Düzce University Journal of Science and Technology
Düzce Üniversitesi Bilim ve Teknoloji Dergisi

14 (2), 2026, 537-550.
https://doi.org/10.29130/dubited.1793166

e-ISSN: 2148-2446
https://dergipark.org.tr/en/pub/dubited

Research Article

Evaluating the Fault Tolerance and Vulnerability of Artificial Neural Networks Under Hardware Errors

Hatice Aktaş Aydın[1*], Albert Nioroge Kahira[2], Gülay Yalçın[3], Osman Sabri Ünsal[4]

[1] Sivas University of Science and Technology Faculty of Engineering and Natural Science Department of Computer Engineering, Sivas, Türkiye.
ror.org/05s32j989

[2] AstraZeneca, Barcelona, Spain.
ror.org/05qqrnb63

[3] Abdullah Gül University Faculty of Engineering and Natural Science Department of Computer Engineering, Kayseri, Türkiye.
ror.org/00zdyy359

[4] Barcelona Supercomputing Center, Barcelona, Spain.
ror.org/05sd8tv96

*Corresponding author:
haticeaktas@sivas.edu.tr

Received: 29.09.2025
Revision: 24.02.2026
Accepted: 24.02.2026
Online: 19.04.2026

Abstract: Artificial Neural Networks (ANN) have gained popularity again due to the increasing interest and developments in artificial intelligence, as well as the increased computational power offered by High Performance Computing (HPC) systems. Since neural network applications are used in large data centers and HPC systems, they face similar reliability issues such as bit slippage in registers and memory structures that are common in these systems. Therefore, they require special robustness and protection mechanisms that can significantly increase the system cost. However, understanding the impact of hardware failures on different components of ANN applications can help determine which parts are more vulnerable and require higher reliability. In this study, the effects of hardware faults on ANN applications when they are run in HPC systems and large-scale data centers are evaluated, and thus, the reliability costs are aimed to be reduced. Fault injection experiments performed with traditional techniques can be quite time-consuming for ANN applications. Therefore, a method is presented to reduce the fault injection time in such applications. When we evaluate the effects of hardware faults on Artificial Neural Network (ANN) applications running on CPU-based (Intel Xeon) and GPU-based (NVIDIA V100) high-performance computing (HPC) systems, our results show that ANNs are vulnerable to some hardware faults, especially those occurring in certain layers and architectural registers.

Keywords: *Fault tolerance, Reliability, Machine learning, Artificial neural networks, Artificial intelligence*

I. INTRODUCTION

Artificial Intelligence (AI) has become a fundamental technology in many fields such as healthcare, finance, transportation, agriculture, and scientific research, and its impact is rapidly increasing with the rise in data access, advancements in algorithms, and improvements in computational infrastructure. Today, AI-based systems are widely used in solving complex problems such as image and video analysis, speech recognition, natural language processing, anomaly detection, and decision support systems. Artificial Neural Networks (ANNs), one of the fundamental building blocks of these systems, play a significant role in modeling nonlinear and high-dimensional problems thanks to their structures inspired by biological nervous systems. In recent years, traditional ANN architectures have largely been extended or replaced by Deep Neural Networks (DNNs) containing numerous hidden layers. In particular, Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and transformer-based architectures can automatically learn hierarchical and high-level representations through data-driven training processes performed on large-scale datasets, and offer cutting-edge performance in visual and language-based tasks (Bengio et al., 2021). These developments have enabled modern AI systems to make significant progress in both accuracy and generalization capabilities. While ANN and DNN models are mostly developed within software frameworks, their training and inference are increasingly supported by GPUs and similar hardware accelerators. Especially when dealing with large datasets and models with a high number of parameters, training processes can require intensive computational processes, lasting from hours to weeks. This has made deep learning workloads a natural application area for High Performance Computing (HPC) systems and large-scale cloud infrastructures. Furthermore, future AI systems are expected to be trained and run on a combination of HPC platforms, cloud data centers, and edge devices, depending on application requirements (Borkar & Chien, 2011). One of the most fundamental challenges encountered in HPC systems is system reliability, as the massive scale, wide physical distribution, and high component count of modern HPC platforms significantly increase vulnerability to both transient and permanent hardware failures. Transient errors are single-bit corruptions that typically occur in processor components. For example, in critical subunits such as register files and cache structures, as a result of environmental influences such as radiation, temperature changes, or power surges. Permanent errors are associated with irreversible physical deterioration of semiconductor hardware components over time. Unfortunately, the continuous downsizing of semiconductor technologies and the increase in the number of processor cores increase the likelihood of both transient and permanent hardware errors, making these errors a persistent threat to HPC systems (Cappello et al., 2009; Di Martino et al., 2015).

Cite as: Aktaş Aydın, H., Kahira, A. N., Yalçın, G., & Ünsal, O. S. (2026). Evaluating the fault tolerance and vulnerability of artificial neural networks under hardware errors. *Düzce University Journal of Science and Technology, 14*(2), 537-550. https://www.doi.org/10.29130/dubited.1793166

This is an open access paper distributed under the terms and conditions of the Creative Commons Attribution-NonCommercial 4.0 International License.

Although the majority of High Performance Computing (HPC) systems consist of CPU-based architectures, the use of GPU-based systems has become increasingly widespread in recent years. However, it has been reported in the literature that approximately twice as many hardware failures are observed in GPU systems compared to CPU systems (Tiwari et al., 2015). The main reason for this difference is that GPUs were initially designed for low-error-sensitivity applications such as video processing and imaging. Therefore, advanced fault tolerance mechanisms such as Chipkill, which are widely used in classical systems, were not initially included in GPU architectures. After GPUs began to be used in critical application areas such as scientific computing and machine learning, basic fault tolerance measures such as simple Corrective Coding (ECC) began to be integrated. However, these basic mechanisms provide limited protection and are insufficient in terms of reliability when compared to advanced fault tolerance techniques such as Chipkill.

Hardware-related errors can have different levels of impact on the output of a running application. For example, a single bit corruption in a critical record can cause the entire application to crash unexpectedly, while an unobserved bit corruption can produce an incorrect but system-valid result in the output of the application. Such errors are called "silent data corruption" (SDC) in the literature. For example, an ANN used in a task such as handwritten character recognition may classify a written letter "A" as "B" due to a single bit corruption. Such corruptions can not only affect the output accuracy, but also extend the application's runtime or cause the system to freeze, which causes inefficient use of limited and valuable resources in HPC environments. In order to develop tolerance against hardware faults in HPC systems, it is necessary to integrate strong fault detection and prevention mechanisms. However, the implementation of such mechanisms can create significant additional burdens on system performance, energy efficiency, and hardware costs. Therefore, it is not practical to make all system components completely fault-free. Instead, it is necessary to identify the most vulnerable components in the system and to establish an optimal balance between reliability and performance. In particular, determining the degree of impact of faults on application output and the most critical fault locations is of great importance in minimizing reliability costs. This information allows for more efficient and targeted design of fault detection and correction mechanisms. Fault injection is an experimental method used to evaluate the robustness of systems or vulnerability of system components. It is based on the injection of artificial errors into the system by randomly changing the values of certain bits (Bautista Gomez & Cappello, 2015) and it provides high accuracy in analyzing system reliability. However, it is computationally expensive in terms of both time and energy, since evaluating fault effects requires injecting a large number of random faults over multiple executions of the same application. Due to the inherently long execution times of ANN and DNN workloads, the cumulative overhead of repeated executions significantly escalates, posing a major challenge for fault injection-based reliability evaluation. The fact that the training processes of ANNs in particular take a long time causes fault injection-based analyses to become more laborious in terms of time and resources.

The main purpose of this study is to comprehensively analyze the effects of hardware-related errors on Artificial Neural Network (ANN) applications using a time-efficient fault injection method. To this end, to reduce the high time cost due to repeated execution of the training process, errors are directly injected into the weight matrices between layers. This approach is based on the assumption that errors that occur during training or in any layer but do not directly affect the weight matrix are benign in terms of system behavior. For the comprehensive analyze, both CPU and GPU based High Performance Computing (HPC) systems were considered within the scope of reliability assessment and fault injection experiments were performed on both platforms. In addition, the general fault tolerance of Deep Neural Networks (DNN), both Multi-Layer Perceptron (MLP) and specifically Convolutional Neural Network (CNN) architectures, is investigated. Considering that ANN applications are one of the fundamental building blocks of artificial intelligence systems and that these applications have a high potential to be exposed to hardware reliability problems, it was thought that the subject should be approached in depth. In addition, the resilience of the ANN architecture against hardware faults is analyzed comparatively with the SPEC CINT2006 benchmark set consisting of seven integer-based applications.

In this context, first a simple ANN model consisting of input, hidden, and output layers, then a full-scale fault injection analysis was performed on a more advanced DNN architecture representing current application trends. A total of three different fault injection scenarios were applied. In the first scenario, system behaviors are investigated by injecting faults into the processor's architectural registers during the training and testing phases. In the second scenario, bit corruptions are directly applied to the weight matrices of the neural network in order to evaluate the effect of faults on different layers and processing phases. In the third scenario, fault injections are performed on GPU cores by developing a CUDA-based version of the ANN application.

According to the findings obtained from the ANN robustness analysis conducted. It was observed that hardware errors applied to the weight matrix between the hidden and output layers in a three-layer neural network architecture (input, hidden, output) caused much more significant effects. In contrast, errors occurring between the input and hidden layers had a smaller impact. It was determined that different architectural registers used in x86-based HPC systems exhibited different sensitivities in terms of ANN applications. As an example, while error injections into the RSP register almost always caused a system crash, registers such as RDX and RBP mostly resulted in silent data corruption (SDC). It was determined that some registers caused the application to take longer than usual or not be completed at all. It was also revealed that the probability of an error occurring at the microarchitectural level causing misclassification varies depending on the input data of the application (e.g., the digit type in the MNIST example). ANN implementations exhibited lower fault tolerance compared to SPEC CINT2006 benchmark implementations. In the scenarios running on GPU, a large

number of errors were recorded by the system as “uncorrected errors” and in most cases, these errors were directly reported by the Linux operating system.

The contributions of this study are as follows:

- A comprehensive reliability assessment of ANN-based implementations is presented, covering the reliability characteristics of ANN and DNN models—including MLP and CNN architectures—as well as non-neural workloads represented by SPECint benchmarks
- A new fault injection approach that significantly reduces the fault injection time is presented in which faults are injected directly to weight matrices between layers.
- The effects of errors on different hardware components such as architectural registers, ANN layers, input data, and processor architectures (CPU/GPU) are presented.
- Potential design choices to increase the reliability of ANNs with low cost are discussed.

The remaining sections of the paper are organized as follows: Section 2 presents previous studies on fault tolerance in ANN and HPC systems. Section 3 describes the fault injection environment and the experiments performed in detail. Section 4 presents and discusses the obtained results. Finally, Section 5 concludes the work.

II. BACKGROUND AND RELATED WORK

In the ANN domain, fault tolerance is the ability of a system to continue to perform to specification in the presence of faults, such as broken connections, connections with erroneous weights or neurons with inaccurate outputs (Kulakov et al., 2015). ANNs are generally considered fault-tolerant due to their parallel structures (Su et al., 2016). They will perform as expected in the presence of errors as their parallel nature offers some redundancy. This argument has been the basis and justification for ANN usage in mission-critical applications or applications where no repair mechanism is available such as space exploration (Tchernev et al., 2005). Nevertheless, as previously mentioned (Piuri, 2001), neural networks cannot be considered completely fault-tolerant. Lacking a built-in fault tolerance mechanism, artificial neural networks can be severely affected by hardware or software-related errors. Much of the research on the fault tolerance of artificial neural networks has focused on errors occurring within the neural network structure at the application level. Some studies also examine errors occurring in hardware-based implementations of artificial neural network applications. Despite the extensive literature on the fault tolerance of artificial neural networks, the impact of hardware errors on such networks, especially in the context of high-performance computing (HPC) systems, has not been adequately addressed. Ruospo et al., (2023) conducted extensive error injection campaigns to evaluate the reliability of CNNs. The study examined the sensitivity of weights, which are static parameters, to hardware errors and showed that the error tolerance can vary significantly for different network layers. The results suggest that architecture-specific resilience strategies need to be developed. Alobaid et al., (2025) presented a comprehensive systematic review of disruptive attacks on artificial neural networks. In this study, various fault injection techniques, especially bit-flip attacks, their effects, and existing detection/protection mechanisms were analyzed in detail. The study reveals vulnerabilities that threaten the reliability of ANNs and suggests important research directions for future defense strategies. Wang et al., (2024) proposed a contrastive learning-based approach targeting detection and recovery mechanisms against fault injection attacks. The developed method aims to improve the system's ability to recover by detecting anomalies in network behavior after fault injection. This approach is noteworthy for demonstrating the effectiveness of learning-based solutions to improve post-attack resilience. Vinck et al., (2025) proposed a fault-aware training approach against multiple SEU errors that occur during DNN inference. It was shown that fault tolerance in the inference phase was significantly increased thanks to the integration of fault awareness into the model during the training process. This study offers an effective solution for increasing robustness at the software level. The NAPER method developed by Rajagede et al., (2025) offers a new error protection framework for real-time and resource-constrained DNN systems. This ensemble-based approach reduces the impact of bit-flip errors while providing viable solutions in terms of latency and resource usage. The study offers a significant contribution, especially for embedded and critical systems. Nazari et al., (2024) investigated the resilience of transformer-based models to bit-flip attacks and proposed specific protection strategies for these architectures. The study reveals that modern deep learning architectures can be vulnerable to hardware failures, while offering resilience enhancement techniques that can be implemented at the model level.

In this paper, we analyze the effects of hardware errors that manifest themselves as bitflips in hardware components on ANN applications. We evaluated both GPU and CPU-based systems with a broad fault injection experiment set. We also present a methodology to reduce the fault injection time.

III. FAULT TOLERANCE ASSESSMENT

This section describes the fault simulation environment, the fault injection process, and the evaluation of these faults.

A. Benchmark Applications

In this study, computer vision applications within the scope of machine learning are focused on and analyses are performed on two widely used datasets, MNIST (Russakovsky et al., 2015) and ImageNet (Deng et al., 2009), using two different models, Multi-Layer Perceptron and Deep Neural Network (DNN). In order to make a comparative assessment in terms of fault tolerance, SPEC CINT2006 applications are also included in the study as a reference. In this section, technical details of all applications used in the study are presented.

A.1. Classical ANN with MNIST Dataset

In the first set of experiments, a simple Artificial Neural Network (ANN) model was trained using the MNIST dataset. MNIST is a widely used image database containing handwritten digits between 0 and 9 (LeCun et al., n.d.). The model used in this experimental study has a classical Multi-Layer Perceptron architecture consisting of input, hidden, and output layers and was developed with the C++ programming language. As a result of the training process, two separate weight matrices were obtained. The first represents the connections between the input and hidden layers, and the second represents the connections between the hidden and output layers. These weight matrices are used in the testing (inference) phase of the model. The model was trained on a total of 60,000 examples and then evaluated with a test set containing 10,000 examples. In the ideal scenario where no error injection was applied, the neural network model achieved an accuracy rate of 94.40% on the test set.

A.2. VGG model with ImageNet Dataset

In the second set of experiments, Deep Neural Networks (DNN), which are Artificial Neural Networks with a large number of hidden layers, were targeted. In this context, a large-scale DNN model based on the VGG architecture was trained on ImageNet, which is a comprehensive dataset containing high-resolution and multi-category images. ImageNet is a benchmark dataset that is widely used for object recognition tasks and contains hundreds of thousands of labeled images. The VGG16 model used in this study is the deep convolutional neural network architecture developed by K. Simonyan and A. Zisserman from Oxford University and proposed in the study titled "Very Deep Convolutional Networks for Large-Scale Image Recognition" (Simonyan & Zisserman, 2014). Deep structures are preferred because they offer stronger generalization and pattern learning capabilities on datasets with high complexity; architectures such as VGG16 enable the successful learning of complex patterns. In this experiment, in order to apply fault injection analysis, 10,000 images predicted with high accuracy were selected from the validation set consisting of a total of 50,000 samples and used in the analysis.

A.3. SPEC CINT2006

The SPEC CPU2006 benchmark suite was developed by the Standard Performance Evaluation Corporation (SPEC) to comparatively evaluate the performance of compute-intensive systems on a wide range of hardware based on workloads derived from real-world user applications (Henning, 2006). This benchmark suite includes 12 integer-based (CINT) and 19 floating-point-based (CFP) applications. In order to perform comparative durability analyses in this study, seven different integer applications were selected from the SPEC CINT2006 benchmark suite presented in Table 1 and examined in detail.

Table 1. SPEC CPU2006 CINT Benchmarks

Application Name	**General Category**
401.bzip2	Compression
403.gcc	C compiler
429.mcf	Combinational optimization/ Single-depot vehicle scheduling
456.hmmer	Search a gene sequence database
458.sjeng	Artificial intelligence
462.libquantum	Physics/ Quantum Computing
464.h264ref	Video compression

B. Fault Injection and Simulation Environments

This section describes in detail the different methods used to inject hardware faults. CPU-based experiments are conducted on a Dell PowerEdge R720 server with the following hardware and software specifications: Intel(R) Xeon(R) E5-2650 0 @ 2.00GHz processor, 16 cores, 128 GB DRAM, Linux 4.4.0-57 operating system, and GCC 5.4.0 compiler. GPU-based experiments are conducted on an NVIDIA V100 Tensor Core GPU. Figure 1 abstracts the life cycle of a typical Artificial Neural Network (ANN) application. In this process, the model is first trained using labeled training data and weight matrices are created accordingly. In the test (inference) phase, the weight matrices obtained in training are used as fixed parameters; this phase is performed by running the model with test data on a CPU or GPU-based system and obtaining the classification results. During the training process, various components such as the input layer, hidden layer,

convolutional layers, and fully connected layers are executed sequentially. In order to ensure the data flow between these layers, the intermediate layer outputs are produced as matrices and transferred to the next layers in the system.

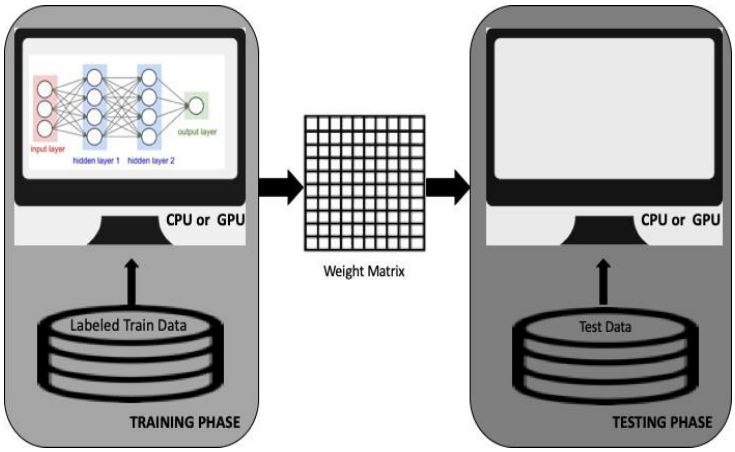


Figure 1. Fault Injection to ANN Applications.

Although fault injection provides a highly accurate method to evaluate system reliability, it is quite costly in terms of time since it requires re-running the application with random errors in each experiment. One of the main objectives of this study is to propose an effective method to reduce the fault injection time for Artificial Neural Network (ANN) applications. For this purpose, faults are directly injected into the weight matrices to evaluate the effects of faults on the training process or different layers of the network. The proposed approach assumes that only the faults affecting the weight matrix are significant, and all other faults have negligible effects on the system output, i.e., are benign. Three different methods were applied within the scope of fault injection: (i) In the first method, faults were directly injected into the weight matrices to examine the effects of hardware faults on different layers and processing stages of the network. In the MNIST application, the weight matrices between the input-hidden and hidden-output layers were targeted. In the ImageNet application, the weight matrices between the convolutional and fully connected layers were targeted, and the faults created on these matrices were evaluated as permanent. As the fault does not disappear unless the matrix is recreated. (ii) In the second method, in systems running on the CPU, faults were injected into the architectural registers of the processor during the training and testing stages, and the system sensitivity was analyzed under different input types, especially representing hard disk corruptions. (iii) In the third method, the CUDA-based ANN application was run on the GPU, and comparative sensitivity analyses were performed with the CPU version of the same application. In this context, the effects of different hardware architectures on the reliability of ANN applications were examined in detail.

B.1. Faults in NN Layers and Phases

An Artificial Neural Network (ANN) consists of many basic components such as neurons, connections, activation functions, and weights, and hardware errors can affect any of these components. In this study, the weight matrices between layers or processing stages are directly manipulated to simulate error occurrences in the relevant layer or stage, and the corrupted matrices are given as input to the next layer/stage. For example, the model obtained at the end of the training process is directly used in the testing stage; this model consists of weight matrices created by updating various parameters during learning. In order to model the effect of hardware errors that may occur during the training process, certain bits in the relevant weight matrices are directly corrupted. Similarly, the error scenarios that may occur in the input layer or the hidden layer were simulated by changing the matrices between the input-hidden and hidden-output layers, respectively. In the Deep Neural Network (DNN) applications, similar error injections were applied to the weight matrices between the convolutional layers and fully connected layers. Depending on the experimental scenario, 1 to 8 permanent errors were injected into each layer. After the errors were injected, the ANN model processes the test data, predicts the output, and the result is systematically recorded and evaluated.

B.2. Architectural Register Faults

Fault injections at the architectural register level were performed using a binary analysis tool called PIN (Program Instrumentation) (Luk et al., 2005) developed by Intel. In this context, analysis codes that would be executed before each machine instruction were injected into the system. However, faults were not applied to each instruction, instead, fault injection was triggered just before a randomly selected instruction was executed. Faults were implemented by bit-flipping a bit in a randomly selected register among 16 general-purpose architectural registers. After the fault was injected, the system continued execution by executing the next instruction. In these experiments, both Single Event Upset (SEU) and Multiple Bit Upset (MBU) scenarios are evaluated. In the MBU scenario, instead of a single fault at each application execution, five different bit corruptions are applied and these faults are distributed at random points

throughout the execution timeline of the application. In case of early termination of the application, fewer faults are injected in that execution. The location of each bit corruption is also determined randomly. Fault injections are performed by performing 50 iterations on each input sample for each architectural record. This method is applied to both the ANN application and the SPEC CPU2006 applications used for comparative analysis, and comprehensive fault sensitivity comparisons are made for both scenarios. In addition, the effects of different input values on the system behavior during the test phase are also analyzed by this method.

B.3. Faults in GPU

Fault injections in GPU-based systems were performed using a publicly available fault injection tool called NVBitFI (Tsai et al., 2021). NVBitFI is a tool that can perform dynamic instrumentation on binary code without requiring access to the source code, and is built on the NVBIT (NVIDIA Binary Instrumentation Tool) (Villa et al., 2019) infrastructure, which allows injecting user-defined analysis codes into applications running on NVIDIA GPUs. In this study, the CUDA version of the ANN application running on the MNIST dataset was implemented—without any optimization or algorithmic improvements—as the same version used on the CPU. In the fault injection process, a transient fault was injected into a randomly selected GPU core after a random number of instructions were executed. The main goal of this experiment is to evaluate the impact of different hardware architectures (especially CPU and GPU) on the reliability of ANN applications and to present comparative analyses.

IV. EVALUATION AND DISCUSSION

In this section, the results obtained within the scope of the study are discussed. The evaluations observed for each process are presented under headings.

A. Evaluation

The evaluation process is structured under three main headings. First, the effects of errors injected into different layers (input layer, hidden layer, convolutional layers, and fully connected layers) in both Artificial Neural Network (ANN) and Deep Neural Network (DNN) models on the system behavior are analyzed. In addition, the effects of errors occurring in different processing stages such as training and testing stages are comparatively evaluated. In the second stage, the effects of error scenarios applied to different input data as well as errors injected into different processor architecture registers are examined; this allows modeling scenarios such as hard disk corruptions or data transmission errors. At the same time, the robustness of neural network-based applications is analyzed comparatively with SPEC CINT2006 benchmark applications. In the third and last stage, the effects of architectural differences are evaluated; the effects of CPU and GPU-based system architectures on the reliability of the ANN application during the testing phase are comprehensively analyzed.

A.1. Vulnerability of Different Neural Network Layers and Phases

The neural network used in the MNIST implementation consists of three layers: an input layer with 784 neurons, a hidden layer with 128 neurons, and an output layer with 10 neurons. Within the scope of the experimental analysis, hardware faults were injected into the weight matrices between these layers. The faults were implemented by considering the bit positions representing the sign, exponent, and mantissa fields in the IEEE 754 floating-point representation. Different bit positions were targeted, especially starting from the 63rd bit, which is the sign bit, to the 48th bit, including the significant bits of the exponent. For each bit position, the number of injected faults was tested from 0 to 1000, with increments of 100 units.

Figure 2(a) presents the accuracy results obtained when errors are injected into the first weight matrix that is, the connections between the input layer and the hidden layer. The findings show that there are serious drops in accuracy, especially when errors are injected into the higher-order bits, such as the most significant bit of the exponent, bit 62. As the number of errors increases, the accuracy decreases linearly, dropping to approximately 30%. However, it is observed that the distortions made in the lower-order bit positions have limited effects and the model accuracy generally remains above 90%. This is consistent with previous findings in the literature indicating that ANNs are relatively robust to certain error scenarios. Figure 2(b) reports the case where errors were injected into the second weight matrix the connections from the hidden layer to the output layer. Errors at this level had much stronger effects on the system output. While a significant decrease in accuracy was observed as the number of errors increased for a fixed bit position, an even faster decrease in accuracy was observed when the bit position was changed for the same number of errors. In particular, the corruptions at the 62nd bit position reduced the model accuracy to less than 10%, almost eliminating the functionality of the system. In contrast, the accuracy rate remained above 90% for error injections made to lower bit positions in the mantissa section.

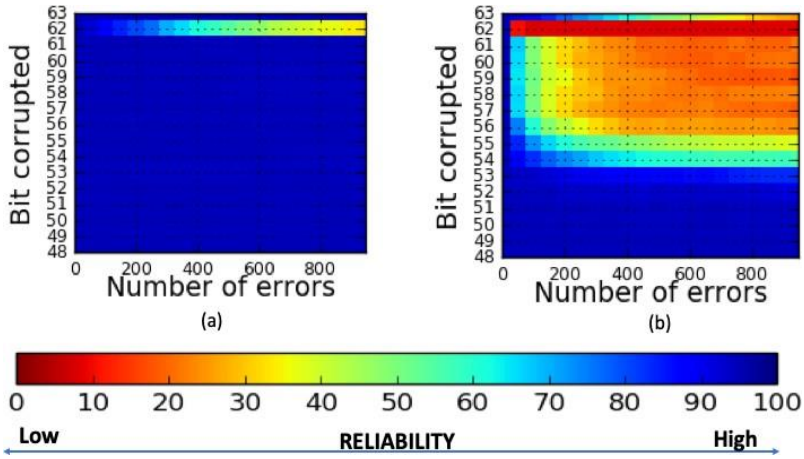


Figure 2. Injecting in the weight matrix (a) between the input layer and the hidden layer (b) between the hidden layer and the output layer.

Figure 3 shows the model accuracies obtained on the ImageNet dataset when persistent errors are injected into different layers especially convolutional layers and fully connected layers. In this experimental scenario, errors are injected into bit positions 29, 30, and 31, and it is observed that bit 30 has the most significant impact on model performance. In Figure 3, only the results obtained for bit position 30 are detailed.

The findings show that there is a significant decrease in model accuracy when the number of errors reaches eight. In addition, it is observed that error injections applied to convolutional layers negatively affect model performance more than fully connected layers. This shows that the convolutional structure is more critical in terms of error sensitivity and these layers should be evaluated more carefully in terms of robustness.

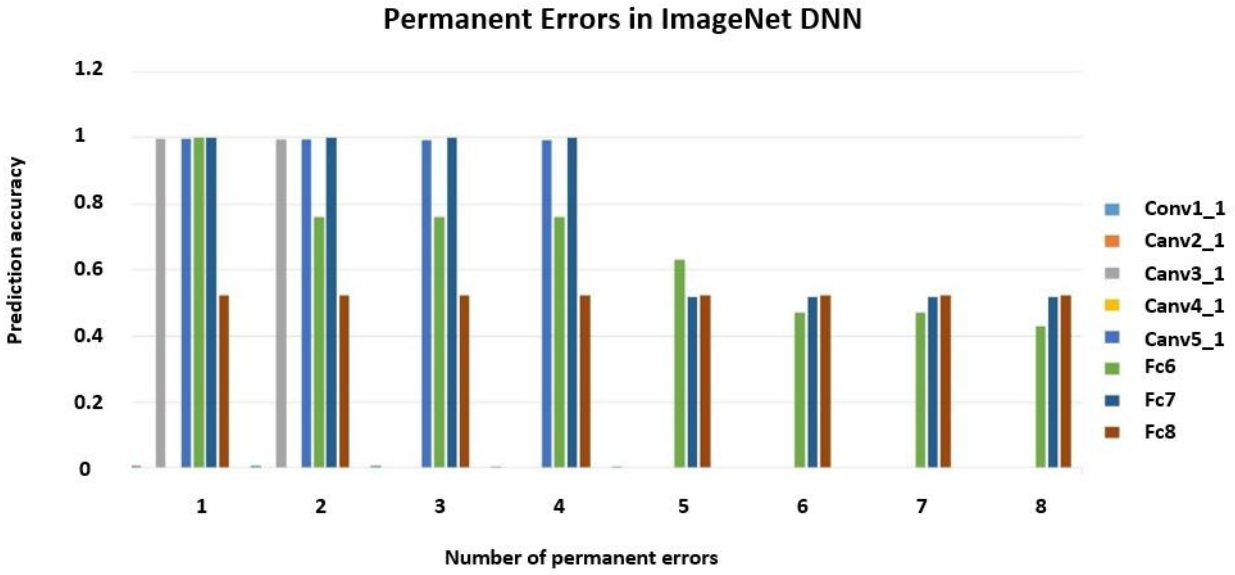


Figure 3. Fault Injection to different layers in ImageNet Application

Finally, Figure 4 shows the impact of errors occurring during the training phase on the overall accuracy performance of the model. In the ideal scenario without error injection, the model achieved over 94% accuracy. However, as an increasing number of errors (between 0 and 1000) were injected into the training process, a significant decrease in model accuracy was observed. This clearly demonstrates that persistent errors occurring during the training phase negatively affect the learning process. However, it is striking that even a model that has been degraded during training can maintain a certain level of accuracy, i.e., does not become completely dysfunctional. This result shows that neural networks can have a certain tolerance to limited levels of deterioration.

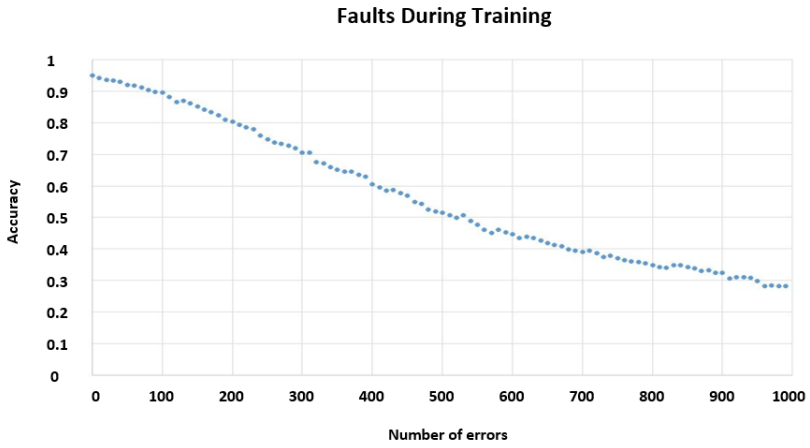


Figure 4. Faults during training.

A.2. Vulnerability of Processor Registers

In this experiment, errors were injected into the processor architectural registers during the execution of the MNIST application. Since these injected errors do not always affect the output of the application. They are directly classified as "faults"; however, they are not evaluated as "errors". This is because if another process overwrites the value of the relevant register after an error occurs, the effect of the error is masked by the system. In this context, one of the main elements that we want to observe experimentally is this "masking" effect. On the other hand, when the error affects the application behavior, this effect can result in a crash or silent data corruption (SDC). In this direction, the experimental outputs were analyzed by dividing them into three main categories: normal operation, crash, and SDC. Figure 5 presents the results obtained when errors were injected into different architectural registers during the test phase, and three main findings stand out in this context. First, it is observed that fault injections on special-purpose registers RSP and RBP significantly increase the crash rate. These two registers play a critical role in function calls; RBP points to the stack bottom and RSP points to the stack top. The results show that ANN almost always fails when these registers are affected. Second, it is found that fault injections on RDX, a general-purpose data register, cause a high rate of SDC. This register can reflect the fault effect on the system output depending on the data it holds and the instructions it executes; in fact, it is observed that faults applied to RDX lead to a high number of misclassifications. This poses a critical threat that can cause dramatic drops in model accuracy, especially when the error affects the weight matrices. In conclusion, this study reveals that, unlike some previous literature studies, Neural Networks are more vulnerable to architectural record-level errors than expected, as any error at the record level can potentially lead to either a crash or severe SDC.

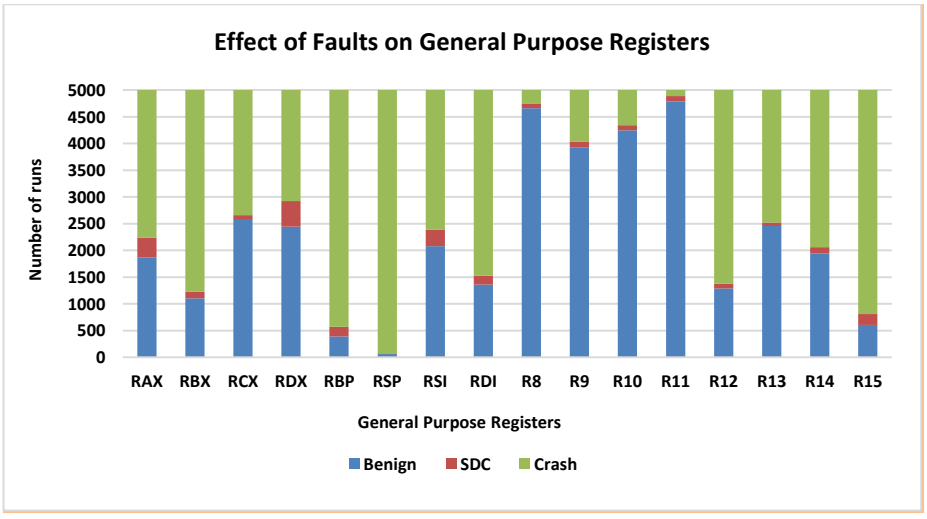


Figure 5. Effect of faults on general-purpose registers

A.3. Characterization of Errors for Different Input Datasets

In this section, we aim to evaluate the effects of corruption on neural network performance in different input data samples. More specifically, we examine whether certain visual samples are more sensitive to prediction errors than others when errors are injected. In this context, first, each sample in the MNIST dataset is labeled according to the digit it represents (e.g., digit 1 → label 1, digit 2 → label 2). Then, random errors are injected into the processor architecture

registers for each group of labels. Figure 6 shows the effect of architectural registration errors on the classification of different labels. The obtained results show that some visual labels are more affected by hardware errors than others. For example, it is observed that almost none of the images belonging to label 0 experience silent data corruption (SDC). This may be due to the low similarity of the visual structure of the number 0 with other numbers. The neural network can classify 0, which is structurally dissimilar to other numbers, with high accuracy, and therefore exhibits a more resilient behavior against errors. In contrast, images belonging to label 4 are found to have higher sensitivity to SDC. This may be due to the higher probability of visual confusion of the number 4 with similar numbers such as 9, especially in cursive form. This leads to hardware-level error injections creating more confusion in the classification process and thus causing more frequent incorrect predictions.

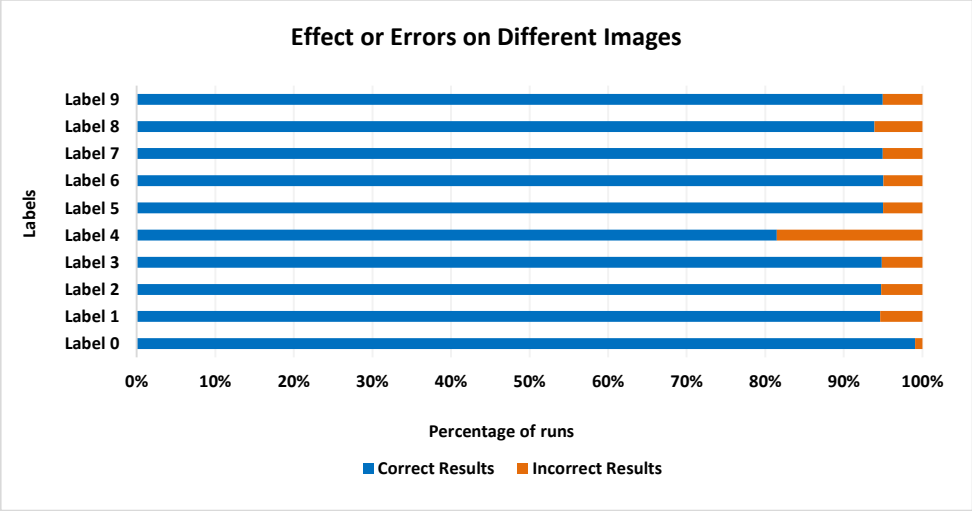


Figure 6. Effect of faults on different images.

In the tested artificial neural network model, an error score is generated for each classification result. For example, if a handwritten number is classified as 9, the model calculates an error score regarding the accuracy of this prediction. If this score falls below a predefined threshold value, the prediction is considered correct; however, if it exceeds the threshold value, the prediction is considered incorrect.

Figure 7 shows the error score changes when faults are injected into different processor architecture registers. The findings reveal that fault injection has significant effects on the error score for all register types. It is also observed that the increase in the error score varies depending on the register type in which the fault is injected. In particular, certain registers such as RDX, RBP, and R15 cause significant increases in the error score, indicating that these registers are more critical to system accuracy.

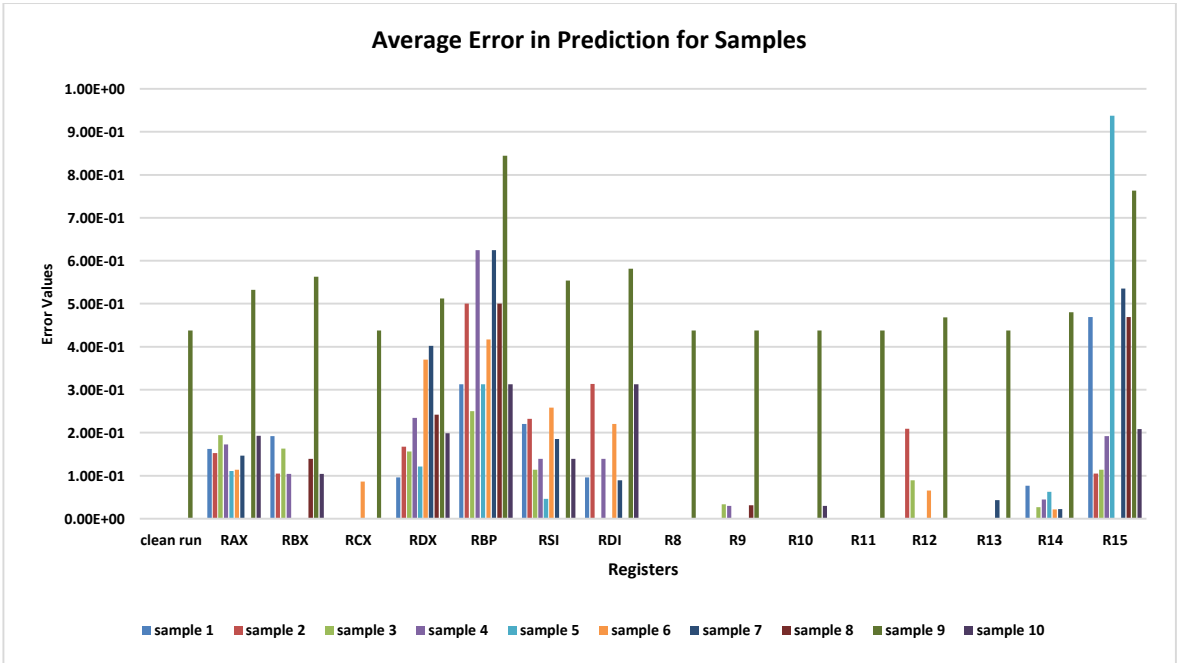


Figure 7. Average error in prediction.

A.4. Vulnerability of SPEC CINT2006 Applications

Figure 8 presents the vulnerability of Artificial Neural Network (ANN) applications to hardware faults in comparison with the SPEC CINT2006 benchmark applications. According to the experimental results, the 401.bzip2 application is the application that experiences the most failures with a 36% crash rate, while the 458.sjeng application is almost completely unaffected by crash errors with only a 1.86% crash rate. These findings reveal that sjeng is the most resilient application against crash-type faults, while bzip2 is the most vulnerable application. It is observed that all benchmark applications are affected by silent data corruption (SDC) to some extent. In this context, the hmmer application had the highest SDC rate of 4.9%, while the gcc and mcf applications were the least affected applications with very low rates of 0.1%. When a general resilience analysis was performed according to the types of faults, it was determined that the sjeng application showed 96% fault tolerance and was the application with the highest overall resilience. On the other hand, the bzip2 application exhibited the lowest reliability with a total vulnerability of 40.5%. In addition, comparisons between ANN implementations and SPEC CINT2006 implementations reveal that ANNs are more vulnerable to hardware errors applied to processor registers. As a result of the experiments, a total of 52% crash, 3% SDC, and 45% benign error rate were observed for ANN implementations. Although the bzip2 implementation is the most vulnerable example in the SPEC CINT2006 set, it is concluded that these implementations generally exhibit higher reliability against hardware errors compared to ANNs.

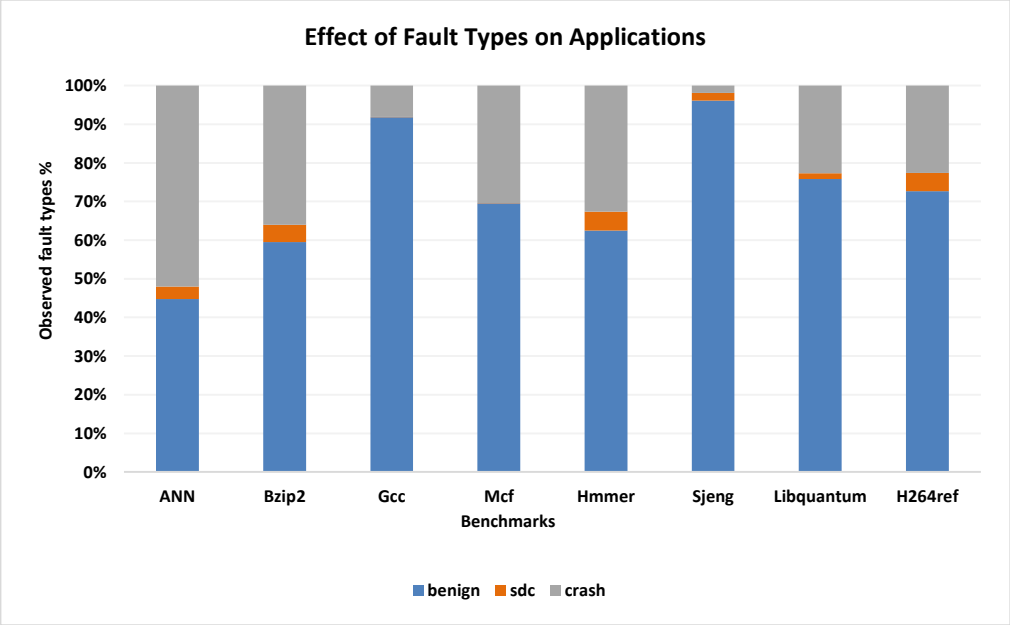


Figure 8. Impact of Faults on ANN and SPEC CINT2006 Applications.

A.5. Vulnerability in GPU Hardware

Figure 9 presents the vulnerability of the MNIST implementation to hardware faults when run on the GPU. In this experiment, the CUDA-based version of the ANN algorithm is used and random fault injections are applied to the GPU cores while the implementation is running on the GPU. The results are compared with the same implementation version on the CPU. The comparison is made only on registers R8-R15, since these registers are used only for internal ALU computations, while the others serve special purposes such as memory addressing or stack marking. This choice was made because the GPU architecture is tasked with large-scale parallel computations (e.g., vector operations). In contrast, the instruction flow and control operations are mainly executed by the CPU. According to experimental findings, less than 1% of the errors injected into the GPU caused silent data corruption (SDC) and produced a different output. Approximately 90% of the errors were masked, i.e., they did not cause any corruption on the system output. Another 3% resulted in a timeout or a system crash, while the remaining 7% produced outputs different from the original execution, and these erroneous situations were reflected in the dmesg log records by the Linux operating system. This 7% part is classified as “Detected Unrecoverable Error” (DUE) because it can be detected by the system. Two main conclusions can be drawn from these results. First, ANN applications can exhibit a certain level of vulnerability when run on GPU; therefore, it is recommended to integrate reliability-enhancing strategies such as checkpointing to prevent recalculation. Second, GPUs are less error-sensitive compared to CPU executions. This can be explained by the fact that GPUs are used predominantly for parallel vector computations and not for system-critical tasks such as control flow, branch routing, or stack pointers.

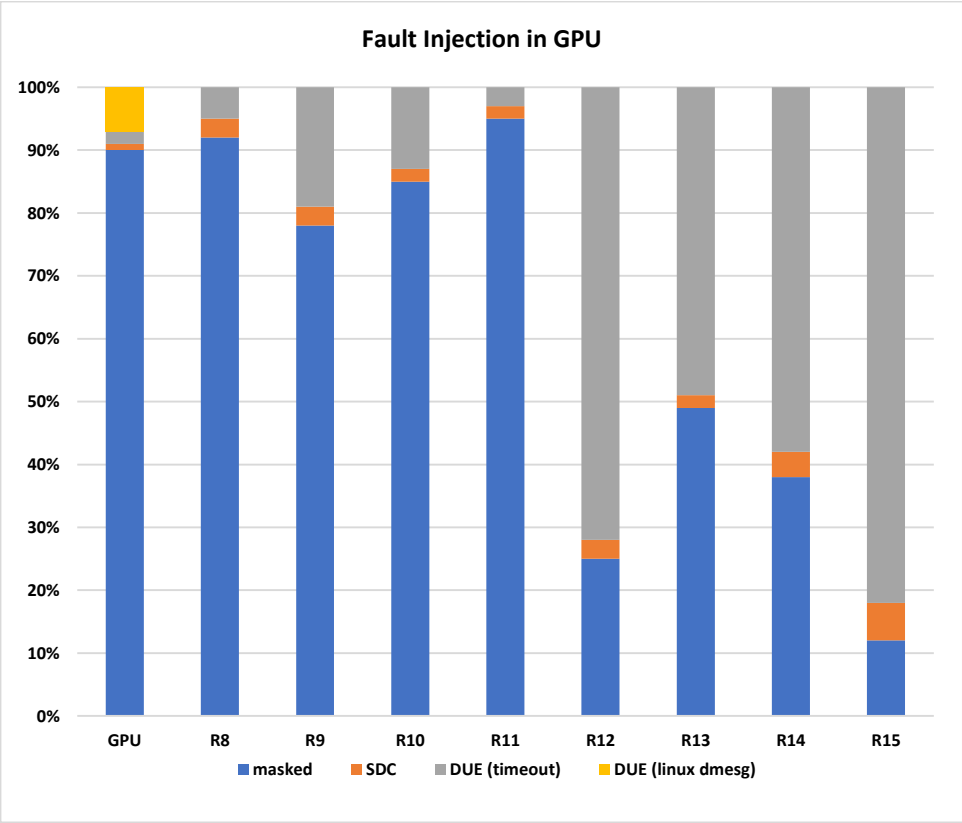


Figure 9. Injected faults in the GPU version of the testing phase of MNIST. The figure presents a comparison between faults in GPU vs faults in CPU for the registers used for the calculation of data.

B. Discussion and Suggestion

This section discusses possible optimizations to reduce the negative impact of reliability measures on performance by balancing minimum reliability with maximum performance gain.

B.1. Input Reliability

The obtained results show that different labels exhibit different levels of sensitivity to hardware errors. In particular, when the input data represents the number 4, the probability of misclassification in a faulty operating environment is at the highest level, and the majority of these faulty predictions result in a 9. This observation suggests that some labels (e.g., label 4) have a higher risk of error due to structural similarities. For such high-risk labels, it is suggested to repeat the execution twice and compare the results. This method can detect fault-induced mispredictions, resulting in a more reliable and stable output.

B.2. Phase Reliability

The findings show that the most critical stage in terms of neural network reliability at the application level is the training stage. The main reason for this is that the model parameters obtained during the training process are directly transferred to the testing stage. Data corruptions occurring during training cause the model to learn incorrectly. This causes not only a loss of accuracy, but also a delay in the learning process (longer convergence time) and a decrease in the overall performance of the testing stage. In this context, considering that the data between the hidden layer and the output layer is more vulnerable in terms of error tolerance, it is recommended to implement additional error detection and/or correction mechanisms for the training process.

In this section, the system load brought by applying special protection only to the training phase is compared with the scenario of applying full protection throughout the entire process. For this purpose, EDDI (Error Detection by Duplicated Instructions), a software-based reliability mechanism proposed by Oh et al., (2002) and his colleagues, is used. EDDI is based on the principle of duplicating each instruction using different registers and variables during the compilation phase. During execution, the original registers are compared with the shadow (duplicated) registers before writing data to memory or at the end of basic blocks. If there is an error, in the event of a possible inconsistency, error detection is performed. These multiplication operations performed at the application level add minimal overhead to the execution time thanks to the possible parallelism at the instruction level (ILP). In addition, an example code fragment converted using the EDDI method and the possible execution order are presented below. In this example, the original instructions are shown in black, the shadow instructions are shown in red, and the compare instruction, which matches the original registers with the shadow registers and checks their correctness, is shown in blue.

I1: ADD R1, R2, R3
I2: SUB R4, R1, R2
I1': ADD R21, R22, R23
I3: AND R5, R1, R2
I3': AND R25, R21, R22
I4: MUL R6, R4, R5
I2': SUB R24, R21, R22
I4': MUL R26, R24, R25
Ic: BNE R6, R26, goToErrorHandler
I5: ST R6
I5': ST R26

The EDDI mechanism increases the total number of instructions executed by more than 100% by duplicating each instruction and adding additional verification instructions. In this study, it is proposed to protect only the training phase to achieve an optimal balance between performance and reliability. Figure 10 compares the overhead caused by protecting only the training phase with the overhead caused by protecting both the training and testing phases together, based on the number of instructions executed in a fault-free system. As can be seen, the training process takes longer and involves more operations than the testing process. However, in practice, the training process is performed only once, while the testing process is repeated for many different inputs. In this context, the evaluation is made for scenarios where the testing process is executed one, five, ten, twenty, fifty, and one hundred times. Two main conclusions are drawn from this analysis: First, applying protection only to the training process imposes a lower performance overhead than replicating the entire system. Second, when the testing process is repeated more often, the overhead caused by protecting only the training process is relatively reduced. For example, when the testing process is repeated approximately 50 times, the total overhead drops below 50%.

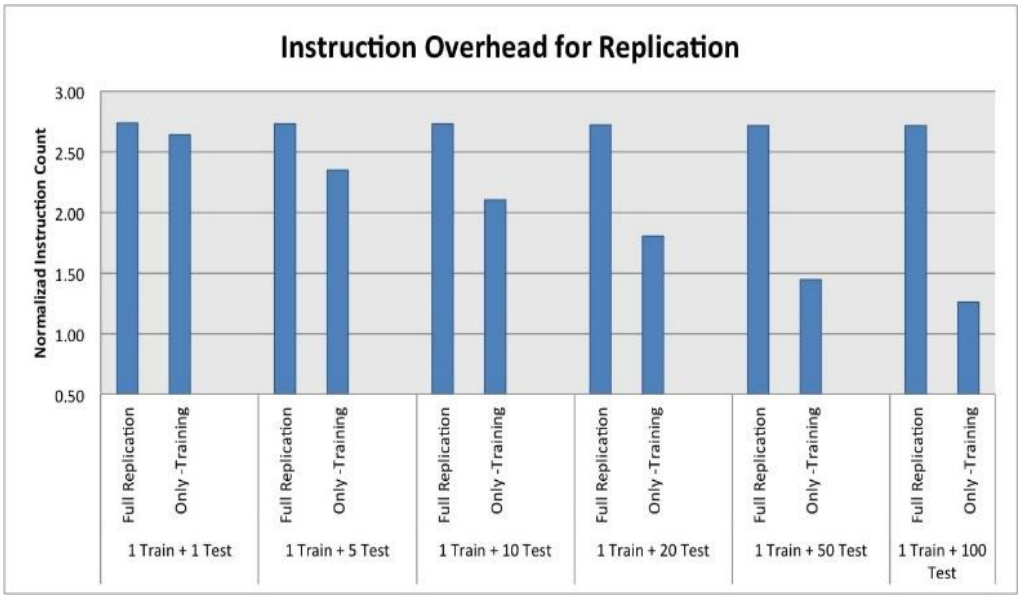


Figure 10. Overheads of protecting only training compared to the overheads of protecting both training and testing.

B.3. Register Reliability

The findings show that some processor registers (e.g., RSP and RBP) have a decisive effect on Artificial Neural Networks (ANNs), while some registers have a negligible effect on accuracy. It has also been observed that different components of ANN applications have different levels of sensitivity to error effects. This situation reveals that application-specific reliability designs can be developed. For instance, different levels of error correction strategies can be applied to registers. ECC (Error Correction Code) can be recommended to be applied to more critical and vulnerable registers such as RSP and RBP, while such precautions may be unnecessary for less-used registers such as R8, R9, and R12. In line with this approach, register groups with different security levels can be defined and this information can be provided to compilers or application developers, thus optimizing the use of registers appropriate to data sensitivity.

V. CONCLUSION AND FUTURE WORK

It is anticipated that Artificial Neural Networks (ANNs) will continue to be used increasingly as a computational model, especially in AI-based applications. Such applications are currently run on High Performance Computing (HPC) systems. As the field approaches exascale systems, the reliability of these applications becomes a critical research

problem due to the high error rates expected on these platforms. In this study, we show that the common belief that ANNs are generally tolerant to errors is not valid in all cases. It is shown that different components of neural networks exhibit different sensitivities to error effects, and that hardware-related errors and silent data corruption in particular can lead to serious deviations in neural network behavior. In addition, the fault sensitivity of ANN applications is analyzed comparatively with SPEC CINT2006 benchmark applications; different fault scenarios on CPU and GPU architectures are evaluated. In addition, improvement suggestions that can be implemented at the design level to ensure the reliability of ANNs are also included.

It is known that machine learning (ML) based models are traditionally trained on HPC infrastructures due to their high computational power and memory requirements. However, it has been observed that these models are recently being moved to energy-constrained end devices such as mobile phones. The main reasons for this trend include reducing the need for data transfer, improving user experience, and meeting increasing security and privacy concerns. However, since energy efficiency is at the forefront in such devices, it is common for models to be run with simplified and low-precision weights. This makes it difficult to verify the accuracy of each prediction process; the effect of silent data corruption on the prediction results is often unobservable. In this context, future studies aim to conduct end-to-end fault tolerance analyses, and to comprehensively examine the effects of failures, especially in end systems where the voltage is reduced for low power consumption.

DECLARATIONS

Acknowledgements: The authors declare that there are no acknowledgements.

Author Contributions: Conceptualization, H.A.A.; Methodology, A.N.H.; Software, H.A.A. and A.N.H.; Validation, G.Y., and O.S.U.; Formal Analysis, G.Y., and O.S.U.; Investigation, H.A.A. and A.N.H.; Writing-Original Draft, H.A.A., A.N.H., G.Y., and O.S.U.; Writing-Review & Editing, H.A.A., A.N.H., G.Y., and O.S.U.; Visualization, H.A.A. and A.N.H. All authors have read and approved the final version of the manuscript.

Conflict of Interest: The authors declare that they have no conflict of interest within the scope of this study.

Supporting Institutions: This research received no external funding.

Ethical Approval: This study does not involve human or animal participants. All procedures followed scientific and ethical principles, and all referenced studies are appropriately cited.

Plagiarism Statement: This article has been evaluated for plagiarism and no instances of plagiarism were detected.

Use of AI Tools: This study benefited from linguistic refinement provided by ChatGPT, an advanced AI language model developed by OpenAI.

REFERENCES

Alobaid, A., Bonny, T., & Alrahhal, M. (2025). Disruptive attacks on artificial neural networks: A systematic review of attack techniques, detection methods, and protection strategies. *Intelligent Systems with Applications*, *26*(1), 200529. https://doi.org/10.1016/j.iswa.2025.200529

Bautista Gomez, L. A. B., & Cappello, F. (2015). Detecting and correcting data corruption in stencil applications through multivariate interpolation. In *Proceedings of the IEEE International Conference on Cluster Computing* (pp. 595–602). IEEE. https://doi.org/10.1109/CLUSTER.2015.108

Bengio, Y., Lecun, Y., & Hinton, G. (2021). Deep learning for AI. *Communications of the ACM*, *64*(7), 58–65. https://doi.org/10.1145/3448250

Borkar, S., & Chien, A. A. (2011). The future of microprocessors. *Communications of the ACM*, *54*(5), 67–77. https://doi.org/10.1145/1941487.1941507

Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., & Snir, M. (2009). Toward exascale resilience. *The International Journal of High Performance Computing Applications*, *23*(4), 374–388. https://doi.org/10.1177/1094342009347767

Deng, J., Dong, W., Socher, R., Li, L. J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 248–255). IEEE. https://doi.org/10.1109/CVPR.2009.5206848

Di Martino, C., Kramer, W., Kalbarczyk, Z., & Iyer, R. (2015). Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 HPC application runs. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (pp. 25–36). IEEE. https://doi.org/10.1109/DSN.2015.50

Henning, J. L. (2006). SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, *34*(4), 1–17.

https://doi.org/10.1145/1186736.1186737
Kulakov, A., Zwolinski, M., & Reeve, J. (2015). *Fault tolerance in distributed neural computing* [Preprint]. https://doi.org/10.13140/RG.2.1.1387.0800
Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., & Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*, *40*(6), 190–200. https://doi.org/10.1145/1064978.1065034
LeCun, Y., Cortes, C., & Burges, C. J. C. (n.d.). *MNIST handwritten digit database*. Retrieved March 18, 2026, from https://yann.lecun.org/exdb/mnist/index.html
Nazari, N., Makrani, H. M., Fang, C., Sayadi, H., Rafatirad, S., Khasawneh, K. N., & Homayoun, H. (2024). Forget and rewire: Enhancing the resilience of transformer-based models against bit-flip attacks. In *Proceedings of the 33rd USENIX Security Symposium* (pp. 1348-1366). https://www.usenix.org/conference/usenixsecurity24/presentation/nazari
Oh, N., Shirvani, P. P., & McCluskey, E. J. (2002). Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, *51*(1), 63–75. https://doi.org/10.1109/24.994913
Piuri, V. (2001). Analysis of fault tolerance in artificial neural networks. *Journal of Parallel and Distributed Computing*, *61*(1), 18–48. https://doi.org/10.1006/jpdc.2000.1663
Rajagede, R. A., Santriaji, M. H., Fikriansyah, M. A., Nuha, H. H., Fu, Y., & Solihin, Y. (2025). NAPER: Fault protection for real-time resource-constrained deep neural networks. In *Proceedings of the IEEE 31st International Symposium on On-Line Testing and Robust System Design (IOLTS)*. IEEE. https://doi.org/10.1109/IOLTS65288.2025.11116827
Ruospo, A., Gavarini, G., de Sio, C., Guerrero, J., Sterpone, L., Reorda, M. S., Sanchez, E., Mariani, R., Aribido, J., & Athavale, J. (2023). Assessing convolutional neural networks reliability through statistical fault injections. In *Proceedings of the Design, Automation and Test in Europe Conference & Exhibition (DATE)* (pp. 1-6). IEEE. https://doi.org/10.23919/DATE56975.2023.10136998
Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., & Fei-Fei, L. (2015). ImageNet large scale visual recognition challenge. *International Journal of Computer Vision, 115*(3), 211–252. https://doi.org/10.1007/s11263-015-0816-y
Simonyan, K., & Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition*. [arXiv preprint]. https://arxiv.org/pdf/1409.1556
Su, F., Yuan, P., Wang, Y., & Zhang, C. (2016). The superior fault tolerance of artificial neural network training with a fault/noise injection-based genetic algorithm. *Protein & Cell*, *7*(10), 735–748. https://doi.org/10.1007/s13238-016-0302-5
Tchernev, E. B., Mulvaney, R. G., & Phatak, D. S. (2005). Investigating the fault tolerance of neural networks. *Neural Computation*, *17*(7), 1646–1664. https://doi.org/10.1162/0899766053723096
Tiwari, D., Gupta, S., Rogers, J., Maxwell, D., Rech, P., Vazhkudai, S., Oliveira, D., Londo, D., Debardeleben, N., Navaux, P., Carro, L., & Bland, A. (2015). Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)* (pp. 331–342). IEEE. https://doi.org/10.1109/HPCA.2015.7056044
Tsai, T., Hari, S. K. S., Sullivan, M., Villa, O., & Keckler, S. W. (2021). NVBitFI: Dynamic Fault Injection for GPUs. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (pp. 284–291). IEEE. https://doi.org/10.1109/DSN48987.2021.00041
Villa, O., Stephenson, M., Nellans, D., & Keckler, S. W. (2019). NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (pp. 372–383). https://doi.org/10.1145/3352460.3358307
Vinck, T., Jonckers, N., Dekkers, G., Prinzie, J., & Karsmakers, P. (2025). Mitigating multiple single-event upsets during deep neural network inference using fault-aware training. *Journal of Instrumentation*, *20*(02), Article C02044. https://doi.org/10.1088/1748-0221/20/02/C02044
Wang, C., Zhao, P., Wang, S., & Lin, X. (2024). *Detection and recovery against deep neural network fault injection attacks based on contrastive learning* [arXiv preprint]. http://arxiv.org/abs/2401.16766