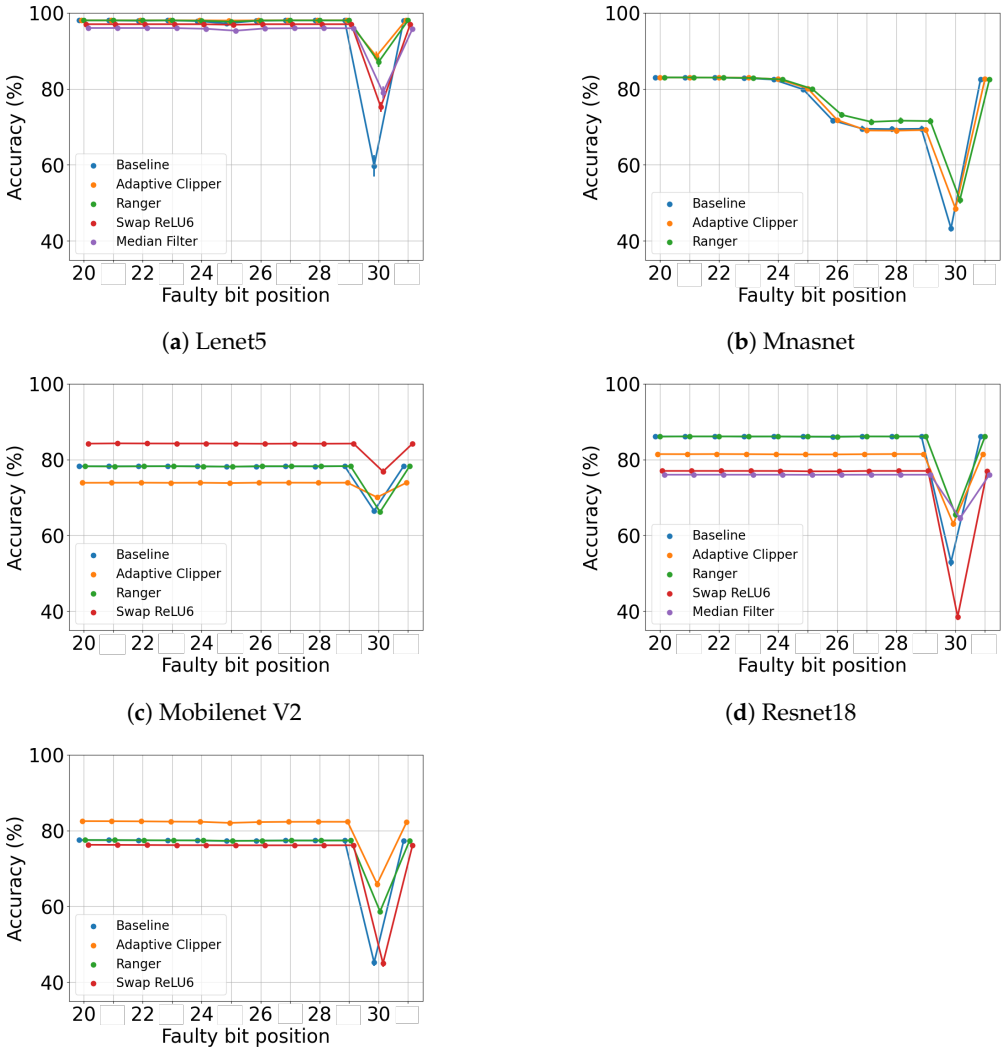
In Lenet5, Adaptive Clipper outperforms the other HTs, reducing the baseline critical SDC percentage from 3.31% to 0.85% by transforming most of the SDCs in masked faults. The significant increase of 58.23% in Masked faults leads to a decrease of 55.78% in safe SDCs as well. In contrast, the critical SDCs have been decreased to 0.97%, 1.84%, and 1.60% by Ranger, Swap ReLU6, and median filter.

A less pronounced enhancement in reliability can be observed when the NNs with a larger model size is considered. For example, in Mnaset, the critical SDC percentage is reduced by 1.17% with the implementation of Ranger with respect to the Baseline. A more sensible hardening effect can be noticed when Adaptive Clipper is implemented on Mnasnet by decreasing the number of critical SDCs by 0.5% with respect to the Baseline. The same weak beneficial effect is reflected by the sensible increase in masked faults (from a Baseline value of 31.41% to 36.31% and 32.95% for Ranger and Adaptive Clipper, respectively).

Similarly, Mobilenet V2 shows limited reliability enhancement effects; in fact, it achieves the highest critical SDC percentage with Swap ReLU6. The Baseline Mobilenet V2 architecture involves ReLU6 activation consisting of activation functions bounded to 6. Consequently, the result demonstrates the ineffectiveness of the step in the Swap ReLU6 on the Mobilenet V2 technique, in which activation functions and batch normalization layers are switched in position. The distribution of faults is very similar to the Baseline when Ranger and Adaptive Clipper are implemented, reporting a negligible decrease in the critical SDCs of 0.01% and 0.20%, respectively. The negligible reliability enhancement effect for Ranger and Adaptive Clipper is explained by the bounding to 6 of the Baseline NN architecture, which leads to computed bounds for the new activation functions that are constantly around 6 regardless of the NN residual block. On the other hand, Resnet18 shows better performance in terms of resiliency, specifically when Swap ReLU6 and median filter are implemented. In fact, a significant reduction in critical SDC percentage (9.87%in Baseline) is registered to 3.52% and 1.06% with Swap ReLU6 and median filter, respectively. The increase in Masked fault percentage is noticeable as well, going from 1.04% in Baseline to 32.38% for Swap ReLU6 and 40.33% for median filter. The results highlighted the importance of an additional training step when an HT is implemented on Resnet18, and it is not based on range restriction. However, the effect of Adaptive Clipper and Ranger is limited both in the percentage of critical SDCs (8.87% and 8.68%, respectively) and in masked faults (1.48% and 0.89%, respectively).

Our results indicate that the HTs implemented on Squeezenet show a negligible decrease in the critical SDC percentage (from 11.13% to, at most, 11.01% with Ranger).

Figure 6 reports further analysis at the bit level when using the WSBF model. This analysis helps to better visualize how the severity of soft errors is mitigated by HTs reporting the obtained accuracy for soft errors injected in different bit locations. Overall, this plot confirms the lack of clear patterns for ranking all HTs regardless of the NN architecture. However, we can notice that in 4 out of the 5 target NN architectures, range-restriction-based HTs (i.e., Ranger and Adaptive Clipper) outperform the others. For example, in Lenet5, Adaptive Clipper helps to reduce the impact of errors affecting the 30th bit by increasing the accuracy of the baseline model from 59.74% to 88.56%, while Ranger, median filter, and Swap ReLU6 introduce a reduced but significant improvement with respect to the Baseline in NN accuracy to 87.05%, 79.01%, 75.23%, respectively. This means that range-restriction-based HTs result in better improvements than other approaches, and the additional training step included in the implementation of Adaptive Clipper helps to improve Lenet5 resiliency more than Ranger.



(**a**) Lenet5 (**b**) Mnasnet

(**c**) Mobilenet V2 (**d**) Resnet18

(**e**) Squeezenet

Figure 6. Accuracy degradation for HTs in front of errors injected in weights per bit location.

Mnasnet, regardless of the implemented HT (i.e., Baseline, Ranger, or Adaptive Clipper), is the only architecture that is not resilient to the faults inducing errors in the exponent bits different than bit 30. However, Adaptive Clipper shows a negligible improvement with faults of 0.2%, while Ranger increases the accuracy with respect to the Baseline by up to 2.03%. However, when the fault location is in bit 30, Ranger and Adaptive Clipper increase the Baseline accuracy by 50.79% and 48.70%, respectively.

When considering Mobilenet V2 architecture, Adaptive Clipper improves the baseline accuracy from 66.48% to 70.03% when the bit-flip occurs on bit 30, while, under the same faulty scenario, Ranger shows performance comparable with the Baseline. As previously mentioned, this highlights the importance of the additional training step of Adaptive Clipper in Mobilenet V2, since the original architecture includes bounded activation functions so that the new activation functions are clipped around the original values. Interestingly, when Swap ReLU6 is implemented on Mobilenet V2, the training algorithm converges to higher accuracy (84.36%) than Baseline (78.99%). This results in an increased resiliency to bit-flips in bit 30, achieving an accuracy that is comparable to the Baseline accuracy when lower significant bits are affected, i.e., 76.84%.

For Resnet18, when the MSB is targeted, the baseline accuracy is 52.95%, and median filter reaches 64.47%, which is comparable to the improvements in Ranger and Adaptive Clipper where the NN reaches 65.52% and 63.05% accuracy, respectively. Ultimately, the

error-severity-based analysis shows the NN is sensitive to faults in exponent bit locations different from bit 30. In general, we can say that Ranger, from bit 26 to bit 30, shows a constant improvement of ≈2.5%. On the other hand, Adaptive Clipper increases the baseline accuracy from 43.31% to 48.51% only when bit 30 is targeted, while when other exponent bits are targeted, the recorded improvement is only 0.46%.

Concerning Squeezenet, Adaptive Clipper and Ranger, in particular Adaptive Clipper, outperform other approaches, reaching an accuracy score of 65.88% and 58.64%, where Swap ReLU6 seems to have no actual beneficial effect on the NN architecture, improving the baseline accuracy by only 0.1%. The ineffectiveness of the implementation of Swap ReLU6 on Squeezenet comes from the fact that its convolutional blocks do not contain any normalization layer, which means that implementing Swap ReLU6 on Squeezenet only concerns the bounding of the ReLU activation function to 6.

Based on this initial evaluation, we can conclude that when single bit-flips are introduced into the weights, it is not possible to establish a clear ranking in terms of the percentages of masked faults or critical SDCs. However, when analyzing the impact of the injected faults on the overall accuracy of the NN, the range-restriction-based HTs outperform the others in mitigating the effects of faults in the MSBs across all NN architectures, except for MobileNet V2. This suggests that despite the range-restriction-based HTs failing to convert a notable proportion of critical SDCs in Safe SDCs or masked faults, they are capable of mitigating the impact on the overall accuracy and reducing, though not eliminating, the number of misclassifications.

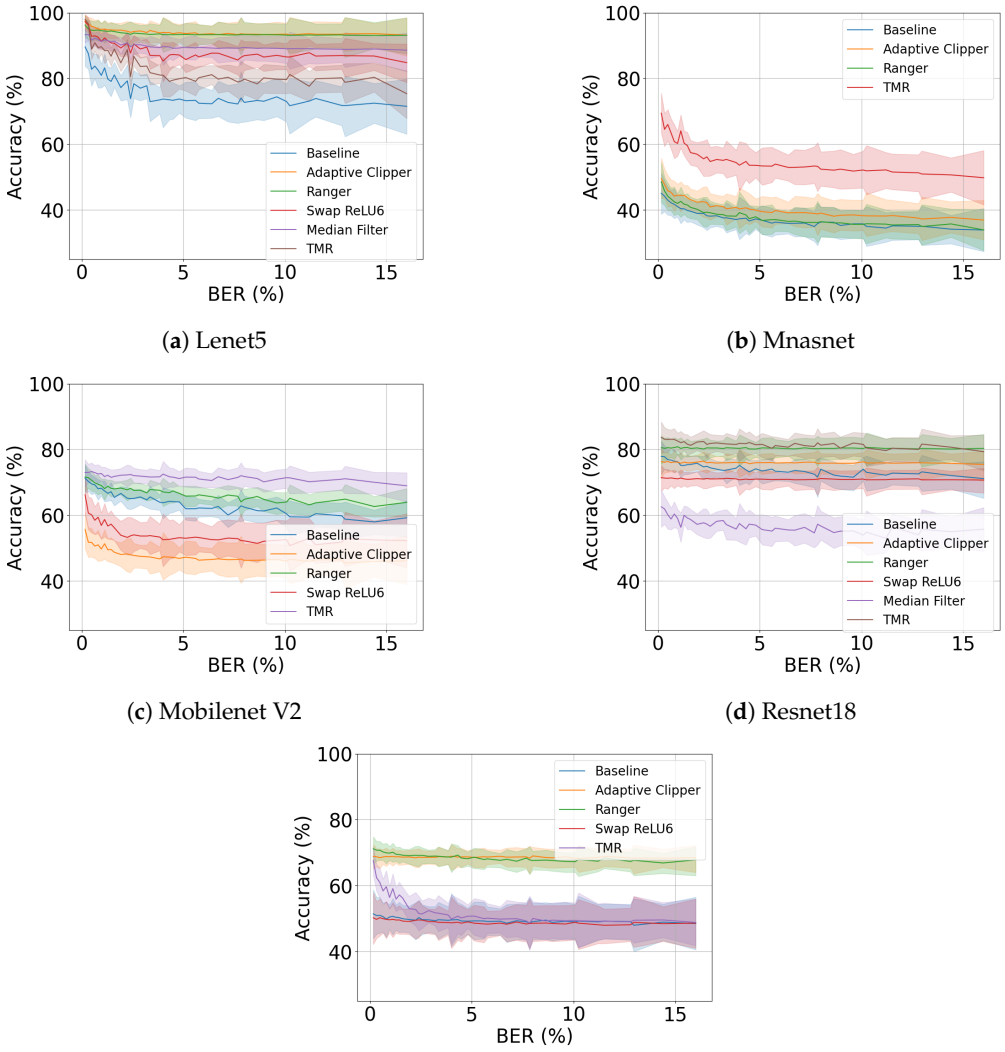
5.3. Fault Mitigation with respect to NBER

Figure 7 reports the analysis of the effect of the selected HTs when multiple bit-flips are performed in different locations of the intermediate output tensor according to the NBER model used in this paper. As mentioned in Section 3.1.2, the NBER determines that the intermediate FM is divided into patches (tiles) based on the TS, and the number of selected patches is selected based on the BlER. The number of target neurons among the selected patches is based on the NER. Thus, considering that the bit_location is fixed for each injection, $BER = NER \times BlER$. In Figure 7, the accuracy trend is reported with respect to the NBER. In general, we can notice that observing the behavior of HTs under NBER, the HTs' effectiveness, thus the final ranking, significantly depends on the NN architecture where they are implemented.

When the baseline LeNet architecture is affected by faults, the accuracy faces a significant drop from 89.60% to 71.50%. At the same time, the resiliency of the hardened version of Lenet is proven by the lower slope of the accuracy drop at increasing BER. For example, fine-grain TMR registers a decreasing trend with a similar slope as Baseline but going from 98% to 75.37%. When Ranger and Adaptive Clipper are implemented, Lenet5 experiences an accuracy drop of only 4.2% and 0.5% from the lowest (0.16%) to the highest BER (16%), respectively. When median filter and Swap ReLU6 are implemented on Lenet5, the NN is less impacted by the bit error rate (BER) FI reaching the highest BER, the accuracy scores of 84.80% and 88.61%, respectively.

Moving to Mnasnet, we can notice that fine-grain TMR outperforms the range-restriction-based HTs, keeping a constant increase in accuracy of up to 17.9% with respect to the Baseline regardless of the BER, which highlights the importance of introducing redundancy, even at the application level, in lightweight NNs. Concerning Adaptive Clipper, the reliability enhancement effect is more sensible, leading to an increase of 3.26% with respect to the baseline with the highest BER. On the other hand, Ranger is ineffective when it is implemented on Mnasnet due to the lower accuracy with respect to the Baseline. In fact, the Baseline can reach an accuracy higher than Ranger of up to 1%.

When Ranger is implemented on Mobilenet V2, it shows better reliability enhancement effectiveness with an average increase in accuracy by 2.7% with respect to the Baseline. Nonetheless, fine-grain TMR outperforms other HTs, reporting an accuracy of 72.96% and 68.93% when the lowest and highest error rates are injected, respectively. Interestingly, Swap ReLU6 and Adaptive Clipper reported the lowest accuracy scores (e.g., 52.23% and 46.07%, respectively, with the highest BER) due to the ineffectiveness of the additional training step that the implementation procedure of the HTs include.



(**a**) Lenet5 (**b**) Mnasnet

(**c**) Mobilenet V2 (**d**) Resnet18

(**e**) Squeezenet

Figure 7. Accuracy degradation for HTs when injecting errors in multiple FM bit locations.

Concerning Resnet18, the decreased slope of all curves with respect to the Baseline suggests that all HTs improve the resiliency to injected errors, except for median filter. However, the Baseline accuracy ranges from 77.89% (with BER = 0.16%) to 71.12% (with BER = 16%), and Baseline consistently outperforms Swap ReLU6, which reports an average accuracy of 71%. On the other hand, Adaptive Clipper, Ranger, and fine-grain TMR achieve an average accuracy of 75.98%, 80.39%, and 81.64%, respectively.

Finally, considering the Squeezenet architecture, Ranger and Adaptive Clipper are capable of limiting the NN accuracy degradation with respect to the fault-free case recording a drop of 9.7% for Ranger and 9.06% for Adaptive Clipper with BER = 16%. On the other hand, similarly to the Baseline, Swap ReLU6 reports an average accuracy of 48.98% and remains stable around that value regardless of the BER due to the lack of batch

normalization layers in the original Squeezenet architecture, which limits the Swap ReLU6 implementation.

To conclude, the results obtained with NBER FIs confirm that in 4 out of the 5 tested NN architectures, the range-restriction-based HTs increase the Baseline accuracy in front of the highest error rates (BER = 16%).

6. Discussion

Most of the prior studies utilize different and hardware-agnostic error models to evaluate the effectiveness of the proposed hardening techniques. Moreover, they employ different NN architectures and datasets, raising concerns on the applicability of the achieved results to other test benches and evaluation setups. In addition, the previous works evaluate the HTs, neglecting the effort required for the HT implementation.

Both the lack of a standardized approach for evaluating hardening techniques under hardware-aware error models and neglecting the effort required for the HT implementation limits the possibility to identify the optimal HT. The technique should find the best trade-off between the impact of its implementation on the normal functioning of the NN and the effectiveness in the NN reliability enhancement.

Our evaluation methodology allows us to assess the feasibility of implementing an HT on a specific NN and to evaluate the effectiveness of application-level HTs across various NN architectures and under hardware-aware error models. The employed hardware-aware error models (WSBF and NBER) simulate realistic fault conditions in GPU-based systems, providing a more accurate assessment of HT effectiveness.

We used our evaluation methodology to assess both the effort of implementation and the effectiveness in terms of reliability enhancement of five state-of-the-art hardening techniques. In Table 6, we summarize the most important results of our evaluations that come from both the HT impact assessment (considering the EoI and the ITO) and the reliability enhancement effectiveness (considering the RAD computed at the end of the WSBF FI when the bit-flip is performed on the bit 30 and at the end of the NBER FI campaign when the simulation is run at the highest bit error rate, i.e., 16%).

As a first result, we can notice that as a result of the training process of Swap ReLU6 when it is implemented on Mnasnet, as well as the training process of median filter when it is implemented on Mnasnet, Mobilenet V2 and Squeezenet, failed to converge. For this reason, the cells of the RAD corresponding to these NN architectures are empty. Specifically, median filter makes Mnasnet, Mobilenet, and Squeezenet converge at the accuracy scores of 35.40%, 54.50%, and 62.34%, respectively. These NNs involve convolutions with the so-called grouping strategy, an optimized version of the original convolution algorithm. However, the grouping strategy involves the processed tensors being shaped accordingly, which makes the implementation of the median filter at the application level almost impractical. On the other side, Mnasnet converged to an accuracy of 35.40% when Swap ReLU6 was implemented due to the higher variance in the distribution of the intermediate feature compared to the variance of the intermediate feature distributions of the original NN architecture (up to 10%).

Adaptive Clipper was initially designed for split computing NNs, but in this work, it has been adapted to general-purpose NNs. The article proposing Adaptive Clipper originally evaluated the technique against WSBF FI. When Adaptive Clipper is implemented on Lenet5, it achieved an EoI score of 4/10 and an ITO of 0.29% and demonstrated a significant resilience against WSBF, decreasing the RAD from 39.06% to 9.75%. Similarly, against NBER , Adaptive Clipper effectively improved Lenet5 resilience, leading to a reduced RAD of 21.69%. Although the EoI score of Adaptive Clipper remains limited (6/10), as well as the ITO (3.52%) when it is implemented on Mnasnet, it is not able to counteract the effect of

errors during NBER FI, leading to an increased RAD of 2.25%. On the other hand, the RAD is decreased by 6.27% against WSBF FI. Subsequently, Adaptive Clipper has a similar EoI score on Mobilenet V2 and on Resnet18 and Squeezenet (6/10, 5/10, and 6/10, respectively) but with different ITO (2.24%, 0.30%, and 0.88%, respectively) due to the higher number of activation functions in Mobilenet V2 architecture with respect to the others. However, Adaptive Clipper improves the reliability of all considered architectures by reducing the RAD by 9.24%, 15.31%, and 21.48%, respectively, against WSBF, and by 7.67%, 7.65%, and 29.48 respectively, against NBER.

Table 6. Overview of the achieved results per hardening technique for each neural network architecture. This table reports the results of the analysis conducted on the (i) HT implementation impact and (iii) the HT effectiveness in the faulty scenario. The HT implementation impact is evaluated with the EoI and ITO metrics. On the other hand, the reliability enhancement in terms of RAD is evaluated during WSBF and NBER FI campaigns. To directly see the effect of the HT in terms of reliability enhancement, the RAD of the NN *with (W)* the HT implemented is compared to the RAD of the NN *without (W/O)* the HT implemented. The empty cells correspond to the FI campaigns that were not performed due to the failed training process during the implementation phase. In addition, the red cells indicate an increase in RAD with respect to the tested NN when the HT is implemented with respect to the original NN architecture without the HT implemented. In this table, the HT names are abbreviated as follows: Baseline (BL), Adaptive Clipper (AC), Swap ReLU6 (SR), and median filter (MF).

HT	***NN***	***HT Impact***		***Faulty Scenario***			
		EoI	***ITO***	***WSBF***		***NBER***	
				W	***W/O***	***W***	***W/O***
AC	Lenet5	4/10	0.29%	9.75%	39.06%	6.80%	28.49%
	Mnasnet	6/10	3.52%	41.64%	47.91%	53.90%	51.75%
	Mobilenet V2	6/10	2.24%	7.44%	16.68%	16.68%	24.35%
	Resnet18	5/10	0.30%	24.22%	39.53%	11.27%	18.92%
	Squeezenet	6/10	0.88%	20.64%	42.12%	8.46%	37.94%
R	Lenet5	1/10	2.29%	11.22%	39.06%	6.86%	28.49%
	Mnasnet	3/10	7.54%	38.93%	47.91%	52.94%	51.75%
	Mobilenet V2	3/10	1.93%	16.63%	16.68%	20.26%	24.35%
	Resnet18	2/10	0.65%	24.76%	39.53%	9.78%	18.92%
	Squeezenet	3/10	2.16%	24.73%	42.12%	13.10%	37.94%
SR	Lenet5	7/10	0.29%	22.48%	39.06%	15.20%	28.49%
	Mnasnet	6/10	2.50%	-	-	-	-
	Mobilenet V2	6/10	2.24%	11.13%	16.68%	38.62%	24.35%
	Resnet18	6/10	0.30%	49.99%	39.53%	8.37%	18.92%
	Squeezenet	3/10	0.88%	41.55%	42.12%	36.77%	37.94%
MF	Lenet5	10/10	2.29%	17.72%	39.06%	7.75%	28.49%
	Mnasnet	10/10	11.7%	-	-	-	-
	Mobilenet V2	10/10	10.2%	-	-	-	-
	Resnet18	10/10	0.65%	15.46%	39.53%	26.82%	18.92%
	Squeezenet	10/10	3.4%	-	-	-	-
Fine-grain TMR	Lenet5	5/10	200%	0%	39.06%	23.09%	28.49%
	Mnasnet	5/10	200%	0%	47.91%	40.11%	51.75%
	Mobilenet V2	5/10	200%	0%	16.68%	11.85%	24.35%
	Resnet18	5/10	200%	0%	39.53%	11.89%	18.92%
	Squeezenet	5/10	200%	0%	42.12%	37.36%	37.94%

The paper that originally introduced Ranger tested this hardening technique on a wide selection of neural networks (eight architectures) and against the two error models that mimic the effect of transient faults. Their evaluations show the effectiveness of Ranger when it is implemented on all the selected NN architectures. Similarly, our evaluations, using hardware-aware error models, demonstrate the effectiveness of Ranger when it is

implemented on any of the considered NNs (five architectures). In general, Ranger always has an EoI score that never goes above 3/10, mainly due to the absence of the training step. At the same time, as Ranger introduces an overhead based on the total number of layers, it never has more than 3% of overhead except for Mnasnet, which is the NN with the highest number of layers, thus, according to the implementation of Ranger, being the NN with the highest overhead (7.54%) with respect to all NN architectures. Despite the low HT impact evaluated in terms of EoI and ITO, Ranger failed to enhance the reliability, scoring a slight decrease in RAD of 0.05% against WSBF and 4.09% against NBER. On the other hand, Ranger effectively enhances the reliability of Lenet5, Resnet18, and Squeezenet, decreasing the RAD of 27.84%, 14.77%, and 17.39% during WSBF and 21.63%, 9.14%, and 24.84% during NBER.

Swap ReLU was originally tested on one NN architecture (i.e., Resnet44) and trained and tested on two different datasets (i.e., Cifar10 and Cifar100) against the effect of transient faults, significantly reducing the critical SDC percentage. However, in our evaluations, despite the highest EoI with respect to the other NN architectures (7/10), Swap ReLU6 introduces a limited introduced overhead (0.29%) and a decrease in RAD of 16.58% during WSBF and of 13.29% during NBER. On the other hand, when Swap ReLU6 is implemented on Mobilenet V2, it introduced an increased overhead (2.29%) with a similar EoI (6/10), and during the WSBF, it reduced the impact on the accuracy by 5.55% with respect to the original NN architecture, but the RAD increased by 14.27% during NBER FI. Similarly, when Swap ReLU6 is implemented on Resnet18, it achieves 6/10 of the EoI score with an introduced ITO of 0.30%. Still, the RAD increased by 10.46% with respect to the NN architecture without HT implemented during WSBF FI while the RAD is decreased by 10.55% during the NBER FI. Indeed, suppose Swap ReLU is implemented on Squeezenet. In that case, it achieves 3/10 of the EoI score with 0.88% of the introduced overhead, and the new NN architecture shows a limited enhanced resiliency against WSBF and NBER of 0.6% and 1.17%, respectively.

Median filter and fine-grain TMR were initially evaluated on three NN architectures trained and tested on three datasets with application-level FI campaigns and a completely hardware-agnostic error model. The employed error model involves the injection of noise in neurons and weights in two different FI campaigns, while the third FI campaign simultaneously corrupts weights and neurons with random noise.

In general, median filter has a high EoI score of 10/10 and the highest introduced overhead on Mnasnet and Mobilenet V2 (of 11.7% and 10.2%), and it failed the convergence of the training algorithm in 3 out of the 5 tested NN architectures (Mnasnet, Mobilenet V2, and Squeezenet). Moreover, when it is implemented on Resnet18, the achieved RAD increases from 18.92% (without the HT implemented) to 26.82%.

Finally, although fine-grain TMR provides the highest resilience against WSBF with a RAD of 0% and with an average improvement of xx% of RAD with respect to the original NN architectures during the NBER FI, the introduced ITO is almost 200% regardless of the NN architecture, which makes the implementation of fine-grain TMR at the software level significantly inconvenient.

As a general result, it must be noted that although Adaptive Clipper, Ranger, and Swap ReLU6 have a tolerable effort of implementation, they never go above 6/10, 3/10, and 7/10, respectively. They fail to enhance the reliability of specific NN architectures against hardware-aware error models. For example, although Ranger achieved at most 3/10 in EoI when it was evaluated with NBER FI, it achieves 52.94% in RAD, while, without Ranger, Mnasnet achieves a better RAD of 51.75%. Similarly, Swap ReLU failed to improve the reliability for Resnet18 against WSBF FI, achieving 49.99% in RAD, while the NN without the hardening strategy achieved an RAD of only 39.53%.

These results support our hypotheses because they indicate that the effectiveness of the hardening technique strongly depends on the NN architecture and on the error model employed for the evaluations.

7. Conclusions

Given the lack of a standardized framework for evaluating the application-level HTs proposed in the literature, our study proposes an evaluation strategy that allows us to present the first comprehensive and independent assessment based on (i) the impact that the HT implementation has on the original NN architecture and (ii) the HTs effectiveness when subject to hardware-aware error models.

In our study, we conducted hardware-aware fault injection campaigns to systematically evaluate five prominent application-level HTs: Adaptive Clipper, Ranger, Swap ReLU6, median filter, and fine-grain TMR. We employed hardware-aware error models, i.e., WSBF and NBER to assess the effectiveness of these HTs in mitigating faults across a diverse range of NN architectures, including LeNet5, MnasNet, MobileNet V2, ResNet18, and SqueezeNet.

The experimental results show that implementing hardening techniques (HTs) requires significant effort, which can render them impractical. For example, applying Swap ReLU6 to MobileNet V2 with WSBF results in a modest accuracy drop to 78.99%, despite being superior to other HTs. Its implementation effort is rated at 6 out of 10, which may lead to a preference for alternative HTs.

In contrast, range-restriction-based HTs, such as Adaptive Clipper and Ranger, offer a better trade-off between implementation effort and performance, with Adaptive Clipper rated up to 6/10 for effort and a maximum accuracy degradation of 4.5%. Additionally, Adaptive Clipper incurs only 3.52% inference time overhead when implemented on Mnasnet, while Ranger shows a higher overhead of 7.54%. Furthermore, range-restriction-based HTs can effectively mask fault effects in over 58% of cases during hardware-aware fault injections, particularly when Adaptive Clipper is used on Lenet5 compared to the unhardened version.

In the future, we plan to expand our test bench to include more neural network architectures commonly used in Edge AI applications, such as split computing NNs, quantized NNs, and sparse NNs. Moreover, we will develop a new software-based hardening technique based on the results presented in this work.

Author Contributions: Conceptualization: G.E., J.-D.G.-B. and J.E.R.C.; methodology: G.E., J.-D.G.-B., J.E.R.C. and M.S.R.; software/hardware: G.E. and J.-D.G.-B.; validation, G.E., J.-D.G.-B. and J.E.R.C.; formal analysis: G.E., J.-D.G.-B., J.E.R.C. and M.S.R.; writing—original draft preparation: G.E., J.-D.G.-B.; writing—review and editing: G.E., J.-D.G.-B., J.E.R.C. and M.S.R. J.-D.G.-B., J.E.R.C. and M.S.R.; visualization: G.E., J.-D.G.-B., J.E.R.C. and M.S.R. All authors have read and agreed to the published version of the manuscript.

Funding: This work has been supported by the National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data and Quantum Computing. HPC CINECA provided the computational resources through the projects try24_limas and try24_guerrero from ISCRA project.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zhou, Z.; Chen, X.; Li, E.; Zeng, L.; Luo, K.; Zhang, J. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* **2019**, *107*, 1738–1762.

2. Bono, F.M.; Radicioni, L.; Cinquemani, S. A novel approach for quality control of automated production lines working under highly inconsistent conditions. *Eng. Appl. Artif. Intell.* **2023**, *122*, 106149.
3. Bourechak, A.; Zedadra, O.; Kouahla, M.N.; Guerrieri, A.; Seridi, H.; Fortino, G. At the confluence of artificial intelligence and edge computing in iot-based applications: A review and new perspectives. *Sensors* **2023**, *23*, 1639.
4. Matsubara, Y.; Levorato, M.; Restuccia, F. Split computing and early exiting for deep learning applications: Survey and research challenges. *Acm Comput. Surv.* **2022**, *55*, 1–30.
5. Dally, B. Hardware for deep learning. In Proceedings of the 2023 IEEE Hot Chips 35 Symposium (HCS), IEEE Computer Society, Palo Alto, CA, USA, 27–29 August 2023; pp. 1–58.
6. Strojwas, A.J.; Doong, K.; Ciplickas, D. Yield and Reliability Challenges at 7nm and Below. In Proceedings of the 2019 Electron Devices Technology and Manufacturing Conference (EDTM), Singapore, 12–15 March 2019; pp. 179–181.
7. Hill, I.; Chanawala, P.; Singh, R.; Sheikholeslam, S.A.; Ivanov, A. CMOS reliability from past to future: A survey of requirements, trends, and prediction methods. *IEEE Trans. Device Mater. Reliab.* **2021**, *22*, 1–18.
8. IEEE. The international roadmap for devices and systems: 2022. In Proceedings of the Institute of Electrical and Electronics Engineers (IEEE), Chiang Mai, Thailand, 25–27 February 2022.
9. Hochschild, P.H.; Turner, P.; Mogul, J.C.; Govindaraju, R.; Ranganathan, P.; Culler, D.E.; Vahdat, A. Cores that don't count. In Proceedings of the Workshop on Hot Topics in Operating Systems, Ann Arbor, MI, USA, 31 May 31–2 June 2021.
10. Dixit, H.D.; Pendharkar, S.; Beadon, M.; Mason, C.; Chakravarthy, T.; Muthiah, B.; Sankar, S. Silent data corruptions at scale. *arXiv* **2021**, arXiv:2102.11245.
11. Rech, P. Artificial neural networks for space and safety-critical applications: Reliability issues and potential solutions. *IEEE Trans. Nucl. Sci.* **2024**, *71*, 377–404.
12. Amazon. How Amazon is Building Its Drone Delivery System. Published in 2022. Available online: https://www.aboutamazon.com/news/transportation/how-amazon-is-building-its-drone-delivery-system (accessed on 18 October 2024).
13. Ozen, E.; Orailoglu, A. Boosting bit-error resilience of DNN accelerators through median feature selection. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2020**, *39*, 3250–3262.
14. Chen, Z.; Li, G.; Pattabiraman, K. A low-cost fault corrector for deep neural networks through range restriction. In Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Taipei, Taiwan, 21–24 June 2021; pp. 1–13.
15. Mahmoud, A.; Hari, S.K.S.; Fletcher, C.W.; Adve, S.V.; Sakr, C.; Shanbhag, N.; Molchanov, P.; Sullivan, M.B.; Tsai, T.; Keckler, S.W. Hardnn: Feature map vulnerability evaluation in cnns. *arXiv* **2020**, arXiv:2002.09786.
16. Zhan, J.; Sun, R.; Jiang, W.; Jiang, Y.; Yin, X.; Zhuo, C. Improving fault tolerance for reliable DNN using boundary-aware activation. *IEEE Trans.-Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *41*, 3414–3425.
17. Ruospo, A.; Gavarini, G.; Bragaglia, I.; Traiola, M.; Bosio, A.; Sanchez, E. Selective hardening of critical neurons in deep neural networks. In Proceedings of the 2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS), Prague, Czech Republic, 6–8 April 2022; pp. 136–141.
18. Hacene, G.B.; Leduc-Primeau, F.; Soussia, A.B.; Gripon, V.; Gagnon, F. Training modern deep neural networks for memory-fault robustness. In Proceedings of the 2019 IEEE International Symposium on Circuits and Systems (ISCAS), Sapporo, Japan, 26–29 May 2019; pp. 1–5.
19. Reagen, B.; Whatmough, P.; Adolf, R.; Rama, S.; Lee, H.; Lee, S.K.; Hernández-Lobato, J.M.; Wei, G.Y.; Brooks, D. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. *Acm Sigarch Comput. Archit. News* **2016**, *44*, 267–278.
20. Pessia, F.; Guerrero-Balaguera, J.D.; Sierra, R.L.; Condia, J.E.R.; Levorato, M.; Reorda, M.S. Effective Application-level Error Modeling of Permanent Faults on AI Accelerators. In Proceedings of the IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS), Rennes, France, 3–5 July 2024.
21. Guerrero-Balaguera, J.D.; Rodriguez Condia, J.E.; Levorato, M.; Sonza Reorda, M. Evaluating the Reliability of Supervised Compression for Split Computing. In Proceedings of the IEEE VLSI Test Symposium 2024 Proceedings, Tempe, AZ, USA, 22–24 April 2024.
22. dos Santos, F.F.; Pimenta, P.F.; Lunardi, C.; Draghetti, L.; Carro, L.; Kaeli, D.; Rech, P. Analyzing and increasing the reliability of convolutional neural networks on GPUs. *IEEE Trans. Reliab.* **2018**, *68*, 663–677.
23. dos Santos, F.F.; Hari, S.K.S.; Basso, P.M.; Carro, L.; Rech, P. Demystifying GPU reliability: Comparing and combining beam experiments, fault simulation, and profiling. In Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Virtual, 17–21 May 2021; pp. 289–298.
24. Dos Santos, F.F.; Kritikakou, A.; Condia, J.E.R.; Guerrero-Balaguera, J.D.; Reorda, M.S.; Sentieys, O.; Rech, P. Characterizing a neutron-induced fault model for deep neural networks. *IEEE Trans. Nucl. Sci.* **2022**, *70*, 370–380.
25. Condia, J.E.R.; Guerrero-Balaguera, J.D.; Dos Santos, F.F.; Reorda, M.S.; Rech, P. A multi-level approach to evaluate the impact of GPU permanent faults on CNN's reliability. In Proceedings of the 2022 IEEE International Test Conference (ITC), Anaheim, CA, USA, 23–30 September 2022; pp. 278–287.

26. Guerrero-Balaguera, J.D.; Condia, J.E.R.; Levorato, M.; Reorda, M.S. Evaluating the Reliability of Supervised Compression for Split Computing. In Proceedings of the 2024 IEEE 42nd VLSI Test Symposium (VTS), Tempe, AZ, USA, 22–24 April 2024; pp. 1–6.
27. Guerrero Balaguera, J.D.; Rodriguez Condia, J.E.; Sonza Reorda, M. Effective Fault Effects Evaluation for Permanent Faults in GPUs executing DNNs. *Acm Trans. Des. Autom. Electron. Syst.* **2025**.
28. Tsai, T.; Hari, S.K.S.; Sullivan, M.; Villa, O.; Keckler, S.W. Nvbitfi: Dynamic fault injection for gpus. In Proceedings of the 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Taipei, Taiwan, 21–24 June 2021, pp. 284–291.
29. Hari, S.K.S.; Tsai, T.; Stephenson, M.; Keckler, S.W.; Emer, J. SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation. In Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Santa Rosa, CA, USA, 24–25 April 2017; pp. 249–258.
30. Guerrero-Balaguera, J.D.; Rodriguez Condia, J.E.; F. dos Santos, F.; Sonza Reorda, M.; Rech, P. Understanding the Effects of Permanent Faults in GPU's Parallelism Management and Control Units. *arXiv* **2023**, arXiv:2306.10856.
31. Bosio, A.; Bernardi, P.; Ruospo, A.; Sanchez, E. A reliability analysis of a deep neural network. In Proceedings of the 2019 IEEE Latin American Test Symposium (LATS), Santiago, Chile, 11–13 March 2019; pp. 1–6.
32. Ruospo, A.; Gavarini, G.; De Sio, C.; Guerrero, J.; Sterpone, L.; Reorda, M.S.; Sanchez, E.; Mariani, R.; Aribido, J.; Athavale, J. Assessing convolutional neural networks reliability through statistical fault injections. In Proceedings of the 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 17–19 April 2023; pp. 1–6.
33. Esposito, G.; Guerrero-Balaguera, J.D.; Condia, J.E.R.; Levorato, M.; Reorda, M.S. Enhancing the Reliability of Split Computing Deep Neural Networks. In Proceedings of the 2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design (IOLTS), Rennes, France, 3–5 July 2024; pp. 1–7.
34. Cavagnero, N.; Dos Santos, F.; Ciccone, M.; Averta, G.; Tommasi, T.; Rech, P. Transient-fault-aware design and training to enhance dnns reliability with zero-overhead. In Proceedings of the 2022 IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS), Torino, Italy, 12–14 September 2022; pp. 1–7.
35. LeCun, Y. Lenet5 Implementation. Published in 1998. Available online: https://github.com/lychengrex/LeNet-5-Implementation-Using-Pytorch/blob/master/LeNet-5%20Implementation%20Using%20Pytorch.ipynb (accessed on 3 April 2023).
36. Tan, M.; Chen, B.; Pang, R.; Vasudevan, V.; Sandler, M.; Howard, A.; Le, Q.V. Mnasnet: Platform-aware neural architecture search for mobile. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, Long Beach, CA, USA, 15–20 June 2019; pp. 2820–2828.
37. Sandler, M.; Howard, A.; Zhu, M.; Zhmoginov, A.; Chen, L.C. Mobilenetv2: Inverted residuals and linear bottlenecks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 4510–4520.
38. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
39. Iandola, F.N. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360.
40. LeCun, Y.; Cortes, C.; Burges, J.C.C. MNIST Dataset. Published in 1998. Available online: https://yann.lecun.com/exdb/mnist/ (accessed on 3 April 2023).
41. Krizhevsky, A. Cifar10 Dataset. Published in 2009. Available online: https://www.cs.toronto.edu/~kriz/cifar.html (accessed on 3 June 2023).
42. Coates, A.; Lee, Honglak; Y., Andrew Available online: March 5th 2025 https://cs.stanford.edu/~acoates/stl10/.
43. Klambauer, G.; Unterthiner, T.; Mayr, A.; Hochreiter, S. Self-normalizing neural networks. *Adv. Neural Inf. Process. Syst.* **2017**, *10*.
44. Hendrycks, D.; Gimpel, K. Gaussian error linear units (gelus). *arXiv* **2016**, arXiv:1606.08415.
45. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016; Available online: March 5th 2025 http://www.deeplearningbook.org.
46. Bottou, L. Large-scale machine learning with stochastic gradient descent. In Proceedings of the COMPSTAT'2010: 19th International Conference on Computational Statistics, Paris, France, 22–27 August 2010; Keynote, Invited and Contributed Papers; Springer: Berlin/Heidelberg, Germany, 2010; pp. 177–186.
47. Chikkagoudar, S.; Wang, K.; Li, M. GENIE: a software package for gene-gene interaction analysis in genetic association studies using multiple GPU or CPU cores. *Bmc Res. Notes* **2011**, *4*, 1–7.
48. Huang, J.; Yu, C.D.; van de Geijn, R.A. Implementing Strassen's algorithm with CUTLASS on NVIDIA Volta GPUs. *arXiv* **2018**, arXiv:1808.07984.
49. Singh, A.; et al. Silent data errors: Sources, detection, and modeling. In Proceedings of the IEEE 41st VLSI Test Symp. (VTS'23), San Diego, CA, USA, 24–26 April 2023.

50. Li, W.; Ning, X.; Ge, G.; Chen, X.; Wang, Y.; Yang, H. FTT-NAS: Discovering fault-tolerant neural architecture. In Proceedings of the 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), Beijing, China, 13–16 January 2020; pp. 211–216.
51. Wang, J.; Zhu, J.; Fu, X.; Zang, D.; Li, K.; Zhang, W. Enhancing Neural Network Reliability: Insights From Hardware/Software Collaboration With Neuron Vulnerability Quantization. *IEEE Trans. Comput.* **2024; pp. 1953–1966; Vol 73**.
52. Peterson, W. Error-correcting codes. *Cambridge, Ma: Mit Press Google Sch.* **1972**, *2*, 208–213.
53. Zhang, J.J.; Gu, T.; Basu, K.; Garg, S. Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In Proceedings of the 2018 IEEE 36th VLSI Test Symposium (VTS), San Francisco, CA, USA, 22–25 April 2018; pp. 1–6.
54. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149.
55. Ozen, E.; Orailoglu, A. Just say zero: Containing critical bit-error propagation in deep neural networks with anomalous feature suppression. In Proceedings of the 39th International Conference on Computer-Aided Design, Storrs, CT, USA, 24–27 Octorber 2020, pp. 1–9.
56. Correia, M.; Ferro, D.G.; Junqueira, F.P.; Serafini, M. Practical Hardening of {Crash-Tolerant} Systems. In Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12), Boston, MA, USA, 13–15 June 2012; pp. 453–466.
57. Lyons, R.E.; Vanderkulk, W. The use of triple-modular redundancy to improve computer reliability. *Ibm J. Res. Dev.* **1962**, *6*, 200–209.
58. Solouki, M.A.; Angizi, S.; Violante, M. Dependability in Embedded Systems: A Survey of Fault Tolerance Methods and Software-Based Mitigation Techniques. *IEEE Access* **2024**, *12*, 180939–180967.
59. Hoang, L.H.; Hanif, M.A.; Shafique, M. Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. In Proceedings of the 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 9–13 March 2020; pp. 1241–1246.
60. Wei, N.; Yang, S.; Tong, S. A modified learning algorithm for improving the fault tolerance of BP networks. In Proceedings of the International Conference on Neural Networks (ICNN'96), Washington, DC, USA, 3–6 June 1996; Volume 1, pp. 247–252.
61. Batcher, K.E. Sorting networks and their applications. In Proceedings of the Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April–2 May 1968; pp. 307–314.
62. Tyagi, A.; Gan, Y.; Liu, S.; Yu, B.; Whatmough, P.; Zhu, Y. Thales: Formulating and estimating architectural vulnerability factors for dnn accelerators. *arXiv* **2022**, arXiv:2212.02649.
63. Xue, X.; Liu, C.; Wang, Y.; Yang, B.; Luo, T.; Zhang, L.; Li, H.; Li, X. Soft error reliability analysis of vision transformers. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2023**, *31*, 2126–2136.
64. Geissler, F.; Qutub, S.; Roychowdhury, S.; Asgari, A.; Peng, Y.; Dhamasia, A.; Graefe, R.; Pattabiraman, K.; Paulitsch, M. Towards a safety case for hardware fault tolerance in convolutional neural networks using activation range supervision. *arXiv* **2021**, arXiv:2108.07019.
65. Yan, Z.; Shi, Y.; Liao, W.; Hashimoto, M.; Zhou, X.; Zhuo, C. When single event upset meets deep neural networks: Observations, explorations, and remedies. In Proceedings of the 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), Beijing China, 13–16 January 2020; pp. 163–168.
66. Cho, H.; Mirkhani, S.; Cher, C.Y.; Abraham, J.A.; Mitra, S. Quantitative evaluation of soft error injection techniques for robust system design. In Proceedings of the 50th Annual Design Automation Conference, Austin, TX, USA, 29 May–7 June 2013; pp. 1–10.
67. Leveugle, R.; Calvez, A.; Maistri, P.; Vanhauwaert, P. Statistical fault injection: Quantified error and confidence. In Proceedings of the 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, France, 20–24 April 2009; pp. 502–506.
68. Li, G.; Hari, S.K.S.; Sullivan, M.; Tsai, T.; Pattabiraman, K.; Emer, J.; Keckler, S.W. Understanding error propagation in deep learning neural network (DNN) accelerators and applications. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 12–17 November 2017; pp. 1–12.
69. Hong, S.; Frigo, P.; Kaya, Y.; Giuffrida, C.; Dumitraș, T. Terminal brain damage: Exposing the graceless degradation in deep neural networks under hardware fault attacks. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; pp. 497–514.
70. Esposito, G.; Guerrero-Balaguera, J.D.; Condia, J.E.R.; Reorda, M.S. Evaluating Different Fault Injection Abstractions on the Assessment of DNN SW Hardening Strategies. *arXiv* **2024**, arXiv:2412.08466.