Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia
COPPE UFRJ

# RESILIENT AND SECURE DEEP LEARNING-ORIENTED MICROARCHITECTURES

Brunno Figueirôa Goldstein

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Felipe Maia Galvão França
Sandip Kundu
Alexandre Solon Nery

Rio de Janeiro
Junho de 2022

# RESILIENT AND SECURE DEEP LEARNING-ORIENTED MICROARCHITECTURES

Brunno Figueirôa Goldstein

Orientadores: Felipe Maia Galvão França
Sandip Kundu
Alexandre Solon Nery

RIO DE JANEIRO, RJ – BRASIL
JUNHO DE 2022

Goldstein, Brunno Figueirôa

Resilient and Secure Deep Learning-Oriented Microarchitectures/Brunno Figueirôa Goldstein. – Rio de Janeiro: UFRJ/COPPE, 2022.

XVIII, 118 p.: il.; 29,7cm.

Orientadores: Felipe Maia Galvão França
Sandip Kundu
Alexandre Solon Nery

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2022.

Referências Bibliográficas: p. 105 – 117.

1. Deep Neural Networks. 2. Hardware Dependability. 3. Resiliency. 4. Security. I. França, Felipe Maia Galvão *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

*Dedicated to my beloved daughter Helena and wife Juliana, and my parents, Adecy and Ricardo.*

# Acknowledgements

To the committee members for kindly taking their time to evaluate this work and propose important feedback.

To my advisors, Felipe França, Sandip Kundu, and Alexandre Nery, for their dedication, inspiration, and ideas during the whole development process of this thesis. Without you, this work would not be feasible. Thank you for your wisdom and friendship.

To all authors cited in this work. This work would not be possible without the knowledge provided by you.

To Leandro Marzulo for presenting the academic life and advising me during my master's and Ph.D. Thank you for being such a friend, life advisor, and inspiration for me and several other students. The world needs more people like you.

To everyone at the department PESC/COPPE/UFRJ who makes the engines run. Your work makes students' lives less stressful by removing the heavy burden of bureaucracy and leaving our time entirely dedicated to the thesis development. To cite a few: Ricardo, Gutierrez, Mercedes, Cláudia. Similarly, to all the professors who make this department so singular and extraordinary. In particular, prof. Valmir Barbosa, prof. Claudio Amorim, prof. Guilherme Travassos and prof. Nelson Maculan.

To Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES Foundation) for providing a full-time scholarship in Brazil and a one-year scholarship as a visiting scholar at the University of Massachusetts, Amherst - US, making possible the development of this work.

To NGD Systems for investing in conjoint research within our laboratory.

To Intel Labs India for the knowledge, guidance, support through the development of this work, and the opportunity for the full-remote ten-month internship. In particular, Sudarshan Srinivasan, Kaul Bharat, and Pradeep Dubey.

To all my beloved friends that I made during this journey. The adventure would probably have become a nightmare without you. By putting some light into the darkness with smiles, jokes, guidance, or even small talk, you not only made this work possible but also forged the human that I have become. Hopefully, I will not miss anyone here. Friends from Mamatinha: Israel, Dani, Renan, Rebeca, Hugo, Spencer, Ana, Pedro, Moyses, Natália, Evandro, and Dani Castelo. Family from LAM: Santiago, Victor, Leopoldo, Hugo, Rui, Diego, and my former students Alan and Pedro. Friends from UMass: Vinay and Nazmul. Family from North Village: Felipe, Nur, Maria, Cecília, Brenno, and Pri. Friends from UERJ: Brazil, Ingrid, Ítalo, Denilson, Mentira, and Wellington. Friends at SC18: Julia, Ana, and Rafa.

To Victor, Santiago, and Leopoldo for their friendship, support, and collaboration during my whole Ph.D. LAM will never be the same without you.

Finally, to my family. To my beloved daughter Helena who was born and raised during the development of this work. Thank you for your love and kindness. To my wife Juliana, who supported me even before the Ph.D. Nothing of this would be possible without you. Thank you for your love, support, patience, and always believing in me – the best is yet to come. To my parents, Adecy and Ricardo, for the love, dedication, support, and patience. To my brothers, Rafael, and Ricardo, for the inspiration and support. To my dog Mel who always is there to give love and support.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

MICROARQUITETURAS ORIENTADAS A REDES NEURAIS PROFUNDAS RESILIENTES E SEGURAS

Brunno Figueirôa  Goldstein

Junho/2022

Orientadores: Felipe Maia Galvão França
Sandip Kundu
Alexandre Solon Nery

Programa: Engenharia de Sistemas e Computação

Redes Neurais Profundas (RNP) emergiram como uma importante solução para problemas complexos de diversas áreas, tais como visão computacional, processamento de linguagem natural e sistemas de recomendação. Um sistema ideal baseado em RNP deveria, de forma precisa, realizar predições e classificações utilizando-se de entradas quaisquer mas sem interferência do ambiente externo. Contudo, os sistemas baseados em RNP são suscetíveis a falhas devido às questões de confiabilidade e segurança. Esta tese se propõe avaliar diversos modelos RPN em condições adversas, como falhas do tipo bit-flip devido a fenômenos transitórios ou permanentes. Esta tese também propõe uma técnica de detecção de falhas baseada em codificação AN, focada em aceleradores de RNPs, empregados em sistemas críticos de segurança. Devido aos altos padrões exigidos, microarquiteturas empregadas em tais sistemas devem ser capazes de detectar mais de 99% de falhas em seu sistema. Além da alta robustez proporcionada pela detecção baseada em codificação AN, a solução possui implementação não custosa, principalmente quando aplicado em conjunto com a solução inovadora de quantização baseada em códigos AN. Em questões de segurança, os métodos de ofuscação de RNPs baseados em treinamento proporcionam uma excelente proteção aos dados sensíveis, como os pesos. Contudo, tais técnicas possuem falhas nas quais informações importantes são vazadas, como a distribuição das classes resultante das classificações. Além disso, soluções de ofuscação têm de ser escaláveis, a fim de proteger todo o portfólio dos provedores de modelos. A técnica de ofuscação baseada em trocas provê uma solução robusta, segura e escalável para a proteção dos pesos das RNPs, evitando assim o acesso illegal aos dados.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

RESILIENT AND SECURE DEEP LEARNING-ORIENTED MICROARCHITECTURES

Brunno Figueirôa Goldstein

June/2022

Advisors: Felipe Maia Galvão França
Sandip Kundu
Alexandre Solon Nery

Department: Systems Engineering and Computer Science

Deep neural networks (DNNs) have emerged as crucial methods for solving complex problems in various domains, including computer vision, natural language processing, and recommendation systems. An ideal DNN-based system should accurately make predictions or classifications based on some input data with no interference from the external environment. However, DNN-based systems are susceptible to failures due to reliability and security issues. This thesis evaluates many compressed DNN models under faulty conditions like bit-flips due to transient or permanent faults. Then, an AN-based detection scheme targeting DNN accelerators deployed into safety-critical systems is proposed. Due to the high compliance standards, microarchitectures employed in this type of system must provide a detection capability of 99% of faults. Also, the AN-based detection offers a lightweight solution, particularly when incorporated with the novel AN code-aware quantization technique proposed in this thesis. Training-based obfuscation techniques have been successfully employed to protect DNN models from illegal access to sensitive data, such as the parameters. However, crucial information such as the output class distribution can be leaked to attackers, indicating that the target model has been compromised. Additionally, an obfuscation scheme must provide a scalable way to protect the model providers' portfolios. The novel swap-based obfuscation scheme provides a robust obfuscation of the DNN model parameters through a scalable and secure solution, avoiding any illegal access and use of the model by non-authorized entities.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Contemporary society is living in the artificial intelligence (AI) era, where Machine learning (ML), broadly embraced by transportation, financial, health care, and space exploration systems, has established itself and will co-exist ubiquitously. Steadily, safety-critical systems incorporate ML to perform additional tasks or even entirely replace the need for human interaction, increasing the degree of safety, integrity, and security demanded. Typical examples are fully autonomous driving cars, where the vehicle performs the driving task under any condition, with no human interference. Such tasks should be accomplished flawlessly, and the system should not be susceptible to any errors or attacks. Therefore, the ML system designs should account for dependability and security, with either hardware or software-based layers of protection, avoiding frequent and severe failures at an acceptable level.

The concept of a dependable and secure computing system is built on five fundamental pillars [12]:

- **Availability** relates to the system's readiness from the start of operability to any random time of request/use.
- **Reliability** comprehends the ability of the system to preserve its correctness during its execution, either facing errors or attacks.
- **Safety** can be defined as a condition of stability without catastrophic results to the user or ecosystem during its execution.
- **Integrity** relates to the ability to avoid improper external interference and keep its data and execution truthfulness.
- **Maintainability** relates to the easiness of maintenance and the ability to improve its reliability with past experiences.
- **Confidentiality** comprehends the capability of keeping its data safe under unauthorized access of known and unknown entities.

Over the past decades, researchers and companies have been working on new technologies to encompass these pillars in every system development stage, ranging from chip manufacturing to software development and deployment. For every new technology generation, new obstacles are added and must be overcome with novel ideas and designs (or evolving the old ones). For example, with the advancement of semiconductor device fabrication, shrinking of feature sizes, and billions of transistors, microprocessors are more prone to errors due to frequency, voltage, and temperature variations than in earlier generations [13]. Datacenter fleets, composed of a massive number of servers, are equipped with different technologies and under different "life periods", leading to constant and complex monitoring, testing, and fault-tolerant routines at scale [13, 14]. Additionally, it is impractical to cover all the possible points of failure during post-fab testing by manufacturers fully, leading to further discovery of vulnerabilities and ways to exploit them [15–17]. Therefore, the demand for novel dependability and security ideas is a constant and vital component in the system design lifecycle.

According to Avižienis *et al.* [12], the means to attain dependability and security in a computer system is comprised of four categories:

- **Fault prevention:** The system should prevent the manifestation or injection of faults.
- **Fault tolerance:** After acknowledging the existence of faults, the system should avoid failures and deliver an acceptable level of service.
- **Fault removal:** The system should reduce the rate and severity of faults under its manifestation.
- **Fault forecasting:** The system should estimate the current number of faults, future rates, and consequences.

While fault [1] prevention and tolerance aim to provide a trustworthy system, fault removal, and forecasting ensure that the former ones will meet the required specifications.

Fault prevention and fault tolerance can be applied to ML systems with various techniques and at different system layers. However, each layer will have distinct requirements and constraints, such as degree of protection, exposure, computation capacity, area, and energy costs. Figure 1.1 depicts high-level blocks presented in the computation stack of an autonomous car system, ranging from physical sensors (lower level) to the software application (higher level). Each block relies on the adjacent ones, trusting that the data received is error-free or not malicious. Hence,

[1]The formal definition of faults, errors, and failures will be clarified in Chapter 2.

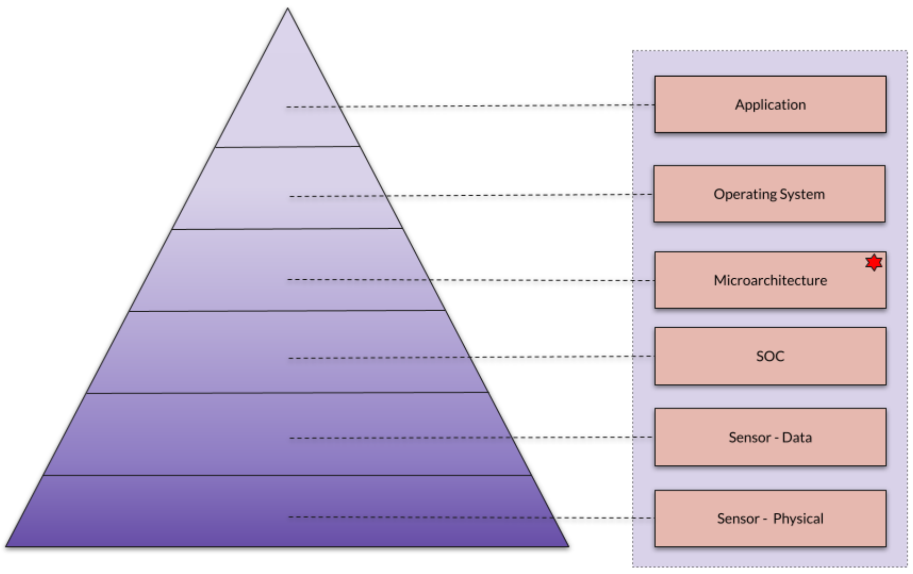


Figure 1.1: Example of a computation stack of a system. Each layer has distinct requirements and constraints, such as degree of protection, exposure, computation capacity, area, and energy costs. The Microarchitecture layer is the primary focus of this thesis.

the design effort to achieve good dependability and security levels must consider a per-block approach and be cost-proportional at each layer, not over-investing in a few of them and leaving the remaining blocks unprotected.

This inherent communication between blocks can be exploited to create and support co-design solutions, where two or more blocks cooperate to achieve protection. For example, by constraining the application layer to run integer-only calculations under a specific set of values, the microarchitecture can perform a fast and robust error detection mechanism through modular arithmetic [18].

To ensure a secure ML system design, confidentiality can be achieved with *hardware-assisted trusted computing solutions*, such as a Trusted Execution Environment (TEE) [19]. *Malicious human-made faults* can be introduced in the ML system to exploit vulnerabilities and give an intruder access to confidential information or even full-system control. TEE creates a walled garden ecosystem, isolating and ensuring that the *data-in-use* is protected. Still, attacks and vulnerabilities of some commercial TEE have been identified and exploited [20]. Therefore, additional layers of protection are of significant importance to safely deliver and run ML models on end devices.

Provably-secure cryptographic techniques could be applied to deal with this problem by encrypting the model's data and preventing unauthorized party accesses its information without a proper secret key. However, the encryption and decryption process would incur a high overhead, limiting the response latency and increasing the energy and area costs necessary to implement such schemes. Currently, the mainstream cryptographic method is homomorphic encryption (HE) applied to privacy-preserving of ML models, where all the required calculation is performed over encrypted data, with no need to decrypt it [21]. Yet, this solution makes the system at least an order of magnitude slower [22]. Therefore, applying cryptographic techniques over ML is still challenging, demanding a complex computational infrastructure.

On the other hand, non-cryptographic techniques can provide some degree of protection while not degrading the overall system behavior. Obfuscation has emerged as an appealing non-cryptographic approach to prevent model piracy attacks. The original model would not be released publicly but instead a disguised version. However, whoever had access to the obfuscated model, even a non-authorized user, would still be able to run it with no consequences.

Furthermore, the obfuscation approach was extended to protect from model exfiltration over end-user devices [7, 23–25]. This approach differs from the original one because whoever possesses the obfuscated model can only run it properly with a secret key and specific device. The "locked" model behaves erroneously depending on the key provided and the device, applying penalties to the final accuracy when used by non-legitimate users. Thus, this lightweight solution guarantees the model confidentiality and provides an additional authenticity layer to the system.

Once the model is deployed safely in the system, there is no guarantee that it will run correctly, with no interference of natural faults. These faults can originate either externally or internally by natural phenomena due to manufacturing defects [26, 27], extreme operating conditions such as voltage and temperature variations [28, 29] or even cosmic radiation [30]. Once active, a fault produces an error in the system that can be propagated, further generating a system failure. At worst, errors will be latent, propagating undetected through the system layers and generating wrong output results. The unpredictable consequences of silent data error (SDE) over a safety-critical ML system can range from a noisy prediction to a life-threatening output, such as the unintended acceleration of a car.

Computational error is a subset of errors that can threaten the system's reliability through the erroneous execution of computational units. Unlike errors in communication channels and storage systems such as network, main memory, and disks, these corrupt execution errors (CEEs) are hard to track and detect [13]. For instance, standard code-based techniques applied to storage and networking subsystems are impractical to implement over arithmetic units due to their per-instruction cost and complexity. Instead, lightweight arithmetic error codes should be explored to detect such errors [31].

Designing a reliable system that encompasses fault tolerance is an exercise in choosing appropriate redundancy strategies. Redundancy is the property of having additional resources (e.g., data) other than the ones minimally needed for the task completeness [18]. Redundancy can be applied in various ways, such as duplicating or triplicating the resource (e.g., dual or triple modular redundancy) or adding extra bits to data to verify its correctness further (e.g., coding). Arithmetic error codes, such as AN code, compose the set of coding techniques that can check the correctness of a set of arithmetic operations by encoding the original operands. The arithmetic operation result over the encoded operands should yield the same result as encoding the result of the original operation (i.e., operands without encoding). For example, in AN code [18, 31], the encode function consist of multiplying the operands by a constant A. Considering the multiplication operation over encoded integer operands, the outcome must be a multiple of A. Although arithmetic error codes provide greatly fault coverage [32], it still incurs a high overhead when applied to ML applications with billions of parameters and operations to be checked.

To summarize, Obfuscation and AN codes are effective methods to improve the security and reliability of systems within constrained resources. Moreover, both techniques can be applied at different system layers or even co-design within multiple layers. Although both mechanisms have been explored, there is still room, especially for ML systems, for improvements and novel ideas to reduce its cost of implementation while keeping the same degrees of security and reliability.

## 1.1 Contribution

This thesis proposes novel techniques and improvements over the state-of-the-art related to model obfuscation and error detection on deep neural networks microarchitectures. Additionally, a custom open-source DNN fault injection framework was created and shared through a GitHub repository during the research development. The main contributions are listed below:

- Extensive reliability analysis of compressed DNN models under the presence of transient faults. It is demonstrated that compression, in the mean of data quantization and model pruning, can dramatically increase the system's overall resiliency under a faulty condition (e.g., cosmic radiation).

- TorchFI, a custom DNN fault injection framework created on top of the classic and highly adopted DNN framework PyTorch. TorchFI can simulate transient, permanent, and intermittent faults over computational units and memory subsystems.

- Custom DNN error detection based on arithmetic error codes (AN code) that can detect more than 99% of errors over DNN MAC units. When coupled with a word-masking error correction scheme, the technique allows the DNN system to operate with no accuracy loss, even under high bit error rate scenarios.

- Novel AN code-aware quantization process that enhances the custom DNN error detection by removing the necessity of pre-multiplying one of the operands (e.g., input features or model parameters) by a constant A; further reducing the cost of an HW detection scheme through a co-design scheme (HW and SW error detection cooperation).
- Novel lightweight DNN model obfuscation technique that does not require a specialized training process to protect the model's IP while providing oblivious output class distribution.

## 1.2 Thesis Outline

The remainder of this thesis is organized as follows: Chapter 2 introduces the pertinent concepts related to Neural Networks, Reliability, and Security, required to understand the remaining Chapters fully. Chapter 3 presents TorchFI, the custom DNN fault injection simulation framework used during the reliability experiments. In Chapter 4, an extensive analysis of compressed DNN models under faulty conditions shows that data quantization and pruning significantly improve the system's overall reliability. Chapter 5 proposes AN code as a low-cost HW error detection scheme for DNNs microarchitectures. Further, in Chapter 6, the error detection scheme proposed in Chapter 5 is enhanced by a novel AN code-aware quantization process, removing pre-processing steps and reducing the detection costs. Chapter 7 introduces a novel obfuscation technique to protect the DNN from model IP theft attacks. Finally, Chapter 8 concludes this thesis by summarizing and discussing possible ideas for future work. Appendix A lists the accepted publications during the development of this thesis.

# Chapter 2

# Background

This chapter provides the relevant background information to better understand the addressed subjects in this thesis. The main topics are Neural Networks, Reliability, and Security of computer systems.

## 2.1 Neural Networks

Neural Networks are a set of machine learning algorithms where a computer is trained to perform some tasks by analyzing a collection of data (e.g., training samples). Its history goes back to the 1940s with Warren McCulloch and Walter Pitts's work [33], where they propose that artificial neurons with a binary threshold (step-function) were analogous to logical expressions. This would allow the creation of Boolean circuits by connecting a set of artificial neurons with each other. Inspired by the biological architecture and behavior of the brain, they modeled a neural system where each neuron is composed of soma (body, dendrites, and nucleus) and an axon. The neuron connections (i.e., synapses) are through the soma (dendrites) of one neuron and the axon of another one (Figure 2.1 - left). Each neuron would have its threshold to determine if an excitation generates a pulse or not. This pulse would be propagated from the soma to the axon and then received by all neuron's dendrites connected to the axon's terminals (Figure 2.1 - left). Depending on the placement, connections, thresholds, and inputs, the network would output a specific result analogously to a logical circuit.

Although the work proposed by McCulloch and Pitts was a vital contribution to the current neural networks' state-of-the-art scenario, it had limitations due to its simplicity. At the end of the 1940s, Donald Hebb [34] proposed what would be entitled Hebb's rule. The rule states that a connection between two neurons gets stronger when one repeatedly or persistently activates the other nearby. Hence, if neuron A axon constantly activates the neuron B axon, a growth process would occur, strengthening the connection between A and B and making the activation of

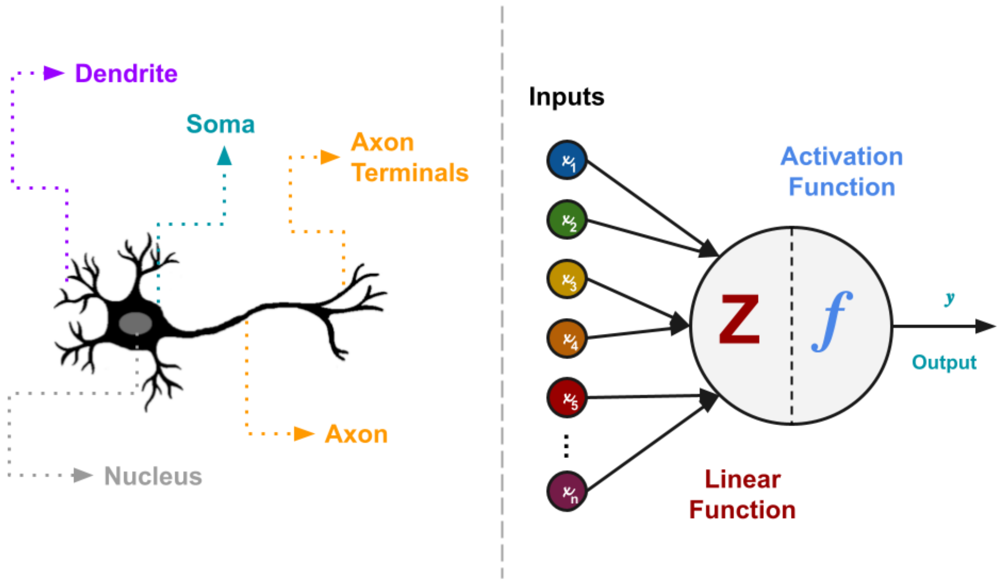


Figure 2.1: Example of a biological neuron on the left and an artificial neuron on the right.

neuron B by A more effective. Hebb's rule changed how neuron connections should be weighted and defined the fundamental procedures for the learning and memory process.

In the late 50s, the idea of a perceptron was introduced and demonstrated by Frank Rosenblatt [35]. It combined the McCulloch and Pitts neuron definition enhanced with Hebb's rule, enabling adjustable weighted inputs and thresholds (i.e., activation functions) in a single layer fashion (Figure 2.1 - right). The promising advancement of Rosenblatt's work made perceptrons capable of learning linearly separable patterns (i.e., linear classifier) but had many limitations that were further proved by Marvin Minsky and Seymour Papert [36]. Until then, research in the area was limited, and neural networks became dormant up to the 80s. Finally, the concept of neural networks was coined due to the placement and connections of artificial neurons. Inputs were connected to artificial neurons that were further connected to another set of artificial neurons. The multilayer concept (Figure 2.2) enabled neural networks to solve more challenging problems, surpassing the limitations pointed out by Minsky and Papert on Rosenblatt perceptrons. However, the growth in the number of nodes and layers of the network had its implications. The networks were now too complex to be solved in current state-of-the-art computer architectures.

Since the 80s, research works in neural networks have advanced, becoming a mainstream area over the past decade. The compute capability of current hardware, the amount of data, and the increasing interest and application of neural networks in the industry such as the internet, automotive, agricultural, manufacturing, defense, space exploration, etc., have enabled exponential growth and evolution of machine learning-based technology. The following sections will bring up a few definitions

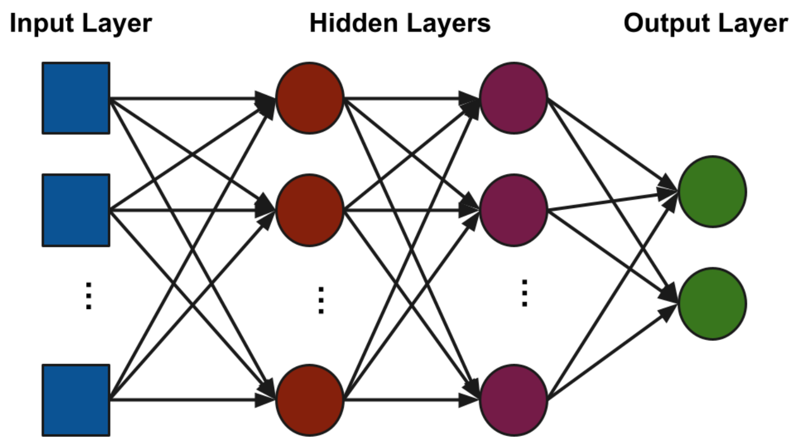


Figure 2.2: Example of a multilayer perceptron with two hidden layers.

that will be crucial for a better understanding of the thesis. Notwithstanding, neural networks are a vast area in constant growth, and the content in this section will not be sufficient for a complete understanding of the area.

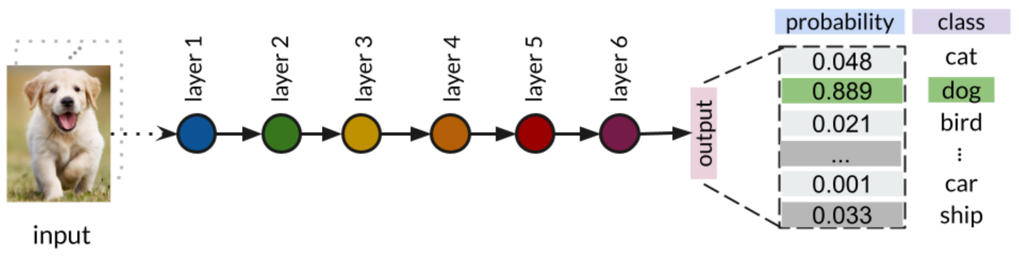


Figure 2.3: Example of a general six-layer deep neural network. Inputs are images that are processed by the layers. The network output is a list of probabilities of each class being classified.

### 2.1.1 Types of NN Architectures

The brain-inspired neural network research area has dramatically evolved since the invention of Perceptron by Rosenblatt in the 50s. New neural network architectures are proposed now and then to tackle various tasks such as computer vision, natural language processing, and recommendation systems with the highest accuracy they can achieve. Fjodor & Stephan's [37, 38] neural network zoo figure [1] shows some well-known architectures.

**Feed-Forward Networks (FFN)** are networks where the inputs flow through the hidden layer up to the output. Thus, there are no cycles in the network. As mentioned earlier, Perceptrons are an example of FFNs. Although they are composed of single (hidden) layers, the data flows through the network without any loopback.

**Convolutional Neural Networks (CNN)** are a specialization of FNNs, where the hidden layers are composed of convolutional operations over the data. This type of network is commonly applied to computer vision tasks since its functionality resembles the visual cortex behavior. Convolution operations extract features from the input images through several filters (Figure 2.4). These filters are small trainable weight matrices trained to extract and identify some patterns.

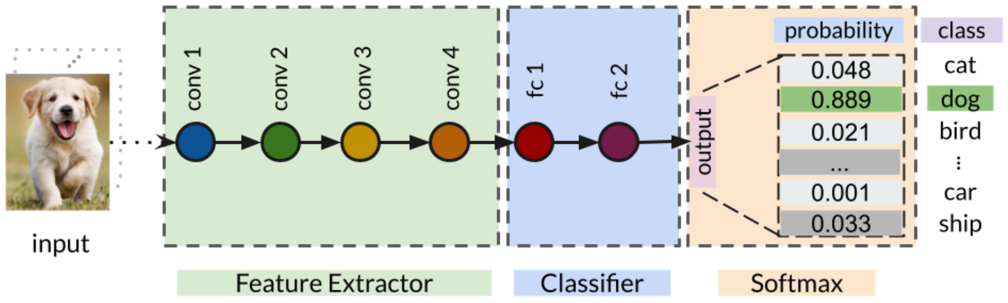


Figure 2.4: Example of a six-layer deep convolutional neural network. Convolutional layers act as feature extractors, extracting low and high-level features from the input images. The fully connected layers then use that information to classify the input. The output is a list of probabilities of each class that compose the classification pool of the network.

Due to the cardinality of different neural network architectures and areas in which they are employed, this work is constrained to Convolutional Neural Networks for Computer Vision problems. Therefore, the reliability and security assessment of the remaining NN architectures is out of the scope of this work.

Before diving into the details of Convolution Neural Networks, the machine learning concepts of supervised learning and backpropagation must be introduced.

[1]`https://www.asimovinstitute.org/neural-network-zoo/`

### Supervised Learning

Supervised Learning is a subcategory of machine learning algorithms in which the training step of the learning system demands a set of labeled data. The algorithm can be seen as an optimization problem where the main objective is to find a function that better fits the input data (e.g., image) to the output (e.g., class) based on previous examples. The training set is composed of pairs of input and output, where the output is the ground-truth label of the input. For example, in an image classification task, the input can be seen as pictures of animals and the output as the name of each animal (e.g., dog, cat, bird, etc.) defined as classes. During the training phase, as inputs flow through, the network must provide a way to measure how good or bad its prediction was. This is done by employing a loss function (i.e., cost function) that will map to a real number (loss) the scores achieved by each class in the classification set with that specific input. Then, the loss is fed back through the network so that each neuron can regularize its weights (i.e., update). This process is commonly known as Backpropagation.

### Backpropagation

Backpropagation is the algorithm that computes the gradient of the cost function (loss) with respect to the network's weights. Backpropagation is part of the training phase and takes advantage of gradient methods to efficiently perform weight updates. The update is accomplished through the means of the chain rule in a recursive way. Figure 2.5 shows how the computation of the gradients is performed using the backpropagation algorithm. Backpropagation makes the learning process of big networks feasible in terms of memory space and computation time.

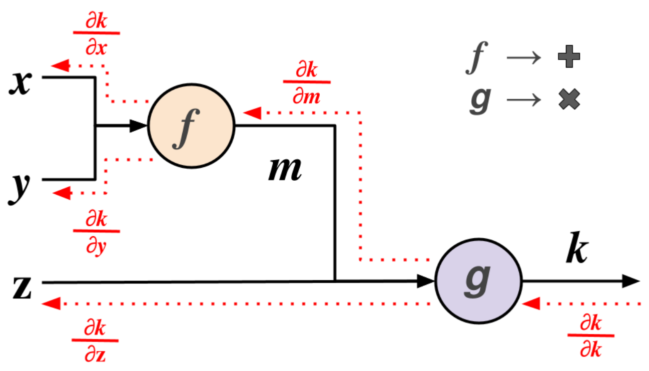

$\frac{\partial k}{\partial x}$
$x$
$y$
$\frac{\partial k}{\partial y}$
$z$
$\frac{\partial k}{\partial z}$
$f$
$\frac{\partial k}{\partial m}$
$m$
$f \rightarrow +$
$g \rightarrow \times$
$g$
$k$
$\frac{\partial k}{\partial k}$

Figure 2.5: Example of the backward propagation of $k$ over part of the computational graph. By recursively applying the chain rule, the gradients (in red) are computed starting from the end up to the beginning of the graph.

### 2.1.2 Convolution Neural Networks

Feed-Forward Networks are the base framework for several architectures. Convolutional Neural Networks compose one of its branches, created to handle images as inputs. CNNs are state-of-the-art for image classification, object detection, and segmentation tasks. Their architecture comprises (mostly) convolutional, pooling, and fully connected layers. The following sections will briefly describe the image classification problem and the concept of CNNs base layers.

#### Image Classification Problem

The image classification problem is a simple computer vision task with extensive real-world applicability. Given an input image, the task consists of identifying its content (e.g., object, person, etc.). Through supervised learning, the model should learn with the support of training pairs. Each pair is composed of one image and one label. Each label is part of the classification set, being unique identifiers or names of the classes. The training images are previously annotated so that the algorithm can check if the prediction matches the ground truth. The main objective is to classify correctly as many images as possible. Models are ranked via an accuracy metric, which is a percentual calculated by the number of correct predictions over the total. The standard way to evaluate models is through their top-1 and top-5 accuracies. Since the output of a model is the probability distribution of N scores, each representing a class of the training set, the top-1 accuracy consists of the correct predictions of the classes. In other words, the prediction with the highest probability was the ground truth class. The top-5 accuracy is calculated by considering the highest probability and the remaining top four ones. If the ground truth class lies in the top five probabilities by the model, the prediction will be counted as correct.

### 2.1.3 Use Case Architectures

Since there are numerous CNN models, the model family concept is employed. A family is a set of model variants that derivate from the same backbone. This backbone implements a specific characteristic or technique that enhances the CNN accuracy for some task over a particular dataset. Usually, families derive from older ones by enabling new features and achieving higher accuracies in the same or different datasets. One example of a feature could be the capacity of adding more layers (more than fifty) with no loss degradation. The variants differ in the number of layers, neurons per layer, and other hyperparameters such as learning rate and the number of epochs. The rule of thumb is to identify each model with its family name followed by the number of hidden layers (e.g., ResNet50). The following sections will describe four model families employed as use cases during the development of this thesis. The models are LeNet, AlexNet, VGG, and ResNet. They range from small and proof-of-concept networks to deep and state-of-the-art ones.

#### LeNet

LeNet is a family of small-size convolution neural networks developed by LeCun et al. [39, 40] and designed for handwritten digits classification tasks. The five-layers deep LeNet (a.k.a. LeNet-5) was primarily deployed to identify zip code numbers from the US Postal Service and successfully achieved the task. Still, the model and its variations are used for experimentations, and proof of concept with databases like MNIST (handwritten digits database) [39] [2].

#### AlexNet

AlexNet is a single model family composed of eight hidden layers, five convolutions followed by three fully connected ones [8]. The AlexNet model was introduced in the 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) with a disruptive architecture with high accuracy, and that was able to run over GPUs. In 2015, the eight-layers deep AlexNet model won the ILSVRC, beating Microsoft Research Asia's CNN with 100 layers. Since then, AlexNet is still used as a mid-size validation network.

#### VGG

VGG is a family of very deep convolutional networks [9]. The name stands for the time who won the 2014 ILSVRC on the classification and localization tasks. The family is composed of 16 and 19-layers deep networks. VGG's networks were considered deep at its creation due to their number of layers and their massive number of parameters (e.g., 138M for VGG16).

#### ResNet

ResNet is a family of models created for a specific purpose: to enable deeper architectures [10]. Before ResNet, deeper models (bigger than the 22-layer deep GoogleNet [41]) struggled to sustain their accuracy during training time due to the vanishing gradient problem. As the gradient backpropagates from the final layers to the early layers, the repeated number of multiplications by small numbers makes the gradient infinitely small. A deeper network means a better approximation of the function (to fit the data) [42]—still, more layers to backpropagate, resulting in the vanishing

[2] `http://yann.lecun.com/exdb/mnist/`

gradient over early-stage layers. ResNet was proposed with a simple but ground-breaking idea to avoid the vanishing of the gradient. ResNet stands for residual network, and the core idea is to add an "identity shortcut connection" that bypasses some layers. This concept works because the network should be able to approximate an identity function. If ones bypass the set of hidden layers by a shortcut that connects the input to the output, the network will still move towards an approximate function that fits the data. The vanishing gradient problem is then solved (or mitigated) through the identities shortcuts that flow the gradients backward, hopping some layers and avoiding multiplications that would eventually cause the gradient to be 0 (or extremely close to it). Currently, state-of-the-art ResNet models can achieve up to 152-layers deep, and its smallest model has 18 layers.

### 2.1.4 Use Case Datasets

#### MNIST

The handwritten digits dataset MNIST has 60,000 training samples and 10,000 validation samples. The images are small-size grayscale ones with 28x28 pixel size box representing values ranging from 0 to 9 (ten classes in total).

#### CIFAR10

The CIFAR-10 dataset comprises 60,000 small-size color images with 32x32 pixel box and a total of ten classes. Each class represents an object, such as cats, dogs, ships and trucks. The training set consists of 50,000 images and the validation set of 10,000.

#### ImagetNet

The 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) dataset is composed by millions of color images. The training set has $\approx$ 1.2 million samples and the validation set has 50,000 images. The dataset has a 1,000 unique classes and images with 224x224 pixels.

### 2.1.5 Compute Optimization Techniques

Neural network applications demand massive computation and memory space. Even small models, such as LeNet, with a couple of layers, can require millions of floating-point operations (FLOPs) and megabytes of data (e.g., weights) to be stored and loaded from memory [3]. Compute optimization techniques can be applied to make deep neural networks feasible to run in resource-constrained devices. Techniques such as quantization and pruning reduce the memory footprint and energy necessary to compute such applications and, most importantly, without affecting their final accuracy. The following sections will detail some state-of-the-art quantization techniques and network pruning.

## Quantization

The quantization concept arises from the signal processing area where continuous infinite values are mapped into a smaller set of discrete finite values. The same idea is applied to ML data quantization, where the data represented in a higher precision data type is mapped to a lower precision data type, as exemplified in Figure 2.6. A straightforward quantization process (Equation 2.1) can be achieved by applying a scaling factor ($s_W$) to the input data($W_{float}$). The scaling factor will act as a mapping function that will transfer the input data to a new output space as exemplified by the unsigned 8-bit integer quantization in Equation 2.2.

$$W_{float} \approx s_W * W_{int} \tag{2.1}$$

$$W = \begin{pmatrix} 0.57 & 0.0 \\ 1.0 & 0.81 \end{pmatrix} \approx \frac{1}{255} \begin{pmatrix} 145 & 0 \\ 255 & 207 \end{pmatrix} \tag{2.2}$$

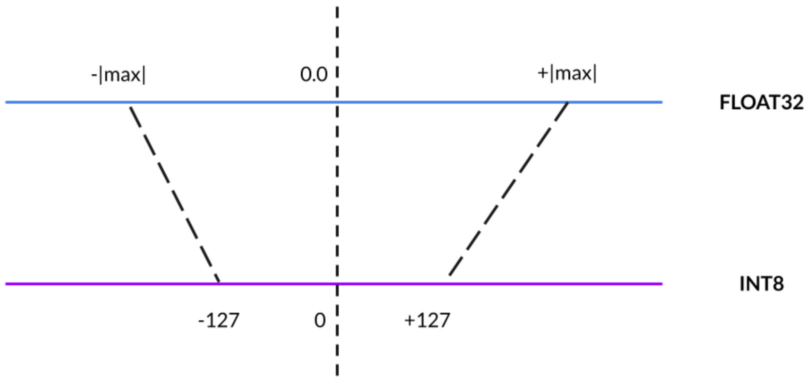


Figure 2.6: Example of a signed 8-bits symetric quantization (i.e., zero-centered) over 32-bit floating-point data.

The IEEE 754 single-precision floating-point (FP32) is the standard data type adopted for activations and weights during a neural network's training and inference phases. Although FP32 provides a wide range and good precision (Figure 2.8), the amount of energy dissipated to compute and transfer the data from the main memory to the floating-point unit limits the potential use of neural networks over constrained resources (e.g., smartphones). Additionally, floating-point units demand a far more extensive area when compared to integer ones, limiting the feature space of the chip design.

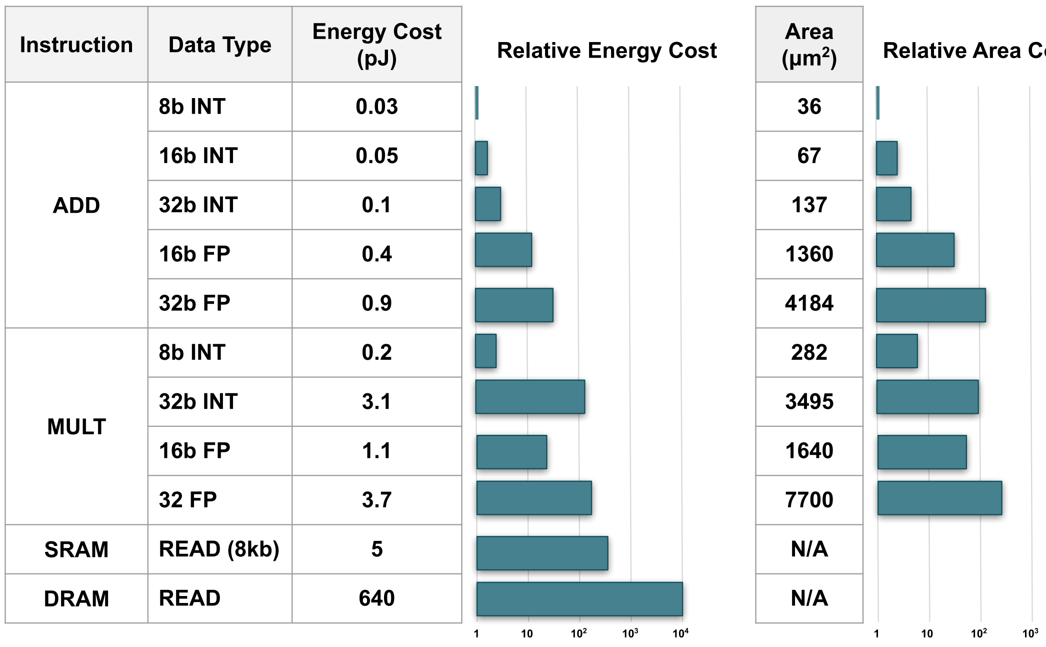
Figure 2.7, adapted from Sze's tutorial slides on DNN Model and Hardware Co-Design [1] and data collected from Horowitz [2], depicts the amount of energy necessary to transfer or compute each data type. Data movement dominates the energy cost, especially when transferring 32-bit data from the main memory (DRAM). In the arithmetic operations spectrum, floating-point data adds up energy cost on top of integers and demands a higher circuit area. By moving from 32-bit to 8-bit data types, the costs in energy and area decrease by orders of magnitude. Therefore, quantization can be the inflection point to enable state-of-the-art deep neural networks on low resources devices by reducing the area necessary for the circuitry and the amount of energy needed to compute and transfer the data from the memory to compute units.

Since Krizhevsky's publication of AlexNet [8], several research works around data quantization for neural networks have been proposed. Although the quantization process incurs an approximation of the data, CNNs showed prominent resiliency to low-precision quantization, such as 8-bit integer (INT8), maintaining the same final accuracy as original FP32 versions. Additionally, the industry has proposed and adopted novel and application-aware data types as new standards. For example, Google's Brain-Float (BF16) is a 16-bits data type proposed primarily for neural network applications. BF16 takes advantage of the wide range of FP32 by having the same amount of bits dedicated to the exponent part (8 bits) but with lower precision than the IEEE 754 FP16 (7 bits), as detailed in Figure 2.8. This makes the training of DNNs in a lower precision feasible without hurting the weight's update during the backpropagation algorithm. Major hardware vendors have adopted BF16 as a standard for training DNNs and incorporated it in their new Instruction Set Architectures (ISAs) [43, 44].

Quantization can either be symmetric or asymmetric. In the symmetric quantization, the zero in the mapping space is fixed over the zero of the original space (Figure 2.9 – left plot). Asymmetric quantization offers more flexibility by including an extra variable (zero-point) in the scaling factor equation. The zero-point ($z$) variable shifts the grid and changes where the zero value will be placed in the mapping space (Figure 2.9 – right plot), allowing a better fit of a different set of distributions.

When applied to neural networks, quantization can be achieved through two main classes of algorithms, Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). But before diving into PTQ and QAT details, such quantization classes' development and testing process should be clarified.

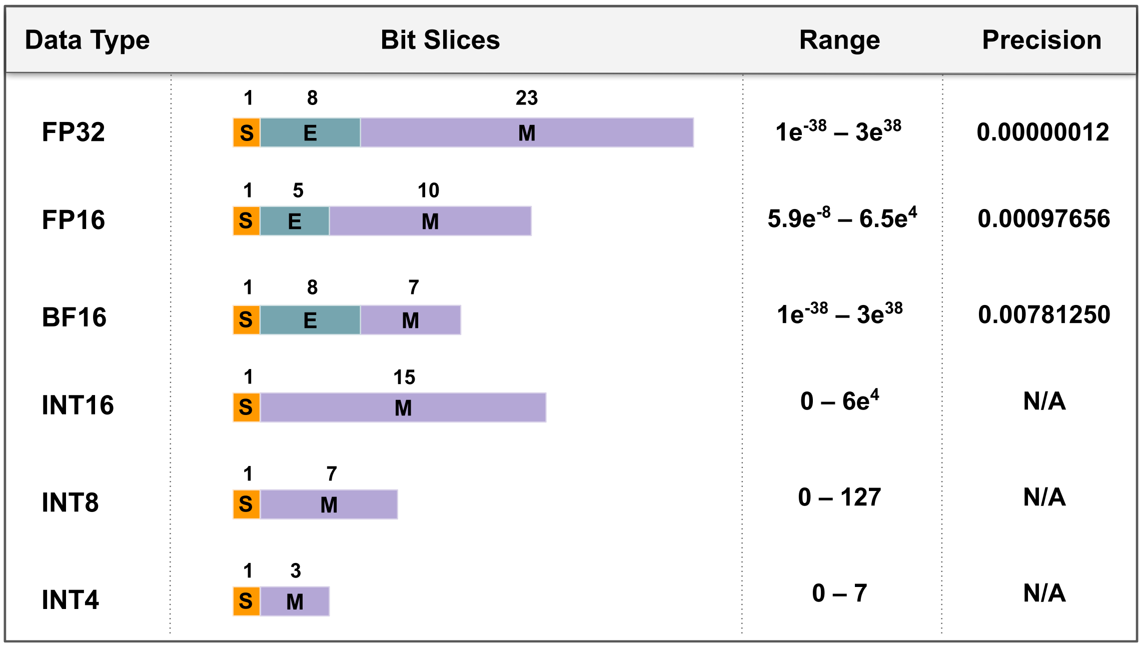
Quantized neural networks are developed and tested through the support of



| Instruction | Data Type | Energy Cost (pJ) | Area (μm²) |
|---|---|---|---|
| ADD | 8b INT | 0.03 | 36 |
| | 16b INT | 0.05 | 67 |
| | 32b INT | 0.1 | 137 |
| | 16b FP | 0.4 | 1360 |
| | 32b FP | 0.9 | 4184 |
| MULT | 8b INT | 0.2 | 282 |
| | 32b INT | 3.1 | 3495 |
| | 16b FP | 1.1 | 1640 |
| | 32 FP | 3.7 | 7700 |
| SRAM | READ (8kb) | 5 | N/A |
| DRAM | READ | 640 | N/A |

Relative Energy Cost
1
10
10²
10³
10⁴
Relative Area Cost
1
10
10²
10³

Figure 2.7: Example of the amount of energy necessary to transfer or compute each data type. The diagram was adapted from Sze's tutorial slides on DNN Model and Hardware Co-Design [1] and data collected from Horowitz [2]. Reproduction authorized by the authors.



| Data Type | Bit Slices | Range | Precision |
|---|---|---|---|
| FP32 | 1 8 23<br>S E M | $1e^{-38} – 3e^{38}$ | 0.00000012 |
| FP16 | 1 5 10<br>S E M | $5.9e^{-8} – 6.5e^{4}$ | 0.00097656 |
| BF16 | 1 8 7<br>S E M | $1e^{-38} – 3e^{38}$ | 0.00781250 |
| INT16 | 1 15<br>S M | $0 – 6e^{4}$ | N/A |
| INT8 | 1 7<br>S M | 0 – 127 | N/A |
| INT4 | 1 3<br>S M | 0 – 7 | N/A |

Figure 2.8: Example of different data types widely adopted in the Machine Learning area. The diagram was adapted from Sze's tutorial slides on DNN Model and Hardware Co-Design [1]. Reproduction authorized by the authors.

simulations. The layer's operations are performed in FP32, but data is pre-processed before and after each computational block. This back-and-forward (FP32 → INT8 → FP32) step allows the network to simulate the effects of error approximation. Figure 2.10 details a simulation of an asymmetric quantization process between two

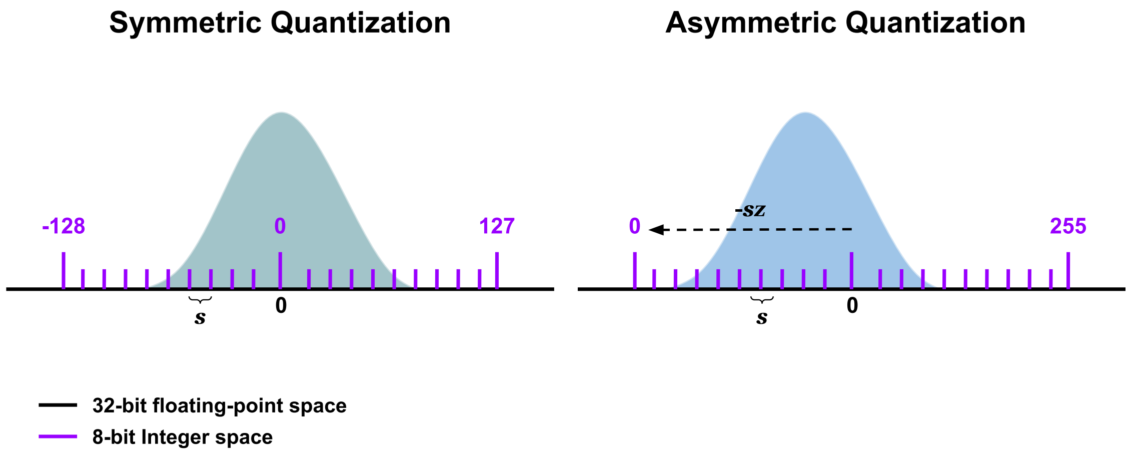


Figure 2.9: Example of differences between symmetric or asymmetric quantizations. Asymmetric quantization shifts the grid by setting a new zero-point ($z$) differently from the original one, providing more flexibility to fit the data distribution.

convolutional layers. Considering $s$ as the scaling factor, $zp$ as the zero-point, $b$ the number of bits to quantize, and $W$ the weight matrix, the quantization blocks perform the Equation 2.3 operation followed by Equation 2.4 to scale back the output to a floating-point matrix. This process approximates the behavior of the network running on actual hardware that provides support with the desired data type.

$$W_{int} = clip\left(round\left(\frac{W_{float}}{s}\right) + z, min = 0, max = 2^b - 1\right) \tag{2.3}$$

$$\hat{W}_{float} = s\left(W_{int} - z\right) \tag{2.4}$$

**Post-Training Quantization** is a class of quantization algorithms that take a pre-trained network model in a single-precision floating-point format and convert it into a quantized model (fixed-point or integer format). The process does not require any retraining of the model and only a few calibrations (or even none) of the quantization functions to be applied (Equation 2.3). Weights are quantized before the network execution, but input activations demand on-the-fly computation.

Quantization parameters, such as scaling factor and zero-point, can be tunned by providing a few input samples in a non-mandatory calibration phase. The clipping parameters ($q_min$ and $q_max$) and the rounding algorithm of Equation 2.3 should be carefully set and are strongly tied to the level of granularity at which the quantization function is applied. A naïve level of granularity can consider the whole network data, but that will lead to a high uncorrelation with the weight distribu-

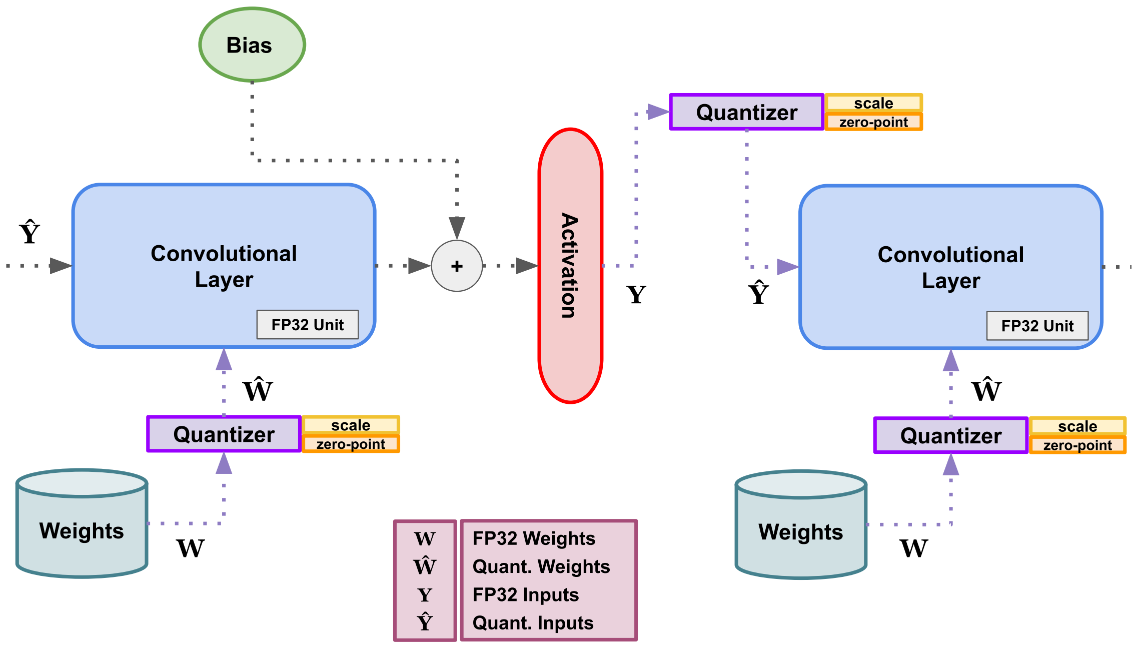


Figure 2.10: Example of how quantized neural networks can be simulated over standard hardware. These steps allow the network to mimic the effects of error approximations that arise from quantization functions.

tions presented in each layer. Common approaches are layer-wise, tensor-wise, and channel-wise quantization, where the quantization functions will be crafted considering the distribution of each layer, tensor, or even more profound, the distributions of each weight's channels inside the tensor's layers. By carefully picking the right granularity and the parameters, the amount of error incurred by the clipping and rounding functions will be minimized (or balanced), leading to lower degradation of the model's accuracy.

Although PTQ provides a "plug-and-play" capability, it presents poor accuracy when the number of bits (quantization levels) is low. The current state-of-the-art PTQ algorithms can handle up to INT8 data types with marginal accuracy degradation [45]. Still, PTQ is largely adopted due to its near-zero cost for quantizing the network.

As the name suggests, **Quantization-Aware Training** involves the model's training pipeline to achieve enhanced quantization functions. This fact implies a more challenging task: handling the backward propagation of the gradients through the quantization blocks, especially the rounding function (no meaningful derivative). The QAT research area can be subdivided into approximation-based, and optimization-based techniques [6]. The approximation-based method tries to solve the issue by applying a straight-through estimator (STE). The STE replaces the zero rounding derivatives with an identity function. Moreover, approximation-based techniques provide the means to learn Equation 2.3 parameters, such as scaling factor ($s$) and quantization levels ($q_{min}$, and $q_{max}$), finding the optimal balance between rounding and clipping error [46–48]. However, the STE ends up leading to the gradient mismatch problem since the approximation of the values during the forward pass differs from the one applied in the backward pass, incurring a noisy training phase [49].

The optimization-based techniques alleviate the gradient approximation problem by employing soft quantizers [6, 49, 50]. The soft quantizers are learnable differentiable non-linear functions that can be applied to any data of the network (weights and activations). The functions are optimized through several finetune epochs by minimizing the ideal and the soft quantization function gap. This is a similar process to the training of the network weight, but now the targets are parameters of the quantization function.

Both approximation-based and optimization-based techniques can achieve reliable accuracies under ultra-low quantization levels (e.g., 1 to 4-bit). However, QAT implies a heavy training process and demands a deep knowledge of the networks to better adjust its parameters.

### Pruning

In the 1980s, Le Cun et al. [51] proposed an iterative process to reduce the size of neural networks by removing irrelevant weights. The weights were classified and sorted through sensitivity analysis (i.e., second derivative) based on their impact on the training error. The technique entitled pruning had as main objectives to achieve a better generalization and improve the speed of learning/classification. However, the pruning idea ended to be much more beneficial than the original proposal. Removing unnecessary weights can drastically reduce the amount of space needed in memory within the constant increase of the current state-of-the-art deep neural networks. Additionally, the cost of transferring the data back and forward from the DRAM to the SRAM and then the compute units is reduced, saving a valuable resource for resource-constrained devices: energy.

Han's work [3, 52] followed up the pruning idea by addressing the sensitivity analysis to define better how the thresholds should be set. A per-layer empirical study was performed, and the findings show that smaller thresholds must be applied to the most sensitive layers (e.g., the first convolutional layer). Therefore, a correlation between layer sensitiveness and pruning rate should be considered. Through an iterative pruning process over pre-trained networks (Figure 2.11), the authors reduced AlexNet and VGG16 parameters by a factor of 9x and 13x, respectively. These results account for more than 80% of parameters pruned and, more importantly, without accuracy degradation on both models.

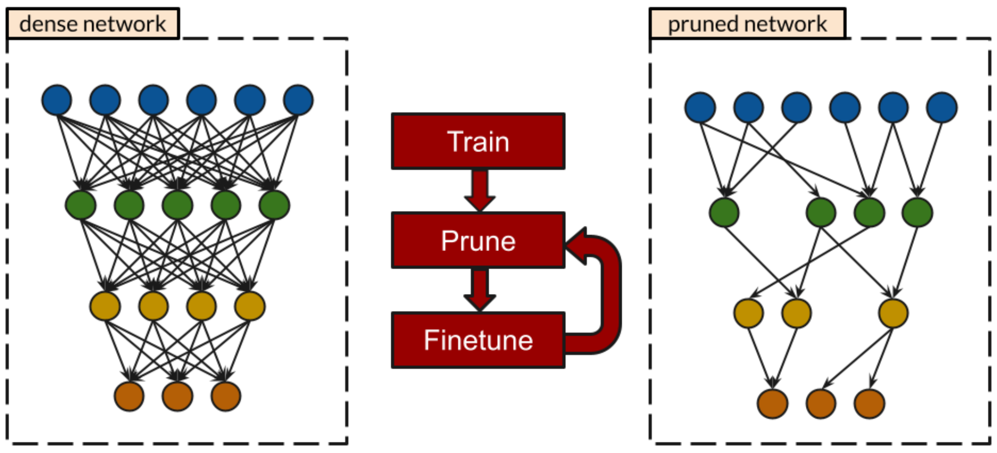


Figure 2.11: Steps of an iterative pruning process. Dense models are transformed into sparse ones by removing non-relevant nodes and connectivities (i.e., weights).

The pruning results were further enhanced by combining it with quantization [52] and by applying an Automated Gradual Pruning (AGP) [53] which removes most of the weights at the initial iterations and gradually reduces the pruning rate over time.

## 2.2 Reliability

Reliability is one of the six fundamental pillars to building dependable and secure computer systems. It comprehends the ability of the system to preserve its correct execution under a faulty condition or attack scenario. Fault-tolerant systems can manage these circumstances by acknowledging that the system has encountered errors and still operates with no failures and an acceptable quality of service. The system must provide ways to detect, mitigate, and, if possible, correct the errors that manifest from permanent, transient, and intermittent faults. This fault-tolerance can be achieved through redundancy. Extra hardware, software, data, or use of time can be added to the system to overcome such faulty scenarios and increase the system's overall reliability. This section reviews basic definitions of faults, errors, and failures and means to detect, correct, and mitigate them. Additionally, techniques to simulate faulty scenarios and some historical evidence of such faults and errors in computer systems are described below.

### 2.2.1 Terminology

First, a few terminologies should be defined beforehand since some terms are used interchangeably and are not absolute [12, 54].

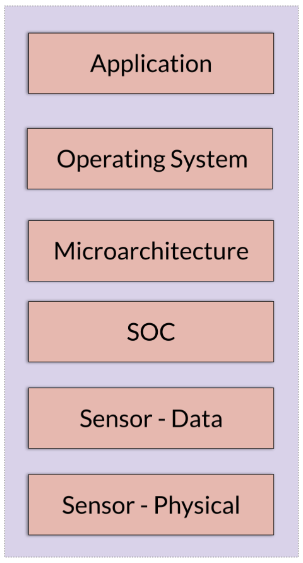


Figure 2.12: Example of a computational layer stack of an autonomous driving system.

- *Faults* are caused by internal or external factors that manifest in the hardware component or software modules. They can arise from chip manufacturing defects, extreme operational conditions, software bugs, or even natural phenomena that interact with the electronic device. Examples of events that cause faults are radiation-induced (e.g., cosmic rays and alpha particles), metal and oxide failures (e.g., electromigration and gate oxide wearout), voltage, and temperature variation. Faults can assume two types of modes during the system's execution. When active, the fault has manifested as an error in the system. However, this fault manifestation (i.e., error) will only be user-visible if it reaches the highest level of systems abstraction layers (e.g., the Application layer in the autonomous driving system depicted in Figure **??**). Moreover, many faults do not manifest as errors and remain dormant during the whole execution of the application.

- *Errors* are consequences of faults in the system. Whenever a fault is triggered, moving from dormant to active, it manifests as an error. As defined by Mukherjee [54], the definition of error is strongly tied to the notion of scope. If an active fault occurs in a specific scope, it will become an error in that scope. If the fault is masked or tolerated, it may not appear in an outer scope as an error. Also, faults detected within the scope will become an error in that scope. Outer scopes will be affected by errors in inner scopes only if the error is not corrected within the inner scope. Therefore, once a fault becomes an error and the error is not corrected, it can propagate through the outer scopes and become a user-visible error. Consider the following example provided by Mukherjee [54]. Suppose a fault occurs in the memory subsystem, such as a bit-flip in the most significant bit. This fault will eventually become a user-visible error if active (i.e., memory location read by the system). Consequently, the scope of the error will be the whole system, since it has been propagated to the last possible level (e.g., user application). However, if the memory subsystem is protected by some redundancy scheme, such as error-correcting code (ECC), the error will be contained and will not propagate. Therefore, the error is considered to have occurred within the scope of the ECC logic. For the remaining scopes outside this logic, the error will be deemed to be tolerated.

- *Failures* mean that the system cannot deliver the expected quality of services, such as correctness or performance. They are considered particular cases of errors. When errors propagate, reaching the utmost boundaries of the system and becoming user-visible, they can lead to system failures. An example of failure could be a multiple bit-flip in the memory subsystem that stores a database. This data corruption can lead to unexpected behavior (e.g., wrong information delivered) of the database system and the services that depend on its data.

### 2.2.2 Types of Faults

Faults (and errors) can be classified into three types. Permanent, transient, and intermittent. They are classified into each type due to the nature of manifestation.

- *Permanent* types of faults or errors constantly occur until the replacement or correctness of the component. Device wear-out (e.g., Oxide wear-out) or manufacturing defects that are not detectable during testing exemplify how permanent faults and errors manifest. Permanent errors are also called hard errors due to the physical nature of the issue.

- *Transient* or *soft errors* have non-deterministic occurrences in terms of time and location. They typically manifest into single or multiple bitflips or gate malfunction in the system at a random place. Transient types commonly manifest due to external factors like alpha particles or neutron strikes.

- *Intermittent* types tend to occur repeatedly but with periods of no manifestation. They could be considered indicators that will evolve into a permanent behavior. They can harm the hardware by generating errors that last one or more clock cycles [55, 56]. Due to its non-deterministic activation nature, intermittent types of faults and errors are hard to diagnose. Unlike transient, intermittent types tend to occur repeatedly in bursts at the same location.

### 2.2.3 Source of Faults

Faults can manifest due to several factors that can be either internally or externally to the system. The following will describe a few of them directly related to each type of fault over semiconductor devices, such as Radiation-induced, Operational conditions, and Hardware wear-out.

#### Radiation-induced

Radiation-induced transient faults compose the extensive list of possible source/threats to any system that runs over a semiconductor device. These faults are introduced into the silicon via two potential particles sources. The first one arises internally due to radioactive impurities used in the chip packaging process [57]. The second occurs externally, via the terrestrial atmosphere, in the form of cosmic rays (i.e., terrestrial cosmic rays). These faults are considered “soft” or “transient” due to their spontaneous nature. Unlike “hard” ones, which are permanent, typically arise from materials wear-out and forcing the device replacement, soft ones are nonrecurring, at a random location, and can result in single or multiple bit flips.

Alpha particles (a.k.a. alpha radiation or alpha rays) are emitted through a radioactive decay process that arises from the nucleus of radioactive components. When an alpha particle is sufficiently energetic and ionized, it can penetrate the die surface, creating electron-hole pairs in the substrate. If this alpha particle hits the right place (i.e., storage well), the electron-hole pairs will cause a bit-flip. The first report of this kind of fault was reported in 1978 by Intel Corporation [57]. Intel observed an unusual behavior of its 16K DRAM, where bit-flips were occurring with no apparent cause. Intel’s May and Woods further investigated the problem and found that it was due to the ceramic packing of their chips. The packing modules, manufactured in a new plant constructed upstream of Colorado’s Green River, got contaminated. Unluckily, the manufacturing plant was built near an old uranium mine, where the radioactive atoms end up being transported from and through the river. Even a slight trace of uranium and thorium in the packaging material was enough to produce alpha particle strikes, causing bit-flips into random locations of the memory chip [54]. During Alpha-Flux experiments, May and Woods found a strong correlation between the chip's error rate (i.e., number of bitflips) and the intensity of alpha particles the chip is exposed to.

Furthermore, IBM Corporation suffered from the same source of transient faults. Over a year of production (1986-1987), IBM chips were contaminated by radioactive materials coming from a chemical plant, far from IBM's manufacturing plant, but part of the manufacturing chain of the chips. The bottles that stored acid used during the chip manufacturing were the source of contamination. The chemical plant used radioactive material to clean the bottles that were used then by IBM [54].

Unlike alpha particles, neutron strikes are consequences of external interference arising from space. Cosmic rays are high-energy radiation particles, possibly as a result of supernovas and black holes [58], that penetrate the earth's atmosphere. However, as they make their way to the earth's surface, these particles end up colliding with air molecules, creating new particles. These new particles are then broken into other particles, such as Neutrons, Protons, Electrons, etc. This "shower" effect of several particles strikes the earth's surface and, consequently, hits semiconductor devices [59]. Like alpha particles, when the neutrons are highly energized, they interact with the semiconductor material, generating sufficient bursts of electron-hole pairs to interfere in the electronic circuit components (i.e., soft faults). Also, it has been observed that the atmospheric neutron flux varies enormously due to altitude, magnetic latitude, time of day, solar cycle, etc. [59], increasing the effort necessary to detect, protect, and mitigate from such faults not only at sea level systems(e.g., supercomputers) but also in high altitudes systems (e.g., aircraft) [60]. The first report of failures in space-borne digital systems due to cosmic ray interaction was reported by Binder, Smith, and Holman in 1975 [61]. The unexpected triggering of flip-flop circuits caused anomalies in the communication system of a satellite. Further, in 1979, Ziegler and Lanford decided to investigate the occurrences of "soft fails" due to cosmic rays. After discovering by Intel Corporation that soft faults arose from alpha particles in their packing system, Ziegler and Lanford suspected that cosmic-ray particles at sea level could impact computer memory devices. IBM Corporation validated the data and confirmed the occurrence of soft errors in computer chips due to cosmic rays [62].

### Hardware Wear-out

Permanent faults that affect semiconductor devices can be classified into two types: extrinsic and intrinsic. Extrinsic types arise due to manufacturing defects, and they manifest and affect the hardware component at the early stages of the component's life. Detection of such defects is done through several tests in the fabrication pipeline, such as burn-in. The technique's name reflects the process where chips are exposed to high temperatures and voltages. The ones that present traces of permanent defects are discarded and considered part of the infant mortality phase.

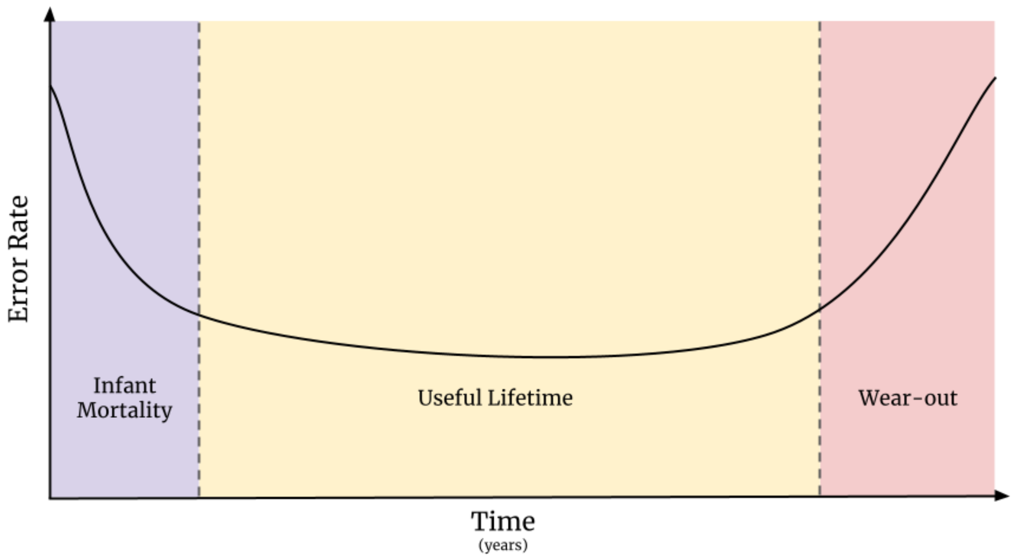


Figure 2.13: Example of a *bathtub-curve* showing the expected error rate of a hardware component over time.

In contrast, intrinsic faults arise over time, with hardware degradation, due to several factors ranging from the age of the component to environmental factors (e.g., ambient temperature) to which the device has been exposed over its useful lifetime. In their end-of-life phase, the materials that compose the silicon device begin to wear-out, leading to transistors malfunction and a permanent faulty behavior of the chip. The *bathtub-curve* depicted in Figure 2.13 captures how the error rate of extrinsic and intrinsic faults are related over the lifetime period of a semiconductor device.

Hardware wear-out can manifest through several intrinsic fault models, such as metal or gate oxide failures [54, 63]. One example of metal failure mode is the *Electromigration* mechanism 2.14 caused by the electron flux that flows through the metal lines and is deteriorated under high temperatures. Electrons flow through the metal lines and collide with the metal atoms that can eventually be displaced. Within sufficient momentum, the metal atoms are dislocated, creating a void in the metal line and becoming part of an extrusion area (Figure 2.14). A closed circuit is created if the amount of metal atoms displaced over time is enough to make the extrusion area connect an adjacent line. Additionally, an open circuit can be formed if the number of displaced atoms is sufficient so that the void is greater than the metal line area. Over CMOS gates, this behavior leads to stuck-at types of faults, where the transistor results will be stuck at 1 or 0.

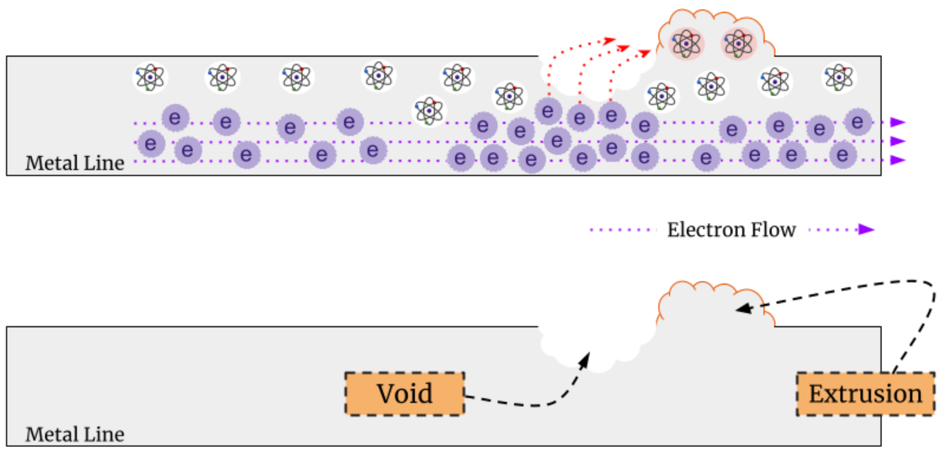


Figure 2.14: Example of an *electromigration* mechanism caused by the high electron flux that flows through the metal lines. The metal atoms are dislocated within sufficient momentum, creating a void in the metal line and becoming part of an extrusion area.

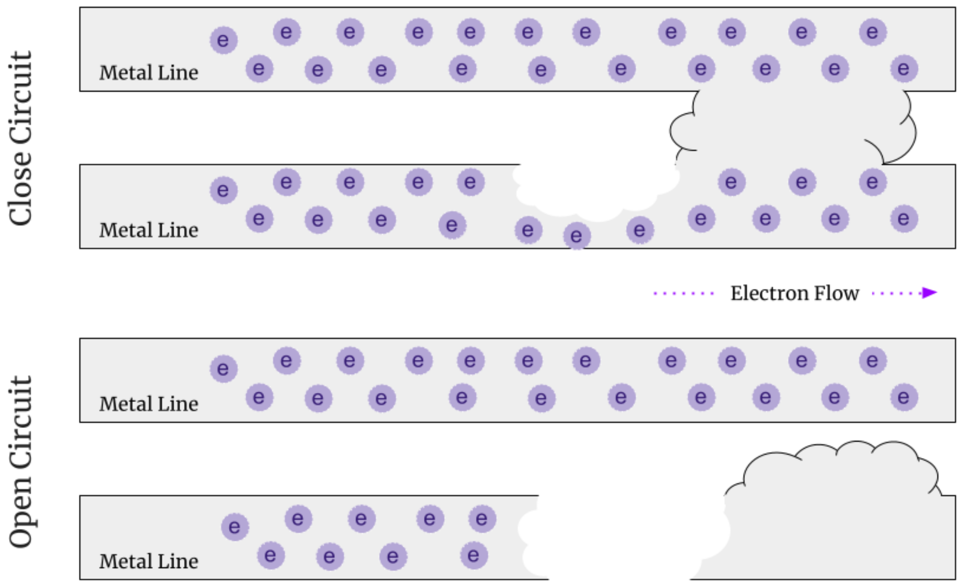


Figure 2.15: Examples of open (top) and closed (bottom) circuits that were caused by significant extrusion and void areas, respectively.

Gate oxide failures are also a form of transistor wear-out. CMOS transistors (Figure 2.16 are composed of:

- **Gate:** control terminal of the transistor.
- **Source and Drain:** terminals where electrons are provided and collected,

respectively.

- **Substrate:** crystalline conductor between source and drain.
- **Gate Oxide:** thin, non-crystalline, amorphous silicon dioxide that insulates the gate from the source, drain, and substrate.

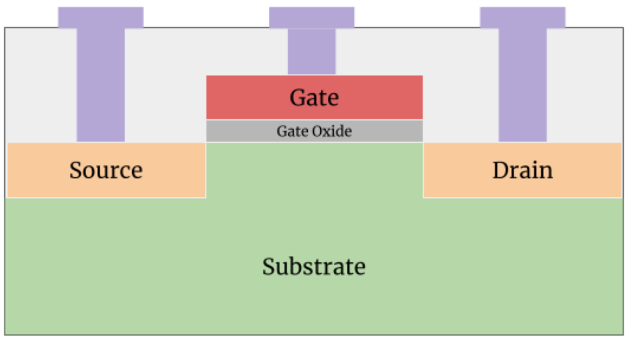


Figure 2.16: Example of a CMOS transistor.

Depending on the type of the transistor (nMOS or pMOS), the substrate is composed of a low concentration of negative (nMOS) or positive (pMOS) charges. The source and drains are areas with a high concentration of negative (nMOS) or positive (pMOS) charges. Considering an nMOS transistor, when a ground charge is applied to the gate, the negative charges will be repelled (Figure 2.17 – bottom plot), acting as an open circuit and avoiding electron flow from source to drain. The substrate charges will be attracted by positive voltage (V+) to the gate, creating a closed circuit, enabling electron flow from the drain to the source. Seamlessly, a pMOS transistor will allow the electron flow with the ground applied to the gate and disable with positive voltage as detailed in Figure 2.17 (top plot).

The gate oxide performs the crucial task of insulating the gate from the substrate, source, and drain. Also, the thickness of this layer dictates the switching speed of the CMOS transistor (on to of or vice-versa) [54]. High-performance chip designers decrease the supply voltage, demanding ultra-thin gate oxide layers to maintain overall power consumption in every new technology generation. This leads the gate oxide to a potential point of failure in a transistor.

When the maximum operating frequency is somehow affected, a Hot Carrier Injection (HCI) [64] can occur, affecting its transistors. HCI happens when the VDS (voltage drain-source) is exceptionally high, causing the income electrons from the source to arrive at the drain at high velocity and with high kinetic energy.

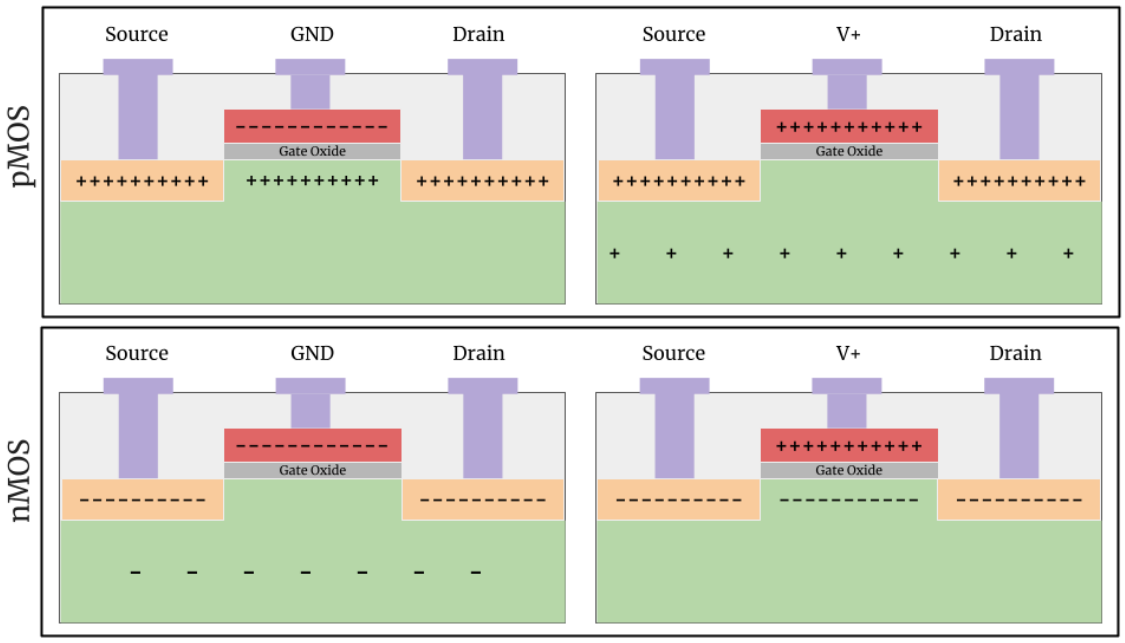


Figure 2.17: Examples of pMOS and nMOS transistors where the substrate is composed of a low concentration of positive (pMOS) and negative (nMOS) charges. When a ground charge (GND) is applied to the gate, the negative charges will be repelled (bottom plot), acting as an open circuit and avoiding electron flow from source to drain. The closed-circuit occurs when a positive voltage is applied to the gate, enabling electron flow from the drain to the source. The reverse happens with the pMOS (top plot).

By striking the atoms in the drain-substrate interface, the hot-carrier phenomenon can cause electrons-hole pairs, making some carriers bounce back to the substrate, causing an increase in substrate current. In the worst scenario, some carries can go through the gate oxide, degrading the insulation layer or even reaching the gate (Figure 2.18). HCI can lead to instability (intermittent fault) or even degradation (permanent fault) of the transistors in the silicon chip.

### Operating Conditions

Intrinsic faults can also arise due to operating conditions such as voltage and temperature fluctuations or due to crosstalk and switching noise interference [63]. Dynamic voltage and frequency scaling (DVFS) is a feature that enables the dynamic adjustment of voltage and frequency of CPUs. It allows the overvolting and undervolting of the system depending on the circumstances. Overvolting allows a higher operating frequency of the CPU, enabling it to operate in high-performance mode. On the other hand, undevolting reduces the CPU voltage, decreasing its power consumption and saving the system's energy. DVFS feature contributes a lot in systems where energy is a significant constraint (e.g., laptops, smartphones). However, such a technique can lead to reliability degradation of the system. Overvolting increases

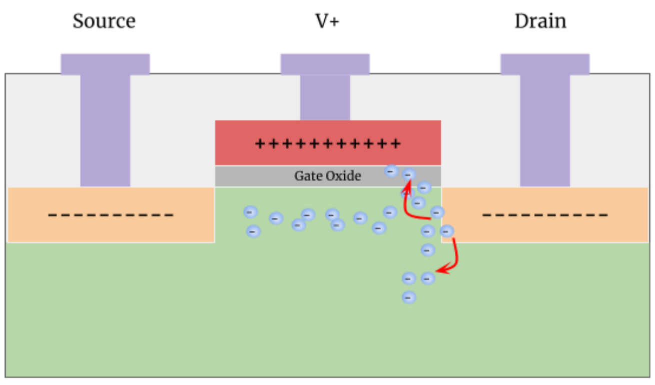


Figure 2.18: Example of Hot Carrier Injection (HCI), when voltage drain-source is exceptionally high. Electrons from the source arrive at the drain at high velocity and with high kinetic energy, striking the atoms in the drain-substrate interface. HCI can cause an increase in substrate current and also degrade the insulation layer.

the amount of heat dissipation, increasing the temperature over the CPU die. Yet, working on operating frequency limits or surpassing them can trigger further reliability issues. Hot-carrier injection and electromigration behavior directly correlates to operating conditions such as temperature and voltage, and intermittent faults might manifest as early advice of the component's possible end-of-life stage (wear-out).

### 2.2.4 Types of Redundancy

Reliability can be achieved in the form of redundancy through several techniques, ranging from low-level (i.e., extra hardware components) to high-level (i.e., redundant multithreading) ones. Designing a resilient system is an exercise in choosing appropriate redundancy strategies constrained by cost and performance requirements. The following sections will describe a few strategies that compose the four base redundancy classes: Hardware, Software, Information, and Time redundancy.

#### Hardware Redundancy

Hardware redundancy is provided by incorporating extra computing units into the design to detect and mitigate the effects of a failed component. For example, having two adders instead of one will allow error detection on add operation scope. Adding a third adder will enable detection plus correction through a voting system. The adder example can be seen as a form of static redundancy, where the goal is immediate detection or correction. Suppose a processor can perform two simultaneous add operations. Instead of providing extra resources for each operation, they may also be shared and activated on-demand. For example, with just four adders, where a pair of adders perform identical add, we can detect some errors. We may then marshal one of the adders from the neighboring pair and re-do the computation for correction. This is a form of dynamic hardware redundancy.

Dual Modular Redundancy and Triple Modular Redundancy are static redundancy strategies where a unit block is duplicated and triplicated to achieve detection and correction (respectively ) of faults that may arise in the unit. However, additional features can be added to DMR or TMR to achieve dynamic redundancy.

A commercial example of dynamic DMR in the scope of microprocessors is IBM's S/390 G5 design [65]. The microprocessor design was composed of several duplicated units (mirrors) that could either be used to achieve a high-performance (e.g., parallel execution) or an ultra-reliable execution. When deployed in mission-critical applications, the system could be dynamically set to provide redundant execution and ensure data integrity and continuous availability as additional features.

Recently, Tesla [66] disclosed information regarding their new hardware responsible for the full self-driving capability. The hardware comprises dual redundant SOCs and power supplies as a safety and security feature. Both SOCs receive overlapping data from external sensors, such as camera, radar, GPS, ultrasonic, and wheel ticks. The system gets this external information, compute and compares the results, validates, and finally performs the action. One should note that this example shows several levels of redundancy beyond DMR since the information provided to the application level (running on both SOCs) is also redundant due to the data overlapping.

### Software Redundancy

Software Redundancy focuses on detecting and correcting errors at a higher level by using additional software components. Redundant multithreading (RMT) is an example of software redundancy [67]. Two or more threads of the same program are launched simultaneously with identical input data, and their outputs are compared for error detection and correction. Examples of RMT usage can be seen in the Operating System code, where it often activates RMT over critical sections of code [68].

### Information Redundancy

Information redundancy is introduced by embedding extra information into the data that will be processed, transferred, or stored in the system. The additional information can be added through several means, such as coding. Coding-based techniques transform the original data through an encoding process. Encoding adds extra bits of information to the data, creating a codeword. After the codeword is processed, transferred, or read, the decoding step transforms the codeword back to its original format, without the extra bits. If errors have affected the codeword for some reason, the decoding process will be able to detect and, sometimes, correct them. Codes can be divided into two types: separable and non-separable. Separable codes provide distinct places where the original and redundant data will be stored in the bit space. Therefore, encoding and decoding separable codewords are straightforward processes. Non-separable codes, on the other hand, merge the original and redundant data, demanding more elaborate encoding and decoding processes. Several coding techniques can provide efficient error detection, and correction capabilities to the system (e.g., Parity, Berger, and Cyclic codes) [18]. However, this thesis will limit the scope to the class of arithmetic error codes, more specifically, the AN Code.

Arithmetic Error Codes are a group of codes preserved during the execution of a set of arithmetic operations. These error codes are of particular interest since they can detect errors in arithmetic units, presenting as a low-cost alternative to DMR or TMR techniques. Given $X$ and $Y$ as input operands, $\hat{X}$ and $\hat{Y}$ their respectively encoded version, and $\cdot$ the arithmetic operation. If $f$ (Equation 2.5) is an arithmetic error code with respect to $\cdot$, the statement in Equation 2.6 must hold true.

$$f : x \to \hat{x} \tag{2.5}$$

$$\hat{X} \odot \hat{Y} = (X \cdot Y) \tag{2.6}$$

AN code composes the non-separable group of Arithmetic Error Codes. AN code works by simply encoding the data with the support of a constant $A$. Therefore, the $f$ of AN code can be considered as seen Equation 2.7 and the set of operations as Equation 2.8

$$f\,(X) = A * X \tag{2.7}$$

$$\cdot \in \{+, -, *\} \tag{2.8}$$

Consider the example of AN code usage on verifying the correct execution of an add unit. The operation performed is an addition ($\cdot = +$) and the operands are 4 and 21. By considering $A = 3$, the codewords are 12 and 63, respectively. Adding up the encoded operands, the result is 75. To check if the operation has been performed correctly, the system must check if the result of the encoded operands is a multiple of $A$. This can be done by applying a modular operation to the output $75 \mod 3 = 0$ (75 is a multiple of 3 – correct result).

When applying AN code, the value of $A$ must be carefully picked since all results multiples of $A$ are considered error-free. Hence, if a fault changes one of the operands or the final result to a multiple of $A$, the error will go through the system undetected. Numbers with few multiples are the best choice in this case. Therefore, prime numbers are good candidates for $A$. To further enhance the choice of $A$, one must decide on a subset of prime numbers, the Mersenne. Mersenne prime numbers are prime numbers of the form $2^n - 1$ for some integer $n$. Their impact lies in the fact that this form of number greatly simplifies the remainder calculation in the checking system. The modulo operation can now be performed over binary chunks of the resulting data.

### Time Redundancy

Time redundancy is provided in the meaning of re-computation. Unlike the hardware version, time redundancy does not require duplicated circuitry to perform the system's check. Detection is performed by re-doing the exact computation over the same component but at a different time. This strategy is excellent for dealing with transient faults over the system component but lacks in detecting permanent or intermittent faults due to their periodicity. Recomputing with shifted operands (RESO) [69] is an example of time redundancy in both the arithmetic and logic operations. In RESO, the input operands (3 and 5 in Equation 2.9) are shifted to the left by one bit during the checking step (Equations 2.10 and 2.11). Then, the operation is performed twice, between the shifted operands and non-shifted ones (Equations 2.12 and 2.13). Both results should match and then be shifted to the right by one bit (Equation 2.14). If both shift-free and shifted operation results match (Equations 2.9 and 2.14), the execution is considered fault-free.

$$3 \times 5 = 15 \tag{2.9}$$

$$3_{10} \rightarrow 011_2 \Rightarrow 6_{10} \rightarrow 110_2 \tag{2.10}$$

$$5_{10} \rightarrow 101_2 \Rightarrow 10_{10} \rightarrow 1010_2 \tag{2.11}$$

$$6 \times 5 = 30 \tag{2.12}$$

$$3 \times 10 = 30 \tag{2.13}$$

$$30_{10} \rightarrow 11110_2 \Rightarrow 15_{10} \rightarrow 11111_2 \tag{2.14}$$

Time redundancy can be combined with self-checking design as proposed by Nicolaidis [70]. The *delay-sampling* idea leverages the lightweight implementation of self-checking designs by monitoring its output. Focusing on transient faults, the proposed idea samples the data twice but adds some delay between each sample. Due to the transient nature of the faults, the technique can detect them in one sample and correct the output by considering the next (or previous) faulty-free sample.

### 2.2.5 Assessment Methodologies

When it comes to reliability, one must provide ways to simulate and understand the behavior of faults that may come from various sources and evaluate the potential detection and correction techniques. However, over such a complex computing system with so many layers, the assessment task reaches a different level of intricacy. Some assessment methodologies can simulate or even reproduce faults in a realistic matter. Still, each method will provide pros and cons, ranging from close to real but with little information on the propagation pattern to simplistic fault models with a practical and fast-paced deployment. Fault injection simulation and physical neutron beam are two traditional methodologies to evaluate the reliability of computing systems.

#### Fault Injection Simulation

Fault Injection Simulation is a technique to assess the dependability of a system under empirical and parametrized experiments through the support of deliberate fault injections [71]. The system behavior is observed to provide insight into critical sections that are more prone to generate errors and demand higher protection. Moreover, fault injection (FI) also enables the evaluation of potential fault-tolerant techniques in a fast-paced fashion. Fault injection can be achieved with the support of software-based simulation tools. Single or multi-bit flips are performed over the execution of a program or system to simulate faults on a software or hardware component. This *injection* of bit flips can be modeled through the support of parameters such as periodicity, amount, and location of bit flips. Fault injection can also be performed over different system levels, enabling the design assessment to varying stages of the development. Examples of FI levels are architecture, microarchitecture, register transfer, and even the software level. The higher level the FI is applied, the faster the analysis can be performed. On the other hand, the lower the level of FI, the more accurate (i.e., realistic) the result is [72]. FI can be performed at the software level through compilation-based or runtime-based simulators [73–77]. Compilation-based approaches have the advantage of knowing the program's structure beforehand but have no data regarding the dynamic state of the program. On the other hand, Runtime-based simulators benefit from accessing the dynamic state of the program.

### Physical Neutron Beam

Physical Neutron Beam [78] is an established assessment methodology that aims to reproduce the behavior of radiation effects on electronic devices. Neutron beans applied directly to the hardware component can mimic the impact of neutron fluxes originating from cosmic rays' strikes at the atmosphere. Although this methodology is the closest to what would be seen in a real scenario, there is no information about the spatial and temporal location of the fault. Faults manifestation will only be observed in the output of the functions or when the system crashes [78]. Additionally, neutron beams are limited to transient faults only, needing extra techniques to evaluate the component under intermittent and permanent faults. Finally, this technique cannot be applied during the development phase, requiring physical access to the device (e.g., demands tape out of the chip).

# Chapter 3

# TorchFI: DNN Fault Injection Framework

As mentioned previously in Chapter 2.2.5, fault injection in a system to assess its reliability can be performed in various ways. One is physical exposure of the device to neutron beans, mimicking the natural radiation-induced type of fault. This technique provides a close-to-real kind of analysis, injecting faults through all physical hardware resources of the device with realistic and proportional probabilities. However, this technique incurs some limitations, such as injection of transient-only types of faults, since the effect will last for a short period and restricted cause-effect information [18, 78]. This impacts the experiment's ability to identify which part of the system is most vulnerable since the effect of a fault will only be observed when system functionality is compromised. Moreover, the physical injection technique demands access to the target hardware device, limiting the assessment of silicon prototypes during the design chain.

On the opposite side, software-based fault injection models provide a flexible, fast, and low-cost way to evaluate the reliability of the target system. It can be performed through simulators that work over different abstraction layers, such as Register Transfer Level (RTL), Microarchitecture, Architecture, and Software. The highest the abstraction layer, the lower the accuracy (and levels of detail) of the results. However, the execution time is reduced by up to 2 orders of magnitude when moving from a lower to an upper layer [78].

Since Deep Neural Networks are compute-intensive workloads, it is impractical to perform fault injections over the low-level abstraction layers, such as RTL, Microarchitecture, and Architecture, to gather statistically significant data. For example, the 16 layers deep VGG model [9] has 138.4 Mega parameters and requires 15.5 Giga MAC operations to run a single inference (single image classification). Consequently, to run a complete evaluation of the model under faulty conditions, the simulator would have to cover the whole ImageNet [11] validation dataset, which comprises 50,000 images in total. Therefore, a custom DNN software fault injection simulator is the most reasonable choice for this task since it can run at native silicon speed.

## 3.1 Back-end

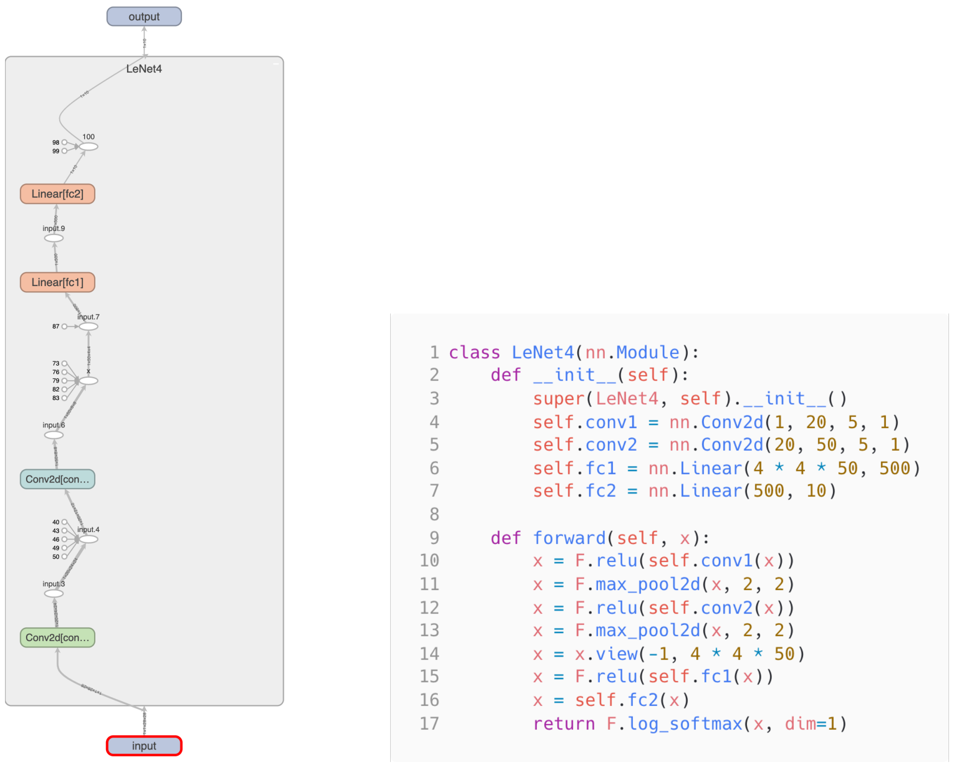
As a design choice, the popular deep learning platform PyTorch [79] was selected to compose the fault injection framework stack base. PyTorch enables the DNN models to run over GPUs and CPUs through its optimized tensor library, allowing fast prototyping and execution. Moreover, the PyTorch environment comprises TorchVision, a library package containing datasets, popular model architectures, and, most importantly, pre-trained models for computer vision tasks (e.g., image classification).

## 3.2 Computational Graph

The PyTorch main component is the computational graph. This graph is dynamically created during the execution of the program. Every node of the graph consists of a function to be executed over an input data/tensor (i.e., NN layers). These functions can range from naive math operations, such as addition, multiplication, and subtraction, to complex ones over n-dimensional tensors, such as convolution, linear transformation, and cross-entropy loss. The graph's edges represent the dependency between nodes and the data flow. To create, train, and test a neural network in PyTorch, the user makes use of the extensive Python API that abstracts the "heavy-work", been only required from the user the DNN architecture specification (i.e., layers and its inputs), the dataset images, handle by custom data loaders classes and information of which backend will be used (e.g., GPU). Figure 3.1 shows an example of PyTorch computational graph (left panel) and the required code to implement it (right panel).

## 3.3 Front-end

TorchFI [77] composes the upper part of the fault injection framework stack. It leverages the whole infrastructure provided by PyTorch to enable high-level fault injections in the computational graph. Users can orchestrate the injection through predefined flags, for example, selecting which nodes (i.e., layers) from the graph will be affected by the fault. The fault location also can be refined, allowing the user to choose lower layer structures, such as channel, filter, and weight, in the case of a



```python
class LeNet4(nn.Module):
    def __init__(self):
        super(LeNet4, self).__init__()
        self.conv1 = nn.Conv2d(1, 20, 5, 1)
        self.conv2 = nn.Conv2d(20, 50, 5, 1)
        self.fc1 = nn.Linear(4 * 4 * 50, 500)
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4 * 4 * 50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

Figure 3.1: LeNet-4 example of PyTorch computational graph (left panel) and the required code to implement it (right panel).

convolution layer. Additionally, the injections can simulate faults in arithmetic units and buffers. In the former case, faults are applied inside the call for convolution and linear operations. The latter refers to fault injections in the layer tensors, such as activations and parameters (i.e., weights). This flexibility provides users an easy way to analyze different types of fault models and error propagation with a minimal overhead of coding and execution time.

## 3.4 Dynamic Runtime Patching

Figure 3.2 shows the code necessary to prepare the current model for fault injections. The user must instantiate the FI class presented in TorchFI's package passing as arguments the model that will be equipped with fault injection capability, the layer id where the fault will appear, the position where the bit-flip will occur, and where it will be placed, buffers (e.g., activations and weights) or arithmetic unit. The only mandatory parameters are the model and the location where the fault will be placed. If not set, the remaining ones will be randomly defined at run-time.

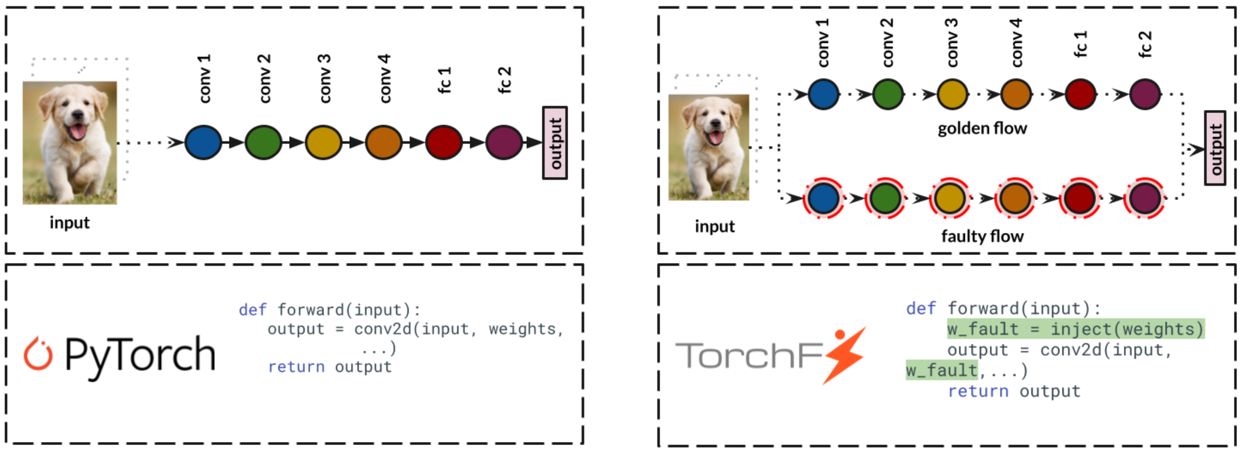
After instantiating the FI object, the user must invoke the traverseModel method. This method will execute a dynamic runtime patching (aka, monkey patch) that will traverse the whole computation graph of the model, replacing the layer's object with custom FI-enabled ones. These customs FI-enabled layers will act as

```python
fi = FI(fi_model, fiMode=args.injection, fiLayer=args.layer, fiBit=args.bit, fiepoch=args.fiEpoch,
        fiFeatures=args.fiFeats, fiWeights=args.fiWeights)

fi.traverseModel(fi_model)
```

Figure 3.2: FI object instantiation. FI prepare the original model for fault injections by traversing the model and replacing the original layer's objects with TorchFI's ones. FI can be fully parameterized to achieve the expected fault injection campaign.

wrappers to the old objects, adding the fault injection capability through new methods and additional fields to track and analyze the fault propagation (Figure 3.3). Whenever the forward method is invoked (Figure 3.3), the custom fault injection methods will be called to check if a fault should or not be injected and its specifications (i.e., structure location, bit position, etc.).



```python
def forward(input):
    output = conv2d(input, weights,
        ...)
    return output
```

```python
def forward(input):
    w_fault = inject(weights)
    output = conv2d(input,
w_fault,...)
    return output
```

Figure 3.3: PyTorch vs TorchFI code. TorchFI specialized classes wraps PyTorch's conv2d and linear (i.e., fully connected) classes, adding fault injection capability over the forward passes.

TorchFI logs several pieces of information regarding each fault injection round, such as golden and faulty values (i.e., faulty-free and disturbed data), layer and tensor location (channel and filter ids), as well as its impact on the final scores on the classifier and the model's accuracy (top1 and top5 predicted classes). This can be further extended to track and store additional information regarding the fault injection campaign.

## 3.5 Fault Models

TorchFI covers both transient and intermittent fault models. For transient faults, single bit-flip are injected during every faulty round. Due to the transient and randomness characteristics of soft errors, each execution of the faulty model may differ from the other. However, TorchFI's flexible parametrization can enforce a deterministic fault injection campaign over the target model, allowing reproducibility of the experiments.

Unlike transient faults, errors produced by intermittent tend to occur in bursts and repeatedly at the same location. Hence, TorchFI was designed to perform multiple bit-flips either. Often over the same structure, reproducing the same behavior of multi-clock-cycle faults. Moreover, due to its non-deterministic activation nature, intermittent faults are injected at a random time and location.

## 3.6 Current Limitations

Since almost 99% of operations of a CNN arise from Convolutional and Fully Connected layers [80], TorchFI's fault injection model is restricted to these two types of layers. This can further be extended to support an additional set of layers and networks, such as Recurrent layers (e.g., RNN, LSTM, and GRU), Transformer layers (e.g., Encoder and Decoder), and Sparse layers (e.g., Embedding and Embedding Bags).

TorchFI was in constant improvement during this thesis development. In its first version, TorchFI leveraged part of Intel's Nervana Distiller [81] package for post-training quantization since PyTorch did not have native quantization support at that time. The latest version of TorchFI leverages the recently implemented post-training quantization of PyTorch. The variety of quantization techniques is limited compared to the ones offered by Distiller, but it provides better code stability and maintainability.

TorchFI is limited to inference phase-only. Therefore, fault injection campaigns are only possible in the forward pass, not in the backward pass. This design decision arises because the training of a DNN model is typically done once while the inference is re-executed several times. Thus, the training phase is out of the scope of this work.

Finally, TorchFI is a high-level fault injection framework that operates at the application level. It provides a fast way to analyze the impact of faults over DNNs and understand its implications of error propagation in the application result. However, this type of fault modeling might not capture the exact behavior of a low-level injection (e.g., register-level injections). Yet, recent studies have shown that high-level fault injections results are prominent and can be used to study the implications of errors at a system-level [75, 82] at orders of magnitude faster.

## 3.7 Relate Work

Reliability analysis of DNN accelerators has received extensive attention over the past years. Most of the prior works are related to the reliability of DNN models concerning faults occurring on buffers, where weights and activation values are stored during computation. The fault injections simulations are mainly performed through software-based frameworks built on top of popular open-source DNN platforms.

PyTorchFI [75] provides a runtime perturbation of DNNs through fault injection campaigns. Like TorchFI, it also leverages the PyTorch stack as the back-end to deliver an easy-to-use API. PyTorchFI differs because it does not apply a dynamic runtime patching to the computational graph, reducing the final code's interference and execution time overhead. Instead, it takes advantage of PyTorch's hook function to inject the perturbations during the execution of the forward method, removing the necessity of a graph instrumentation phase.

TensorFI [76] was designed to operate on top of TensorFlow [83]. Since TensorFlow uses a compilation-based approach to build its computational graph, in-place fault injection is impossible. Instead, TensorFI works in a "two-phase injector" by first duplicating TensorFlow's generated dataflow graph (i.e., computational graph) fully. Then, it replaces the original operators with new ones and places them in the replica. These new operators are fault-injection enabled and controllable by external configuration files. During the execution of the original dataflow graph operators, the copies emulate their behavior but are capable of injecting faults.

TorchFI provides a lightweight and easy-to-configure way to perform fault injection campaigns. Unlike PyTorchFI, which has a more mature and plug-and-play design, TorchFI requires an instrumentation phase over the layers where faults will be injected. Yet, it does not duplicate any instance of the computational graph as seen on TensorFI.

Although some reliability works have been published during the development of this thesis, none of those open-source frameworks mentioned above were available at that time (first commit on GitHub in December of 2018). Therefore, the demand for a fast and flexible tool such as TorchFI was latent.

# Chapter 4

# Reliability Evaluation of Compressed Deep Learning Models

Deep neural networks (DNNs) have been established as state-of-the-art models for several tasks, such as object detection, image classification, machine translation, and fraud detection. Some of these tasks are currently being performed by DNNs in safety-critical autonomous systems (e.g., self-driving cars), increasing the demand for reliability.

Understanding the behavior of such applications under threatening conditions is of paramount importance. Faults 2.2.1 can cause perturbations over the model's behavior, significantly affecting the final prediction. Its propagation can result in classification error, putting at safety at risk. Therefore, an extensive analysis considering several different variable conditions can significantly contribute to a better understanding of DNN's reliability.

Silent data corruption (SDC) composes the list of possible threats to any system that runs over a semiconductor device. SDCs are faults that manifest as errors, corrupting the output data of an arithmetic unit or storage system. It is named "silent" due to its undetectable characteristic. Transient faults (Section 2.2.2), such as from alpha particles or neutron strikes, are the most typical source of SDCs. The soft error caused by these radiation-induced events can go undetected since they occur at a random location and time. Therefore, the assessment of DNNs facing SDC is crucial to understand how the application will behave and adequately estimate the reliability requirements of DNN-oriented microarchitectures.

DNNs have achieved unprecedented accuracy results over several types of tasks. However, the model size, memory footprint, and computational complexity required by these networks make them hard to be deployed on limited resource devices (e.g., smartphones, small single-board computers, etc.). This has led to a surge of research focusing on model compression methods.

The main goal of model compression techniques is to reduce the amount of

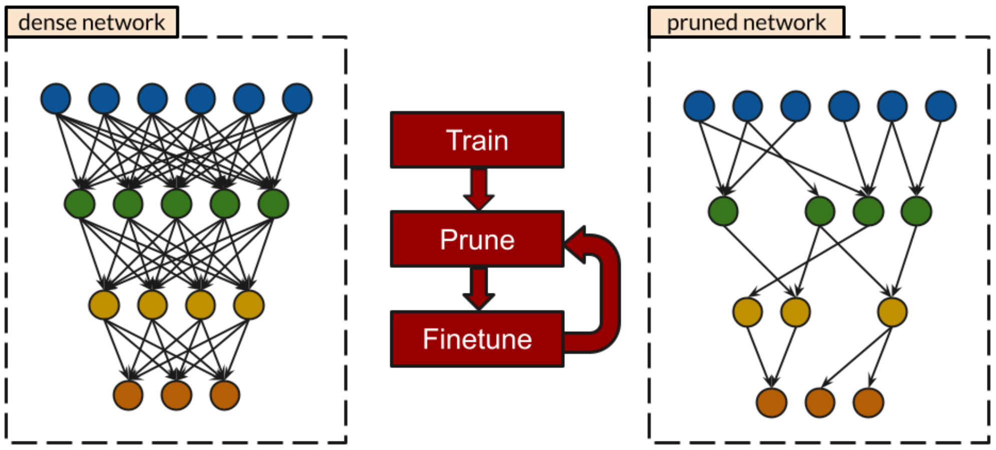


Figure 4.1: Iterative pruning process proposed by [3]. Weights and connections are removed based on their relevance on the final classification accuracy.

space required to store the model and the amount of computation and bandwidth demanded to process and transfer from the main memory to the processing element. The space reduction can be achieved by removing unnecessary (i.e., redundant) nodes from the neural network. The pruning technique (Section 2.1.5) can remove up to 90% of the weights of a network by applying an iterative process that measures the importance of each weight to the output. Weights that are not capable of achieving the minimum threshold are removed from the networks (Figure 4.1). Data quantization (Section 2.1.5) is a compression technique that aims to reduce the computation cost and the bandwidth requirements of a neural network. By reducing the number of bits necessary to represent the numbers, consequently decreasing the available range and precision, the area and space requirements in a chip design are dramatically lowered (Figure 4.2). DNNs were found to be remarkably resilient to data quantization, enabling lower-precision configurations (e.g., INT 8-bits) with marginal or no accuracy loss.

Although some research has been performed on the reliability assessment of DNNs, especially over fixed-point quantized ones, there are a few questions to be answered. Since pruning removes unnecessary nodes and reduces the amount of weight redundancy of the network, how does a pruned model behave under a faulty condition? Do fewer nodes relate directly to a higher exposure of the relevant weights? Does the model get more susceptible to miss classification with the most relevant nodes exposed? Since the gap between quantization levels is lower (e.g., INT 16-bits to INT 8-bits), is the impact of a bit-flip more significant. Does quantization act as a contention mechanism, avoiding data overflow due to error propagation (e.g., no bit-flip over exponent part)?

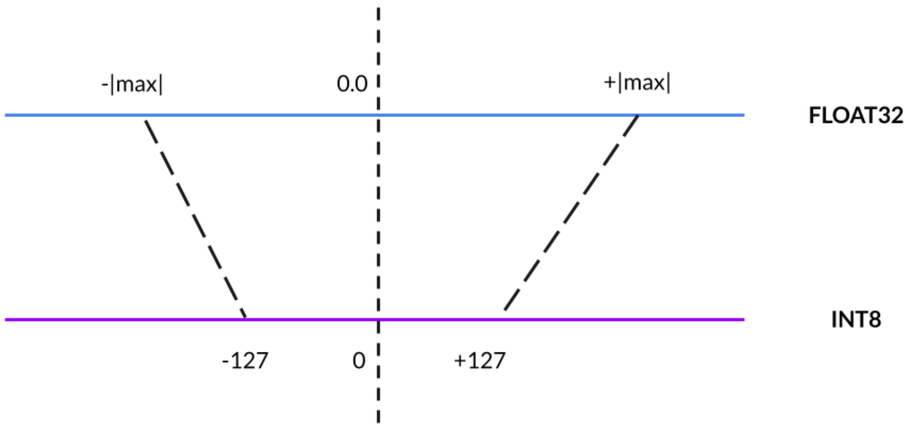


Figure 4.2: Signed 8-bits symetric quantization (i.e., zero-centered) of 32-bit floating-point data.

The following sections aim to answer these questions by providing sufficient empirical analysis to support them.

## 4.1 Evaluation Methodology

The fault model and metrics used to perform the reliability evaluation of DNNs are described in the following sections.

### 4.1.1 Fault Model

The reliability evaluation methodology applied in this section simulates transient faults over the datapath components of deep learning-oriented hardware devices. The simulation is achieved by injecting single bit flips on random live values stored in latches or registers. The units above are mapped in a high-level DNN framework, and faults are injected with the support of TorchFI (Chapter 3). The code snippet in Figure 4.3 describes at a high level how the injection is performed over the inputs of a two-dimensional convolutional layer. Faults can either affect input features or the weights of a neural network. However, due to the size of random space, the fault model is restricted to weight buffers only, making the analysis more meaningful. Moreover, unlike the input features, weights are held over the buffers for a long time and reused in other inference iterations over several different input images, being more likely to experience transient faults.

```python
class FIConv2d(nn.Conv2d):

    def __init__(self, ...):
        self.fi = fi
        self.name = name
        self.weightFI= weight
        self.id = fi.addNewLayer(name, FIConv2d)

        super(FIConv2d, self).__init__(...)

    def forward(self, input):
        # injecting fault over the weights
        indices, faulty_val = self.fi.inject(self.weightFI.data)
        self.weightFI.data[tuple(indices)] = faulty_val
        # call forward pass with input and faulty weights
        super(FIConv2d, self).conv2d_forward(input, self.weightFI)
```

Figure 4.3: High level code of how injections are performed over the inputs of a two-dimensional convolutional layer. FIConv2d wraps the original object from PyTorch's original Conv2d class and adds fault injection capability on the top of it (in the forward method).

### 4.1.2 Metrics

As the faults are injected, they will eventually propagate through the layers, generating SDCs that can or cannot lead to misclassification. Therefore, a metric to estimate the probability of an SDC occurring should be accounted for. Unlike general programs, where a bit-by-bit checking can be performed over the result to evaluate the occurrence of an SDC, neural networks output a set of scores associated with each class. The scores are further transformed in a probability distribution where the prediction accuracy consists of the ground truth class probability. The network output is correct if the ground truth class has the highest probability. This metric is named top-1 accuracy. The top-5 accuracy is calculated by considering the highest probability and the remaining top four ones. If the ground truth class lies in the top five probabilities, the prediction will be counted as correct; otherwise, misclassification. If an SDC occurs, it might affect both the top-1 and top-5 accuracies, the former being a much more harmful form of SDC. Li *et al.*[84] introduced new metrics to measure the impact of SDCs over DNNs that we partially follow in this work as:

- SDC@1: The highest probability class differs from the top one of a fault-free execution.
- SDC@5: The highest probability class is not in the top five probabilities of a faulty-free.

Finally, to evaluate the overall resilience of the DNNs, an SDC probability is calculated. Through the execution of multiple faulty runs and a single faulty-free (golden) run, the SDC@1 and SDC@5 probabilities ($P$) are calculated by Equation 4.1, where $golden_i$ and $faulty_i$ are the output of fault-free and faulty runs, respectively, and $N$ is the number of input samples evaluated over each dataset.

$$P = \frac{\sum_{i=1}^{N}(golden_i \neq faulty_i)}{N} \tag{4.1}$$

## 4.2 Experimental Analysis and Results

A set of empirical experiments were performed to understand and evaluate the impact of transient faults over compressed DNNs. The experiments were divided into three categories, one to assess the effect of low-precision quantization, such as integers with 16 and 8 bits, one to determine the impact of pruning over a diverse group of rates, and the last one to assess the reliability of both, quantization and pruning, together. Each experiment round consists of 50,000 inferences over the ImageNet validation dataset [11]. Since the faults were randomly injected, the SDC's probabilities are presented as averages of five rounds (50,000 inference runs each) . For each group of experiments, there is a faulty-free golden run used as the baseline.

The analysis was performed on a layer-wise fashion and overall fashion. The layer-wise manner outputs a fine-grained impact of the transient faults since the location is restricted to the weights of a single layer. The overall approach considers the whole network as a single unit so that faults can affect any network layer.

The symmetric post-training quantization technique was performed for the quantization analysis of the weights and activations. Since, during the development time, the PyTorch framework did not support native quantization, the Intel's Nervana *Distiller* package [81] was incorporated into TorchFI. Weights and activations were either quantized from single-precision floating-point 32-bit numbers to 8-bit or 16-bit integers (both in signed formats). Therefore, no retraining was required, and marginal to no drop in the model's accuracy ($< 0.5\%$ drop).

The pruned models [1] were obtained by applying the *Automated Gradual Pruning* (AGP) method introduced by Zhu and Gupta [53]. The AGP follows the idea proposed by Han *et al.*[52] but enhances it by gradually decreasing the pruning rate over iterations. It relies on the assumption that initially, more weights are susceptible to be pruned, and the rate decreases as you keep iterating (i.e., pruning).

The following sections will detail the models, datasets, the experimental setup, and the results of each experimental category.

[1]Pre-trained pruned models were obtained with the support of *Distiller* [81] and provided by *DeGirum* (`https://github.com/DeGirum/pruned-models`).

### 4.2.1 Models and Datasets

The DNN models used during the reliability analysis is listed in Table 4.1. AlexNet, ResNet18, and ResNet50 are considered mid-size and large-size models with 8, 18, and 50 convolutional and fully connected layers.

AlexNet achieves a top-1 accuracy of 56.52% without any compression technique (FP-32 data type). By pruning 88.31% of its weights, the model increases its original accuracy by 0.09%. Note that the increase in the model's accuracy is expected since pruning involves retraining. Quantization into a 16-bit integer also incurs an accuracy increase, but the model presents a marginal degradation when going for an 8-bit integer (0.53%). On the other hand, by applying both pruning and quantization, AlexNet can sustain the same accuracy with 16-bits and a reduced drop in accuracy with 8-bits (0.17%).

ResNet18 achieves a top-1 accuracy of 69.76% with no pruning and FP32b data type. With 59.92% of sparsity, the pruned version achieves even higher accuracy and only a tiny degradation when INT8b quantization is applied.

In ResNet50, three pruned versions are provided. The original model with no compression achieves a top-1 accuracy of 76.13%. The pruned models present some degradation, but no more than 0.51%, with 70.66%, 83.37%, and 84.57% of pruning rates. Quantization incurs a marginal degradation of less than 1%.

The 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) validation set was employed as the dataset. It consists of 50,000 images separated into 1,000 categories (i.e., classes). The images are cropped to 224x224x3 size and input into the network. Each model was pre-trained over $\tilde{1}.2$ million images (training set).

### 4.2.2 Experimental Setup

The experiments were performed over thousands of machines of the Intel Vlab Cluster. Each machine was equipped with an Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz with 112-cores each. The fault injection tool TorchFI v0.1 (Chapter 3) and the DNN framework PyTorch v1.3.1 [79] were employed to run the simulations.

### 4.2.3 Quantization Impact

Faults injected at different locations over different data types can generate distinct propagations. Therefore, the injection campaigns are restricted to specific areas of the network in the first round of experiments. Over a layer-by-layer fashion, depicted in Figure 4.4, faults are injected, exposing which layers are more prone to deliver misprediction caused by SDC. Without any contributions of quantization (fp32 lines), the network shows a higher probability of SDCs in the first set of layers.

Table 4.1: Models top-1 and top-5 accuracy under compression. Pruning (P) drops the top-1 accuracy by less than 1%. Quantization (Q) only incurs a 0.5% drop on accuracy.

| | **Model Optimizations** | **Data Type** | **Pruning Rate** | **Top-1 acc.** | **Top-5 acc.** |
|---|---|---|---|---|---|
| AlexNet | Original (O) | fp32 | - | 56.52% | 79.07% |
| | Pruned (P) | fp32 | 88.31% | 56.61% | 79.36% |
| | O + Q | int16 | - | 56.61% | 79.36% |
| | P + Q | int16 | 88.31% | 56.52% | 79.08% |
| | O + Q | int8 | - | 55.99% | 78.87% |
| | P + Q | int8 | 88.31% | 56.35% | 79.04% |
| ResNet18 | Original | fp32 | - | 69.76% | 89.08% |
| | Pruned | fp32 | 59.92% | 69.87% | 89.16% |
| | O + Q | int16 | - | 69.76% | 89.08% |
| | P + Q | int16 | 59.92% | 69.88% | 89.16% |
| | O + Q | int8 | - | 69.33% | 88.91% |
| | P + Q | int8 | 59.92% | 69.54% | 89.06% |
| ResNet50 | Original | fp32 | - | 76.13% | 92.86% |
| | Pruned | fp32 | 70.66% | 75.94% | 92.96% |
| | Pruned | fp32 | 83.37% | 75.76% | 92.92% |
| | Pruned | fp32 | 84.57% | 75.52% | 92.72% |
| | O + Q | int16 | - | 76.13% | 92.86% |
| | P + Q | int16 | 70.66% | 75.93% | 92.96% |
| | P + Q | int16 | 83.37% | 75.76% | 92.92% |
| | P + Q | int16 | 84.57% | 75.52% | 92.72% |
| | O + Q | int8 | - | 75.75% | 92.73% |
| | P + Q | int8 | 70.66% | 75.63% | 92.80% |
| | P + Q | int8 | 83.37% | 75.53% | 92.69% |
| | P + Q | int8 | 84.57% | 75.11% | 92.52% |

These results are aligned with the pruning behavior detailed by Han *et al.*[3], where the first set of layers are more sensitive to higher pruning rates compared to the remaining ones. This issue arises because the first layers are closer to the raw input data, and any disturbance during the feature extraction will significantly impact the remaining ones.

The model benefits greatly by considering quantization. Integer data type with 16 bits can reduce the SDC probability by 27x, keeping it below 1%, even for the first set of layers. On the other hand, integer data type with 8-bits incurs a higher SDC rate for the first set of layers, and the rate reduces as it goes deeper into the network. This behavior can be better understood by looking at the second experiment scenario, which targets the bit position where the state is changed (Figure 4.5). Faults now flip the same bit position but at a random place in the network. Figure 4.5 shows that the majority of misclassification for the FP 32bits occurs when the bit-flip hits the most significant bit, overshooting the weight value, causing a

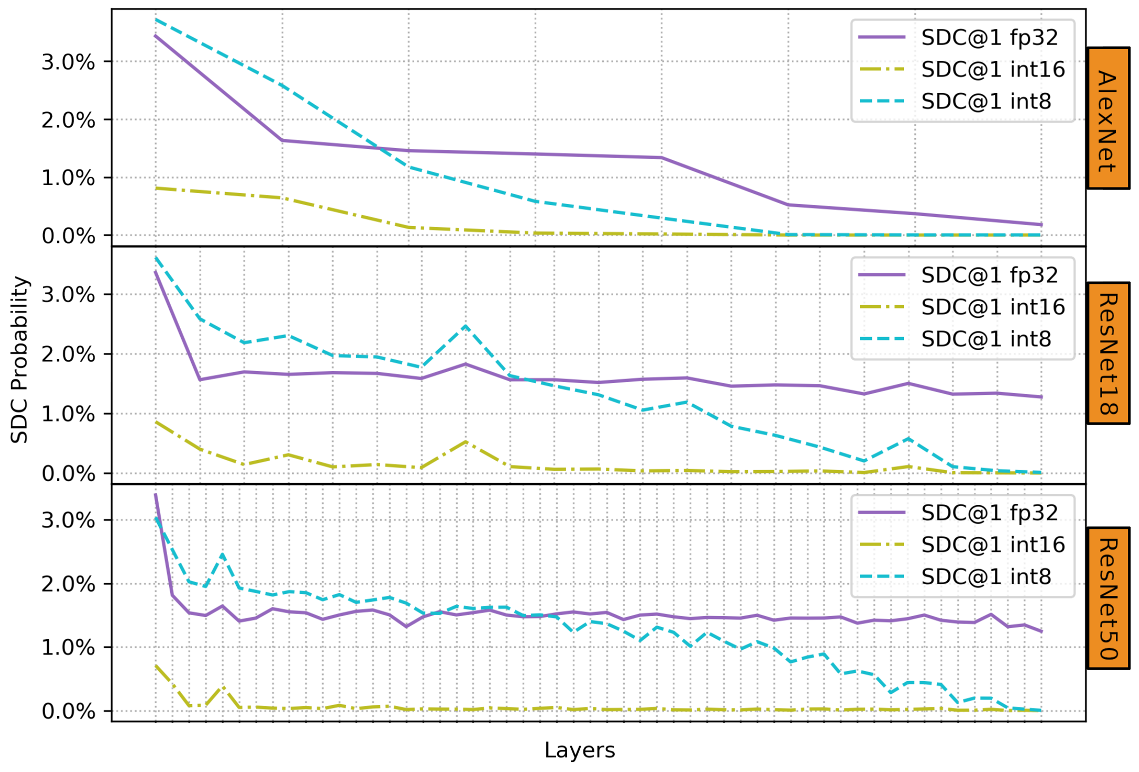


Figure 4.4: Impact analysis of transient fault over the layers with different levels of quantization.

cascade of close-to-maximum range numbers flowing in the network. When moving to integer format, this overshooting behavior is contained. INT 16-bits suffer more changes over the sign bit, followed by the higher-order ones. However, the ranges observed in the network are controlled by the quantization range, and overshoot is mitigated. Moving to INT 8-bits, the number of levels is reduced by half, and the bit-flips over any position can almost provide equally disturbance.

Overall, integer-only quantization can act as a mitigation technique by reducing the maximum and minimum ranges that fault can affect the data (up to 27x SDC reduction). However, due to the low number of quantization levels, INT 8-bits provides instability and has a higher probability of delivering misclassification when hit in the first set of layers.

### 4.2.4 Pruning Impact

Pruning compacts the model by removing non-relevant weights while keeping the baseline accuracy. In a real-world scenario, the sparsity created by pruning would save memory space, bandwidth, and compute cycles. However, pruning can act as a double-edged sword from a reliability point of view. Since only relevant weights would be stored and computed, the most critical ones would be more exposed to

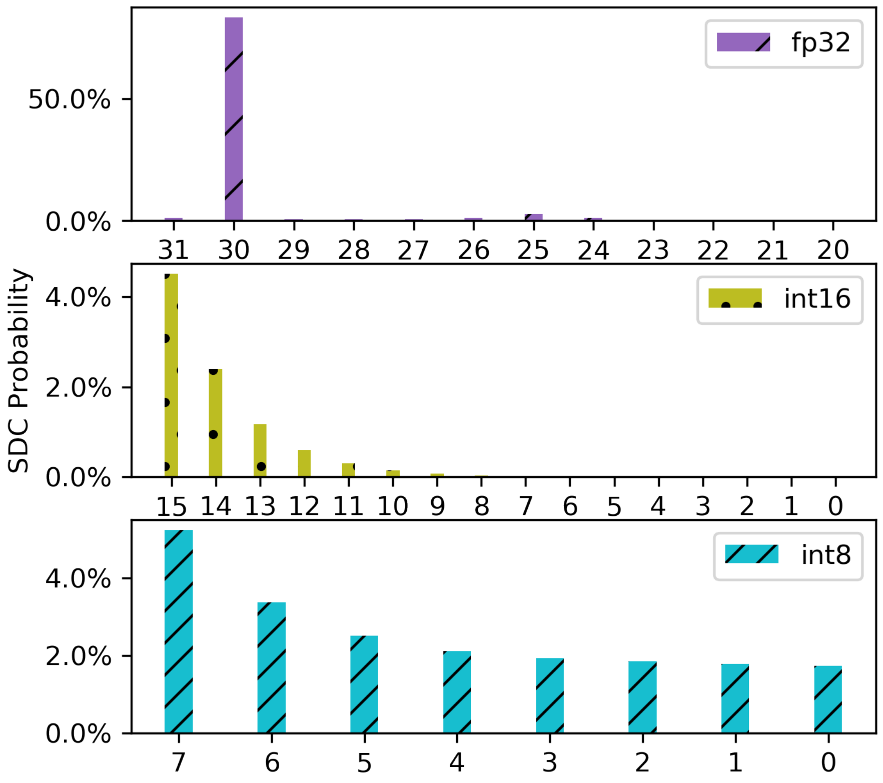


Figure 4.5: Error rate of each bit position when hit by a fault. Floating-point 32-bit plot is truncated for a better visualization.

faults, but at the same time, the space exposed (in memory) and the time required to run the network would decrease substantially. Therefore, these facts should be considered when performing the reliability assessment of pruned models. Changes over the fault rate are needed, and some upper and lower bounds estimations should be provided.

First, the experiments measure the SDC probability of a pruned model under the same amount of faults applied in a non-pruned one (100% faults). This explains how the network behaves with relevant-only nodes exposed to faults by not considering the computational benefits of pruning. Second, the number of faults injected is reduced based on the model's sparsity (70% pruned weights = 30% faults applied). This roughly approximates the pruned results to a real-world scenario, where the model would run faster (fewer clock cycles required) and consume less memory (save area) when compared to a non-pruned model. Finally, an upper and lower bound estimation of the gains in reliability considering the memory and computation reduction (SDC Est. Reduction) is provided by Equations 4.2 and 4.3, respectively.

$$upper\ bound = \frac{sdc\_rate}{param\_reduc} \tag{4.2}$$

$$lower\ bound = \frac{sdc\_rate}{param\_reduc * comp\_reduc} \tag{4.3}$$

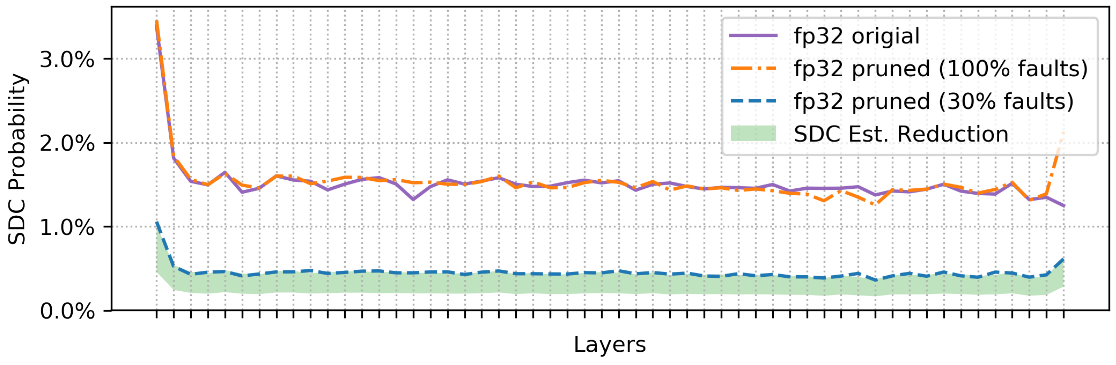


Figure 4.6: Impact analysis of transient fault over the layers in a dense and 70% sparse model with floating-point 32-bit data type. Fault compensation (30% faults) fits in the lower and upper bounds of estimated reduction calculated using the performance gains of pruning.

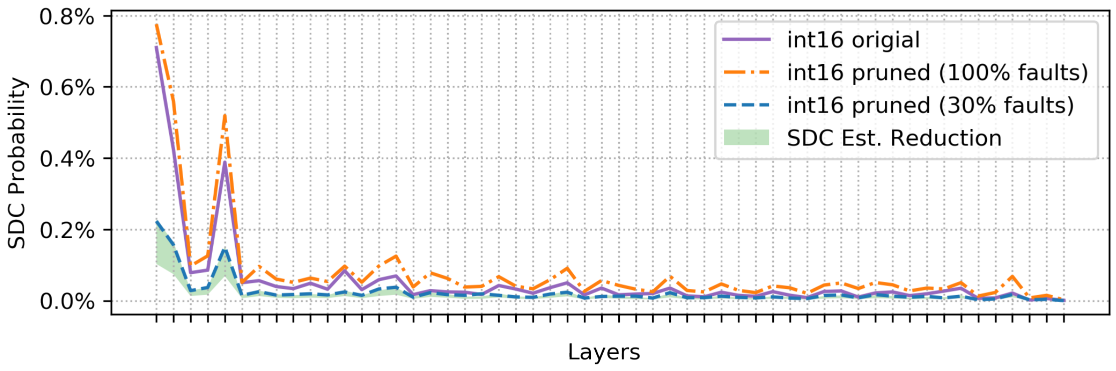


Figure 4.7: Impact analysis of transient fault over the layers in a dense and 70% sparse model quantized to integer with 16 bits. Fault compensation (30% faults) fits in the lower and upper bounds of estimated reduction calculated using the performance gains of pruning.

Figures 4.6, 4.7, and 4.8 show both scenarios in a layer-wise fashion, where the number of faults injected does not consider the benefits of pruning and where it is considered (fewer faults are applied). By applying the same rate of faults as for non-pruned models (100% faults), the compressed version behaves poorly, especially over the quantized versions. This observed trend is mainly due to the higher exposure of critical nodes and less redundancy provided by pruning. When faults hit the nodes of a pruned model, there is a higher probability that an SDC will occur, generating a misclassification.

Figures 4.6, 4.7, and 4.8 also provide the estimate SDC reduction, with lower and upper bound defined by Equations 4.2 and 4.3, respectively. This rough estimation

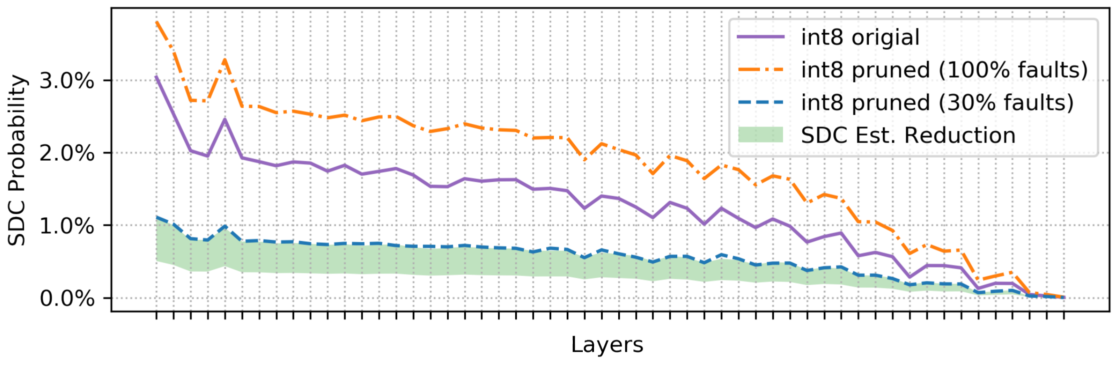


Figure 4.8: Impact analysis of transient fault over the layers in a dense and 70% sparse model quantized to integer with 8 bits. Fault compensation (30% faults) fits in the lower and upper bounds of estimated reduction calculated using the performance gains of pruning.

Table 4.2: Parameter and computation reduction rates for the ResNet50 model over different sparsity levels. GMACs stands for Giga Multiply and Accumulate operations. Source: `https://github.com/DeGirum/pruned-models`.

| **ResNet50 Model** | **Number of GMACS (Compute Reduction)** | **Number of Parameters (Parameter Reduction)** |
|---|---|---|
| Dense | 4.089 (1.00x) | 25.5M (1.00x) |
| Pruned 70.66% | 1.846 (2.21x) | 7.48M (3.41x) |
| Pruned 83.37% | 1.143 (3.58x) | 4.24M (6.01x) |
| Pruned 84.57% | 0.714 (5.73x) | 3.93M (6.48x) |

considers the parameter reduction rate and the computation reduction rate (Giga MACs) depicted in Table 4.2 for ResNet50 model. By reducing the number of faults injected over a pruned model and compensating it w.r.t. the percentage of parameter reduction, the SDC probabilities decrease and fit into the estimated SDC reduction area provided by the upper and lower bounds. Therefore, over a real-world scenario and hardware capable of performing high-performance computation over sparse DNNs, the SDC rates would fall closer to the lower bounds, providing substantial reliability gains.

### 4.2.5 Overall Impact

By combining both compression techniques, the results provide noticeable gains in the overall reliability of models. Figures 4.9, 4.10, and 4.11 gives an overview of the potential gains on the SDC rate when combining pruning and quantization for AlexNet, ResNet18, and ResNet50, respectively. Note that the results consider a non-layer-wise approach, and the numbers reflect the average SDC with faults injected at a random location.

The reliability improvements of INT16 bits quantization compared to the FP32 data type can reach up to 27.4x. When combined with high pruning rates (85% sparsity), the model compression techniques can deliver up to 108.7x SDC reduction compared to the non-pruned FP32 baseline.

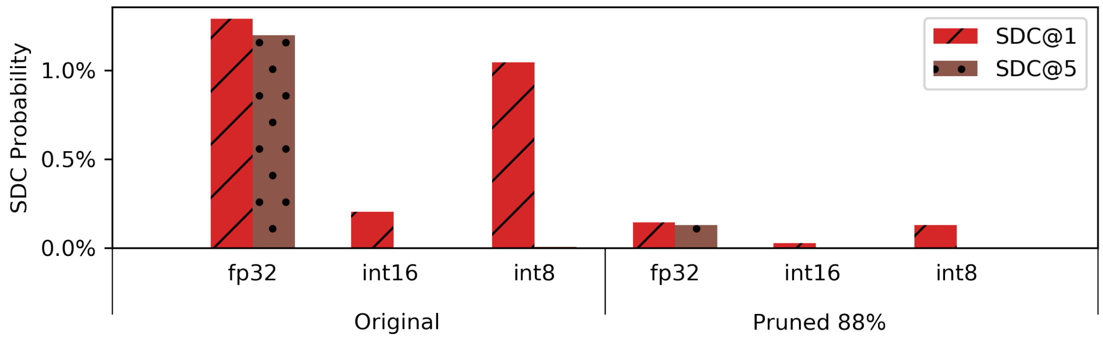


Figure 4.9: Overview of the overall error reduction provided by model compression techniques on AlexNet compared to the dense version (*Original*).

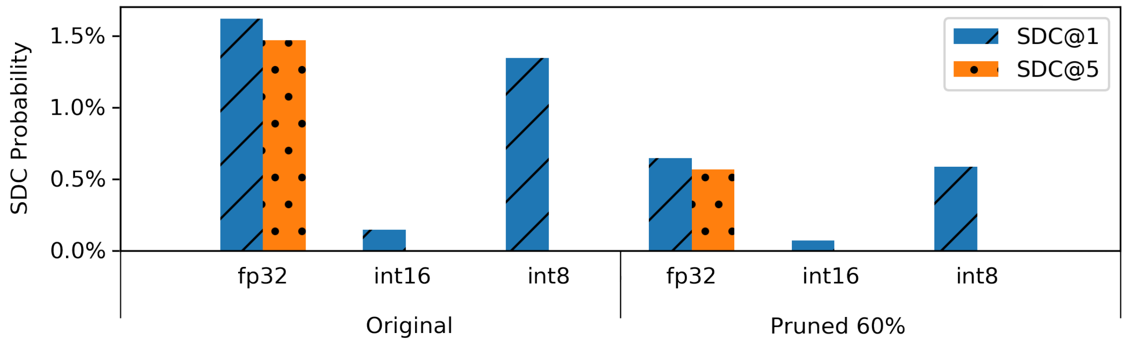


Figure 4.10: Overview of the overall error reduction provided by model compression techniques on ResNet18 compared to the dense version (*Original*).

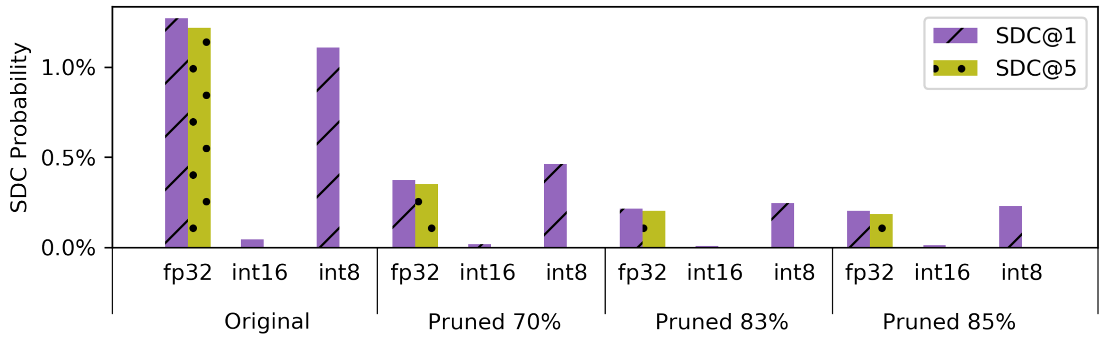


Figure 4.11: Overview of the overall error reduction provided by model compression techniques on ResNet50 compared to the dense version (*Original*).

## 4.3 Concluding Remarks

Model compression techniques can significantly benefit the deployment of deep neural networks over resources constrained devices by reducing the space, bandwidth, and computation requirements. However, the reliability implications of such compression techniques must be evaluated to provide enough guarantees that DNNs applications will keep delivering the expected classification outputs. Quantization improves the overall reliability of the network by containing potential overshoots on parameter values due to the high range offered by the 32-bit floating-point data type. The empirical fault injection experiments show that integer-only quantization with 16 or 8 bits is beneficial to the application resilience by adding a 27.4x SDC reduction, avoiding a higher misclassification rate. However, aggressive quantization with fewer levels, such as provided by 8-bit integers, should be prudently applied since it can increase the SDC rate (22.5x) compared to 16-bits.

Pruning solely can decrease the system's reliability when not accompanied by performance improvements. The hardware must provide ways to deal with sparsity over DNNs, by avoiding storing and fetching zero values and computing them. Overall, the performance gains offered by pruning can improve the system's reliability by up to 9x. Both techniques, pruning, and quantization, combined, can deliver up to 108.7x more reliability.

## 4.4 Related Work

Reliability analysis of deep neural networks applications and hardware is an active area of research. Most of the works focus on transient types of faults, but the absolute majority rely on small and non-pruned (dense) models using single or half floating-point precision, or even some fixed-point variations [84–88]. Reagen *et al.*[86, 89] proposed the analysis of compressed models but considered a different fault model. The assessment of DNN models was performed through the injection of multiple bit-flips to understand at which rate a model begins to degrade its accuracy. The first work [89] considers pruned but tiny models with up to 5 layers. The second work [86] performs the assessment of more prominent models (e.g., ResNet50) but does not consider pruned models and only evaluates fixed-point quantization. Li *et al.*[84] performs an extensive investigation on the impact of transient faults over deep learning neural network (DNN) accelerators. However, the quantization considered in their work is fixed-point, and no pruning is taken into account.

# Chapter 5

# Lightweight Error-Resiliency Mechanism for Deep Neural Networks

Deep Neural Networks (DNNs) have been successful in many areas by delivering state-of-the-art accuracy results [90–92]. This success has attracted considerable attention, especially from major tech companies, pushing DNN deployment targets to the limits. Safety-critical systems, such as autonomous cars, are one of the many targets for DNNs. This type of system demands high dependability standards in order to be considered safe for end-users. Therefore, performing a resilient execution of DNNs is of paramount importance.

To achieve the best performance (e.g., execution latency) at the lowest cost (e.g., energy and area), several companies have reported the development of a custom set of hardware and sensors (e.g., cameras, processors, and radars) specialized for DNN applications applied to autonomous car systems [93–95]. However, achieving resilient application execution over such components demands some expenses. In other words, there is no free lunch. Ones must trade area and energy to implement reliability techniques over the HW components. Additionally, such systems must detect 99% of the faults in any component and not exceed 10 Failures-in-Time (FIT) – i.e., 10 errors in 109 hours of operation. Fully autonomous cars fit into the Automotive Safety Integrity Level D (ASIL-D) risk classification – the highest classification level available in the ISO26262 [96] standard for road vehicles' functional safety. Hence, DNN-based hardware targeting such systems must be low-cost and provide at least 99% fault coverage.

## 5.1 Arithmetic Error Codes

Deep Neural Networks layers, such as convolution and fully connected layers, comprise mainly multiply-and-accumulate (MAC) operations. Since more than 90% of the overall computation is spent on these two layers, maintaining MAC units' correct execution heavily impacts the system's reliability. Therefore, adding a fault detection scheme targeting MAC units is imperative.

Coding techniques stand out as good candidates for fault detection. By adding up some redundancy over the data, a fault can be easily detected and, in some cases, corrected. Since the potential fault source is the MAC units, the arithmetic error codes would be a suitable coding technique choice. Arithmetic error codes are a group of codes that can be preserved during the execution of a set of arithmetic operations, such as multiply-and-accumulate ones. Moreover, these codes are of particular interest since they can detect errors at a low cost.

Consider the following as an example of arithmetic error codes. $X$ and $Y$ are input operands of the arithmetic operation $\cdot$. $\hat{X}$ and $\hat{Y}$ the input operands encoded version, respectively. Then, $f$ (Equation 5.1) can be considered as an arithmetic error code with respect to $\cdot$ only and if only the statement in Equation 5.2 holds true. Thus, $f$ over $X$ and $Y$ must yield the same result as the operation over $\hat{X}$ and $\hat{Y}$.

$$f : x \rightarrow \hat{x} \tag{5.1}$$

$$\hat{X} \odot \hat{Y} = (X \cdot Y) \tag{5.2}$$

### 5.1.1 AN Code

The most straightforward way to apply an arithmetic error code is through AN code. AN code is a non-separable type of coding that works by simply encoding the data with the support of a constant $A$. One input operand of an arithmetic operation is multiplied by the constant $A$ before the actual computation (Equation 5.3). After the result between both operands is computed (Equation 5.4), the output correctness can be verified by checking its modulo $A$ (Equation 5.5). If the *outputmodA* differs from zero, the result is not divisible by $A$ and, therefore, wrong (i.e., corrupted).

$$x_c = A * x \quad with\ A > 1 \tag{5.3}$$

$$output_c = x_c \circledast y_c \tag{5.4}$$

$$output_c \bmod A = 0 \tag{5.5}$$

This simple arithmetic coding redundancy scheme provides error detection that is hardware agnostic. Its efficiency only depends entirely on the choice of A [97]. For example, $A = 2$ yields the least expensive encoding scheme but does not offer the best coverage since 2 has many multiples. Errors that modify the result by a multiple of 2 will go undetected. Hence, $A$ should be carefully selected, not only by its detection capability but also by the cost of implementation.

Another characteristic of AN-codes that make them a good fit for fault-tolerant DNNs is the *low-cost* implementation factor. The checking algorithm (Equation 5.5), which only consists of applying modulo to the output, can be significantly simplified by choosing a value of A with the form of $A = 2^p - 1$, for some integer $p$. This number format enables a parallel calculation of modulo $A$ over chunks of $p$ bits them adding them up [18, 32]. In addition to the low cost, $A = 2^p - 1$ numbers are a particular case of prime numbers. By picking $p$ as a Mersenne exponent number[1], the resulting $A$ will also be prime, enhancing the error coverage provided by the coding strategy. Since a prime number has only two factors (divisible by itself and one), only errors that modify the result by these two factors can go undetected.

## 5.2 Error Resilient DNN

The structural and functional properties of DNNs are exploited to implement the AN code detection scheme. As mentioned earlier, more than 90% of the overall computation is spent over two types of layers: convolutional and fully connected. The operations performed in these two layers are depicted in Figure 5.1 and its mathematical form in Equation 5.6. Inputs are multiplied by the weights and then accumulated in a partial sum fashion. This base MAC operation is the heart of every DNN accelerator and can straightforwardly implement AN code. By multiplying any of the operands (input features or weights) by a pre-defined $A$, the resulting partial sum must yield a multiple of $A$ (Section 5.1.1). Therefore, any output of a convolutional or fully connected layer (before bias addition) can be checked by a modulo $A$ operation over its output tensors.

$$partial\_sum_c = \sum_{i=1}^{n} input_{ci} \times w_i \tag{5.6}$$

$$partial\_sum_c = A \times \sum_{i=1}^{n} input_i \times w_i \tag{5.7}$$

A variety of inference DNN hardware platforms has been developed over the past years. Each design comprises specific features to take full advantage of DNNs

[1]`https://oeis.org/A000043`

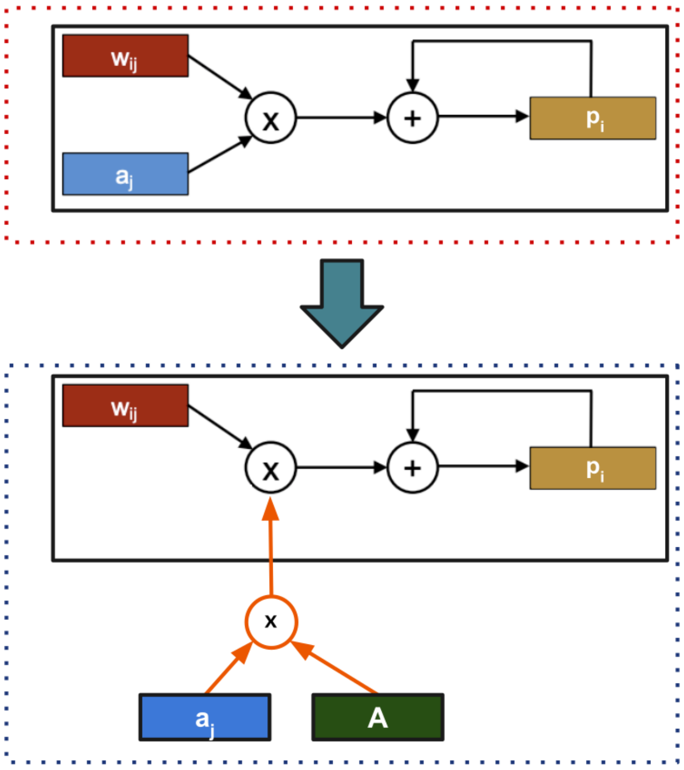


Figure 5.1: Encoding of the input data ($a_j$) by pre-multiplying it by $A$.

specificities to achieve the best performance and energy consumption. These designs can be divided into two main classes of architecture: temporal and spatial. Temporal architectures (e.g., CPU and GPU) follow the idea of a unique centralized control that will trigger several processing elements (PE) to perform a task. PEs access the data from a register/cache array to perform the task in parallel. This parallelism is achieved using single-instruction multiple-data (SIMD) and single-instruction multiple-threads (SIMT), both implemented in CPUs and GPUs, respectively. A spatial architecture follows a different design paradigm. PEs are independent units with their control and small register/cache. PEs can communicate and exchange data/tasks through a specific type of topology. The parallelism is achieved through the mean of data dependency. As the data flows into the PEs, tasks that do not have any input dependency can be triggered.

For the sake of simplicity, consider the Eyeriss DNN accelerator [4, 98] as a spatial architecture example. Figure 5.2 presents a reduced version of Eyeriss, which originally consists of a $14x12$ grid connected to a global buffer, where input feature map and weights are fetched from. A single PE comprises a control unit, local buffer, and the MAC unit. Weights and input features arrive at FIFO buffers and are multiplied and accumulated. Figure 5.2 example consider quantized inputs with 16-bits and outputs with 32-bits. By considering the simplified representation of a

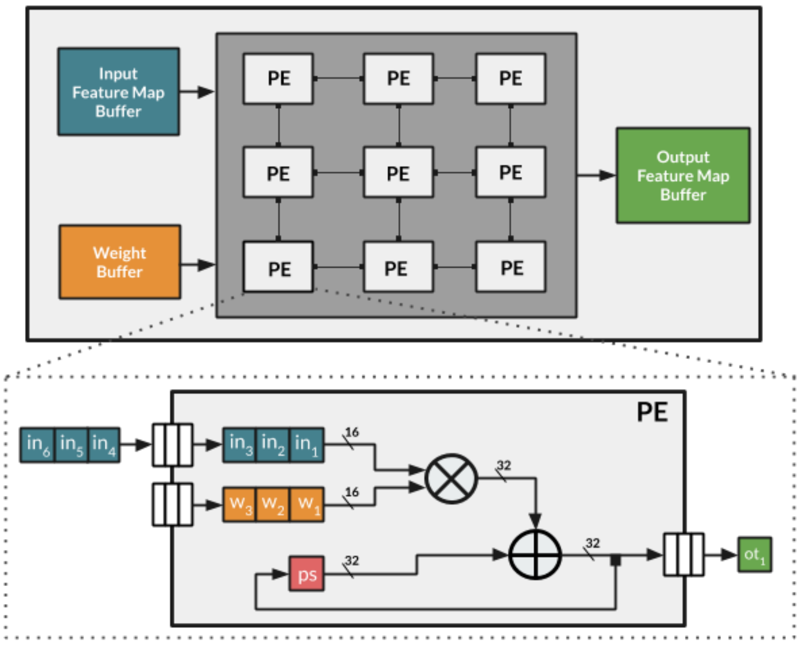


Figure 5.2: DNN system architecture composed by mesh of PE, each one with a multiply-and-accumulate (MAC) unit [4, 5].

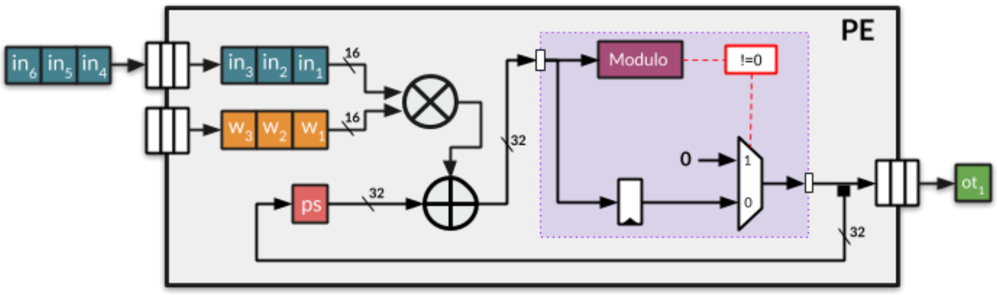


Figure 5.3: PE architecture with the proposed detection scheme (detailed in purple). The path in red shows the control signal for word-masking correction.

PE in Figure 5.2, AN code checking system can be implemented on top of it. Assume that one of the PE inputs is pre-multiplied by $A$, the outcome of the ADD operation can be redirected to a checking system that applies modulo $A$ before storing back the result into the *ps* buffer. Figure 5.3 depicts the same PE but now with an AN code checking system on it.

## 5.3 Fault Model

The fault model in this work considers the cases where errors arise due to intermittent faults (Section 2.2.2). This type of fault occurs repeatedly but with some periods of no manifestation. When manifest, intermittent faults can generate errors that last for a single or several clock cycles [55, 56]. Due to its non-deterministic activation nature, intermittent types of faults are hard to detect and tend to occur in bursts at the same location. The appearance of such faults is an indication of early-stage device wear out [99, 100].

This work considers the particular case of intermittent faults over MAC units of a DNN accelerator. The fault model simulation consists of random bit-level fault injections (FI) over the MAC operations. Since the hardware component can be affected by bursts of faults, the FI simulations comprise different bit error rates (BER) campaigns, following a uniformly random distribution pattern. Moreover, fault injections are restricted to convolutional and fully connected layers only during the *inference phase* of a DNN.

## 5.4 Premises and Scopes

This work targets 16-bit capable MAC units of DNN accelerators, but the proposed detection scheme can be extended to any integer-only inference engine. In this case, the inputs and weights are expected to be integers with 16 bits. To implement AN code over such MAC units, quantization with a lower number of levels must be applied to make sufficient room over the data for the encoding step. Thus, the networks used in the experiment section were quantized using an 11-bits post-training quantization (i.e., no re-training necessary) scheme instead of 16-bits. By following this approach, input features and weights have 5 bits of "free space" left. The extra room not only allows the encoding with a constant $A$ equal to 7 or 31 (3 or 5 bits, respectively) but also enables the scheme to leverage the current 16-bits circuit without hurting the baseline accuracy, as illustrated in Table 5.1.

The encoding can be applied over both input features and weights. However, input features (i.e., activations) are up to 50$\times$ less sensitive to faults when compared with weights [86] since they are not kept for a long time in memory. Therefore, considering only the encoding of the inputs avoids extra costs and reduces the probability of a fault changing the encoded value. Furthermore, previous works have shown that weights can have additional layers of protection at a zero or marginal cost [89, 101, 102].

Note that the additional overheads of the proposed work rely on:

- Pre-multiply the input data, which can be done offline or as soon as the input is received in a real-time system.

- Detection logic inside each MAC unit, which can be performed with chunk-based modulo operation.

## 5.5 Experimental Analysis and Results

A set of empirical experiments were performed to evaluate the effectiveness of the proposed AN-based error detection scheme under the presence of intermittent faults. The experiments were divided into error coverage, the impact of $A$, and error mitigation. The main objective of the first one is to understand how efficient the detection capability can be under different fault rates. Additionally, the experiments will show how good can be the AN coverage for a different set of possible error values. The output of this experiment will define if the proposed scheme is compliant with the ISO26262 [96].

The second category of experiments shows the impact of choosing a different set of $A$'s other than 7 and 31. A group of prime numbers is selected, and their fault coverage and impact on the model's accuracy are analyzed. The result of this experiment is essential to show the trade-off of choosing a Mersenne prime number, which leads to a simple HW implementation, over a pure prime number, with a more costly HW.

Finally, the word masking technique is evaluated to mitigate the impact of a detected error. As errors are detected, the checking system sets the output to zero. This idea is inspired by [89], where weights that are affected by faults (i.e., errors) are set to zero. The results of these experiments are compared to an Oracle solution, where detected faults are "corrected". This analysis shows the effectiveness of a simple and inexpensive solution for error mitigation over DNN MAC units.

The following sections will detail the models, datasets, the experimental setup, and the results of each experimental category.

### 5.5.1 Models and Datasets

The DNN models used during the AN-based experiments are listed in Table 5.1. LeNet-4 and ConvNet are considered small-size models with 4 and 5 convolutional and fully connected layers. AlexNet, VGG16, ResNet18, and ResNet50 are considered mid-size and big-size models with 8, 16, 18, and 50 convolutional and fully connected layers.

Each model presented in Table 5.1 is quantized with 16 and 11 bits through a post-training quantization method (i.e., no re-training needed) provided by the Intel's Nervana Distiller [81] package. Quantization incurs into a marginal decrease in the top-1 accuracy (less than 0.05%).

This set of models was selected due to their variety of the number of layers, weights, and MAC operations performed. Models such as ResNet18 and ResNet50 perform a high number of MAC operations but have few parameters compared to AlexNet and VGG16. Although VGG16 comprises only 16 layers, the massive number of parameters (i.e., weights) and MAC operations make it an interesting case to be analyzed. A set of small models was also selected to understand better the behavior of the proposed detection scheme under faulty conditions. LeNet-4 is the smallest in terms of the number of layers but comprises a good number of parameters and MAC operations. It also achieves the highest top-1 accuracy among all the models. ConvNet has more layers than LeNet-4 but is the smallest in terms of the number of parameters and MAC operations performed.

The 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) validation set was employed as an inference dataset for AlexNet, VGG16, ResNet18, and ResNet50 models. The ILSVRC consists of 50,000 images with 1,000 unique classes. All the networks were pre-trained over $\approx$ 1.2 million images (training set) of 224x224x3 size. Therefore, the images are cropped before being input into the network.

The handwritten digits dataset MNIST was used to evaluate the LeNet-4 model. The model was pre-trained with 60,000 training samples and evaluated over 10,000 samples. The inputs are small-size grayscale images (28x28 pixel box) with values ranging from 0 to 9.

ConvNet is a custom-made mid-size model trained over the CIFAR-10 dataset. It consists of two convolutional layers followed by three fully connected ones. The CIFAR-10 dataset comprises 60,000 small-size color images (32x32 pixel box) with ten classes. The model is pre-trained over 50,000 images and evaluated over the remaining 10,000 ones.

### 5.5.2 Experimental Setup

The experiments were performed with the support of *TorchFI* (Section 3), the custom fault injection framework built on top of PyTorch v.1.3.1. The fault injection campaigns were done over several machines of the Intel Vlab cluster, with each machine equipped with an Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz with 112-cores. To simulate the proposed detection mechanism, *TorchFI* was modified to enable full access to the layer's multiply-and-accumulate operations.

### 5.5.3 Error Detection

The proposed AN-based detection scheme is expected to cover most errors. However, faults that change the output into multiples of $A$ might go undetected. Figure 5.4 presents the fault coverage results considering bit error rates (BER) ranging from $10^{-4}$ to $10^{-1}$. For each model, the simulations are performed by setting the $A$ value to 7 or 31. For a better data visualization, the plots are restricted to the first 100 batches out of 157 in total for MNIST and CIFAR-10 and 782 in total for

Table 5.1: Summary of the six DNN models that compose the benchmark set. Accuracy is measured based on top-1 error. Quantization only incurs less than 0.05% drop on accuracy. Number of weights and MACs are displayed in mega (M) and giga (G) units.

| | Data Type | Top-1 acc. | # Weights | # MACs | Dataset |
|---|---|---|---|---|---|
| **LeNet-4** | fp32 | 99.14% | 1.2M | 36.2M | MNIST |
| | int16 | 99.14% | | | |
| | int11 | 99.14% | | | |
| **ConvNet** | fp32 | 67.88% | 62K | 658K | CIFAR-10 |
| | int16 | 67.92% | | | |
| | int11 | 67.91% | | | |
| **AlexNet** | fp32 | 56.52% | 61M | 724M | ImageNet |
| | int16 | 56.61% | | | |
| | int11 | 56.49% | | | |
| **VGG16** | fp32 | 71.59% | 138.4M | 15.5G | ImageNet |
| | int16 | 71.59% | | | |
| | int11 | 71.58% | | | |
| **ResNet18** | fp32 | 69.76% | 11.7M | 1.8G | ImageNet |
| | int16 | 69.76% | | | |
| | int11 | 69.72% | | | |
| **ResNet50** | fp32 | 76.13% | 25.5M | 3.9G | ImageNet |
| | int16 | 76.13% | | | |
| | int11 | 76.10% | | | |

ImageNet. Each data point (batch) in the figure comprises 64 image inferences, and the remaining batches provide similar results. Although all the simulations were performed with BER starting from $10^{-6}$, these results in Figure 5.4 display mid to **extreme cases** only (BERs from $10^{-4}$ to $10^{-1}$).

The minimum fault coverage is 99.6% for both $As$ up to $10^{-3}$ BER scenarios. As the BER increases, the gap between the $As$ gets more significant, but the mean keeps around 99.6% and 99.3% when $A$ is equal to 31 and 7, respectively. There are sporadic cases where the coverage gets below 99%, mainly over the small-size models. These results indicate that even using a small encoding value such as $A = 7$ over extreme (and unusual) BER of $10^{-1}$, the proposed AN-based detection mechanism can deliver a fault coverage of $\geq$ 99% overall, complying with industry standards such as ASIL-D defined by the ISO 26262 [96]. Additionally, the proposed technique can achieve a near-perfect detection rate of 100% with very low variance for both $A$ under small (and more realistic) BERs.

### 5.5.4 Impact of $A$'s Choice

The choice of $A$ directly impacts the effectiveness of the proposed AN-based detection scheme. $A$ must be a value with the smallest number of factors possible to avoid aliasing. Aliasing is defined as when a fault affects the data such that the resulting faulty value is multiple of $A$. More factors (i.e., multiples) mean a higher probability of aliasing occurrence, reducing the coverage effectiveness. Primes are numbers that cannot be composed by the product of two smaller numbers. This fact dramatically reduces the aliasing probability, making primes a good fit for $A$. Picking a big prime number seems to be the best choice for the encoding step. However, big numbers mean a higher demand for "free" space in the data required by the encoding bits. It also incurs a higher cost for the checking system to compute the modulo $A$. Therefore, a trade-off between coverage effectiveness and implementation cost must be considered.

Figure 5.5 shows the impact of different choices for $A$ (primes and $2^{p-1}$ numbers) over the fault coverage for AlexNet, ResNet18, and ResNet50 models under the BER $10^{-1}$ scenario. Although $A = 29$ provides the highest fault coverage, $A = 31$ differs by only 0.3%. However, due to its binary nature, the implementation of Mersenne prime numbers with the format $2^{p-1}$ is less costly than a number equals to 29 [18] (Section 2.2.4).

By looking at Figure 5.5, the decision seems to be straightforward; picking 31 is the best choice over any other one due to its cost of implementation. The trade of 0.3% of coverage for a reduced cost (i.e., area and energy) seems to pay off. However, that slight difference in the fault coverage can have huge implications for the model's

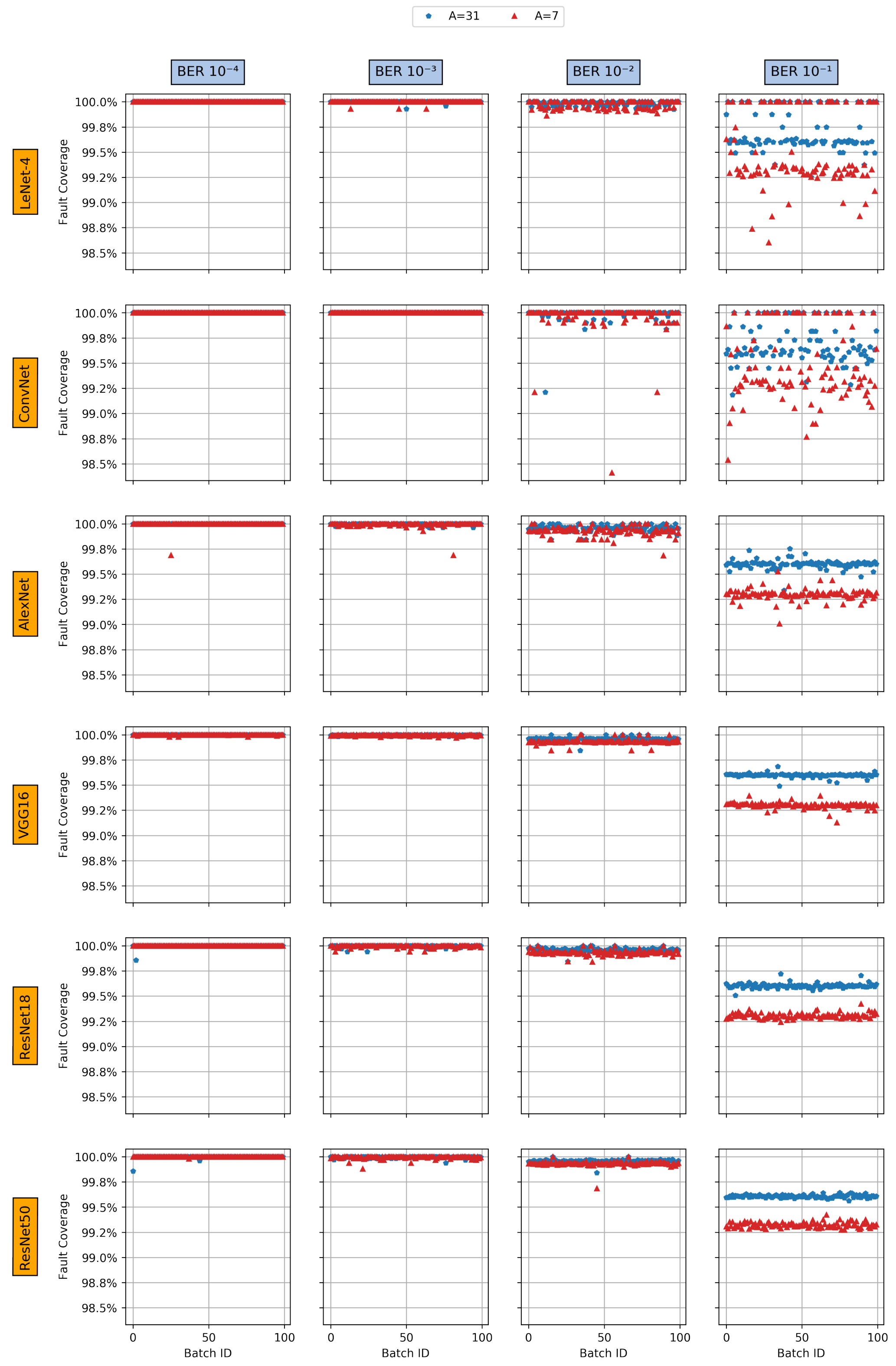


Figure 5.4: Fault Coverage of the proposed mechanism with modulo-7 and modulo-31 configurations. Each batch id point is composed by 64 image inferences. We restricted it to the first set of 100 batches out of 157 total for MNIST and CIFAR-10, and 782 total for ImageNet due to space constraints.

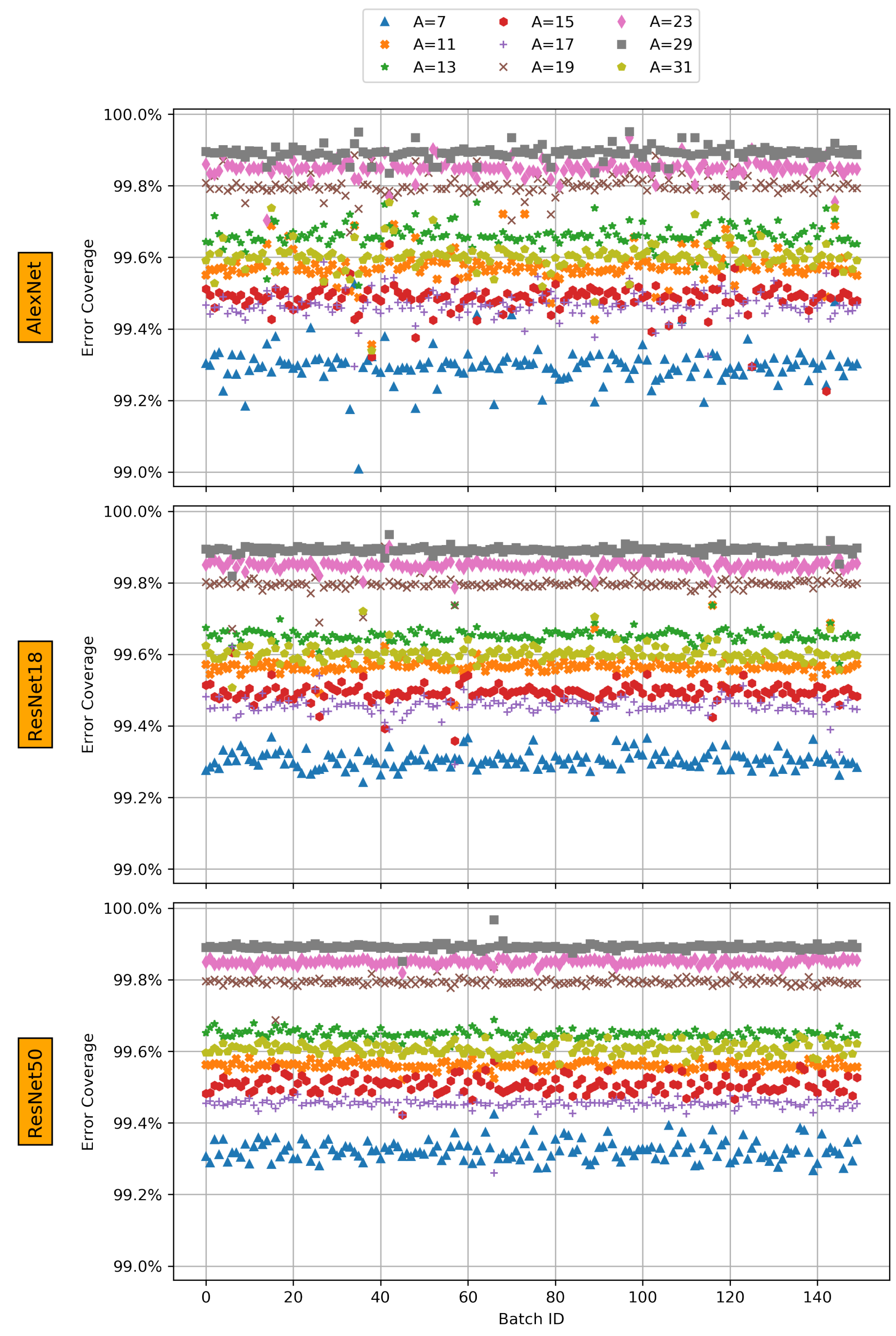


Figure 5.5: Impact of different A's on the fault coverage for AlexNet, ResNet18, and ResNet50 over BER $10^{-1}$.

accuracy. Assuming the detected faults can be corrected, the overall impact of 0.3% reaches almost 20% on the classification error for all the three models as Figure 5.6 shows. **However, this big difference is only observed under extreme conditions such as BER $10^{-1}$. The difference is almost negligible for every BER under $10^{-2}$.** Therefore, $A = 31$ stands as the best choice, delivering the best trade-off between fault coverage and implementation cost.

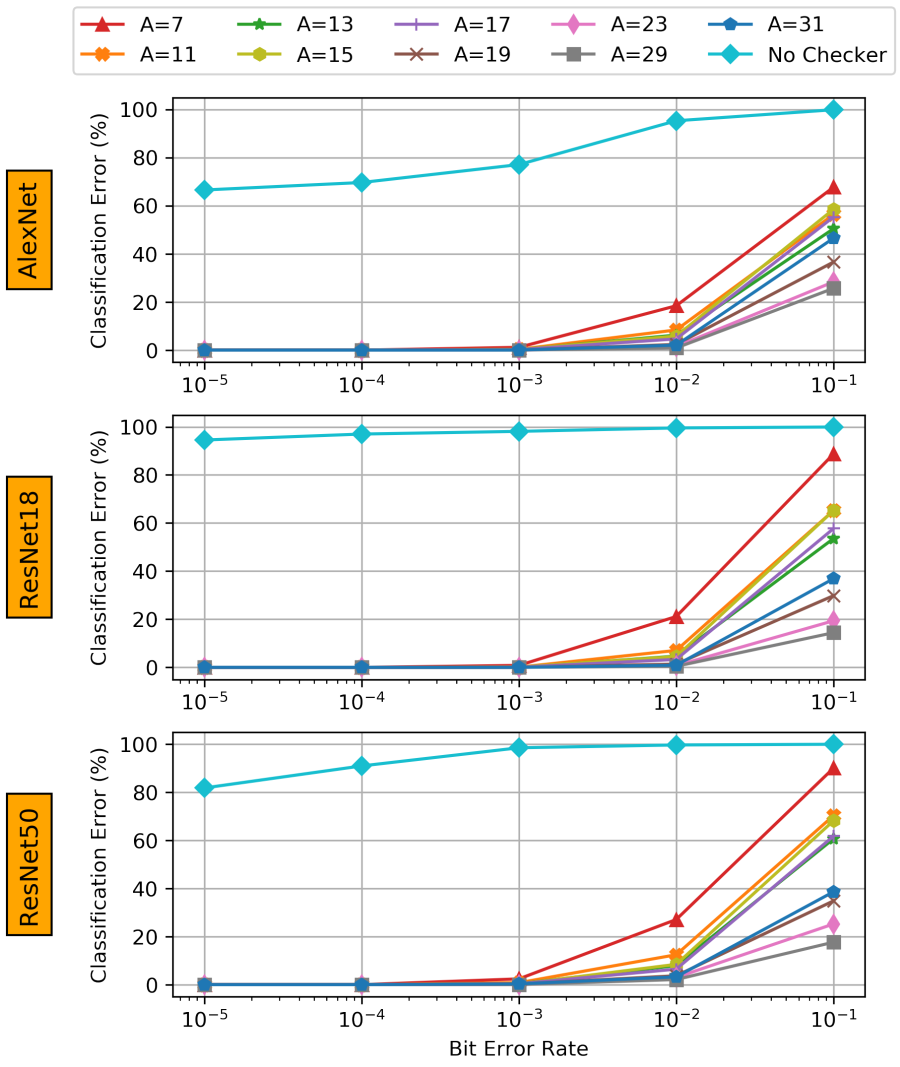


Figure 5.6: Impact of different A's on the accuracy for AlexNet, ResNet18, and ResNet50.

### 5.5.5 Error Mitigation

As observed in the previous analysis, a small fraction of undetected errors can impact the model's accuracy, especially over extreme BERs. Figure 5.4 shows that the proposed detection scheme can achieve more than 99% fault coverage. However, how does 1% of undetected errors translate over each model's final accuracy?

To address this question, a set of error mitigation experiments were performed. The experiments considered three scenarios:

- No Detection (faulty): MAC units have no detection and protection mechanisms. Faults are injected, and the resulting errors are not contained and will flow throughout the whole execution of the network.
- Word Masking (zero): Faults are injected, and once errors are detected, the corrupted outputs are set to zero. This approach is in line with the idea

proposed by Reagen *et al.*[89], where corrupted weights are set to zero.

- Oracle: Once errors are detected, the corrupted outputs are set to their correct/expected values. Only undetected errors due to aliasing will be responsible for any drop in the overall accuracy during this scenario.

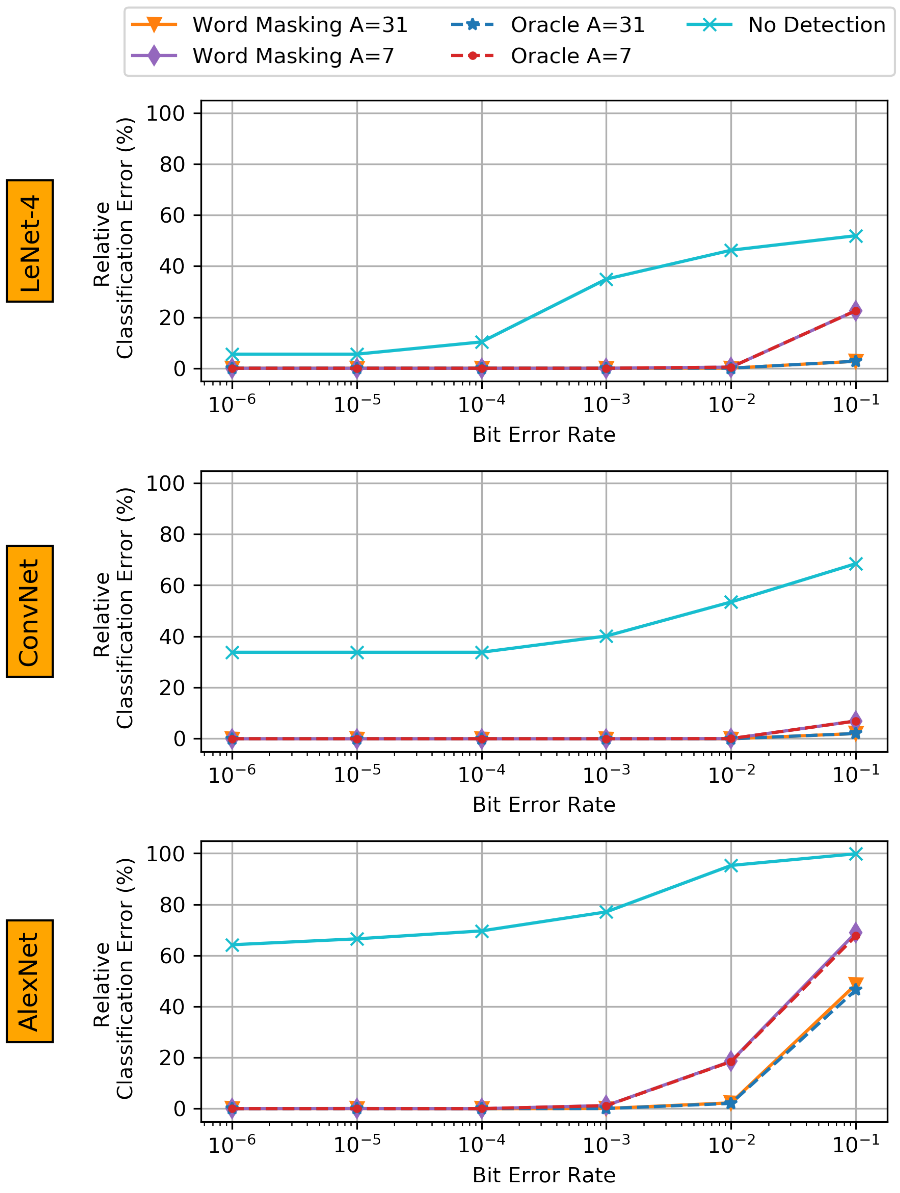


Figure 5.7: Relative classification error over different bit error rates (BER) with no error detection, the proposed detection mechanism coupled with word masking correction scheme, and an oracle for LeNet-4, ConvNet, and AlexNet models.

Figures 5.7 and 5.8 present the **relative classification error** for each DNN model. The BERs ranges from $10^{-6}$ to $10^{-1}$ and $A$ is set to 7 and 31. The *No Detection* scenario drastically increases the classification error of all the models by more than 50% with a BER of $10^{-6}$ and reaches 100% with $10^{-1}$ BER. In this case,

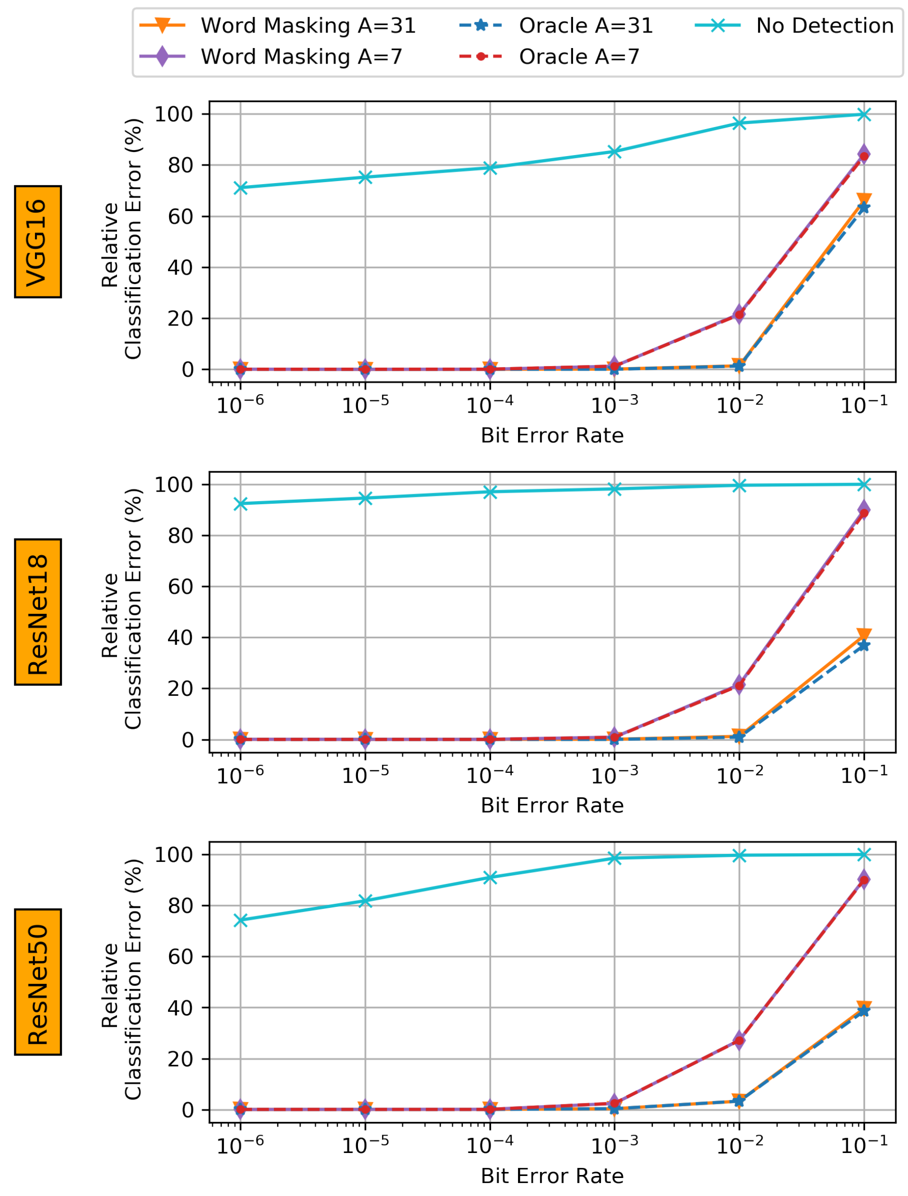


Figure 5.8: Relative classification error over different bit error rates (BER) with no error detection, the proposed detection mechanism coupled with word masking correction scheme, and an oracle for VGG16, ResNet18, and ResNet50 models.

faults go through the whole network undetected, causing silent data corruption over several inferences, causing the overall accuracy of the networks to plummet.

When the AN-base detection is activated, and the *Word Masking* technique is applied, the classification is pushed back close to its original baseline, with up to 0.3% difference. The same results are promising since it almost reaches the *Oracle* version, with a slight increase of 0.75% for $10^{-1}$ BER scenario. The *Oracle* version is considered the best scenario, where a correction scheme is under operation, and only aliasing can hurt the model's accuracy.

For both, *Oracle* and *Word Masking* version, the classification error only starts to increase by $10^{-3}$ BER and $10^{-2}$ BER for $A$ equal to 7 and 31, respectively.

These results encompass the direct impact of the almost 1% undetected error due to aliasing, presented in Figure 5.4.

## 5.6 Concluding Remarks

Safety-critical systems demand high dependability standards. The components must follow a variety of high-grade criteria to be considered safe for end-users. Such types of systems that rely on DNNs accelerators must provide ways to ensure the reliable execution of their applications while delivering high-performance execution at low costs (i.e., area and energy). Therefore, the need for a lightweight solution targeting such systems is of great importance.

The AN-based detection scheme proposed in this work can greatly improve the reliability of DNN accelerators employed in safety-critical systems. The solution provides a lightweight high coverage error detection capability and provides means to mitigate them, sustaining high accuracy results from all the tested models. The detection solution achieves a minimum fault coverage of 99.6% for both $As$ up to $10^{-3}$ BER. Through the *Word Masking* technique, the proposed solution can preserve the MAC units' correct execution without compromising the original model accuracy even at unusually high bit error rates (up to $10^{-2}$ BER with $A = 31$). Therefore, the solution is compliant with the highest classification risk level (ASIL-D) of the ISO26262 [96] standard for road vehicles' functional safety.

## 5.7 Related Work

Hardware reliability is a significant concern among the research and industry community. Over the past years, research on hardware reliability focused on DNN accelerators has received considerable attention. Part of the research is related to the reliability assessment of DNN models regarding faults over register/latches that store weights and input activations. An extensive assessment study of silent data corruption over DNN models and fault propagation of soft errors has been performed by Li *et al.*[84]. Their research explores a different set of possible fixed-point quantization over many models. Reagen *et al.*[86] further enhance the analysis by performing an assessment study of DNNs in the presence of permanent faults. Faults are injected with different bit error rates (BER) over the weights and input activations. Their findings show that deeper models are more prone to experience a more significant loss in accuracy under higher BER. Additionally, they showed that weights were much more sensitive to faults than the input activations for most models they experiment with.

Some attention has been given to fault mitigation solutions over DNNs. The majority focus on solutions to reduce faults over memory units or their propagation effects. Reagen *et al.*[89] employed the *Razor double-sampling* technique coupled with the *Word Masking* technique, where the corrupted data is moved towards zero. By applying both, *Razor double-sampling* and the *Word Masking* techniques, the authors were able to decrease the SRAM supply voltage while keeping the same baseline accuracy of the DNN models. Li *et al.*[84] proposed a *Selective Latch Hardening* mechanism, which, combined with three well-known hardening techniques, can mitigate the propagation of transient faults in hardware. Qin *et al.*[103] proposed a single-bit fault detection technique, where a redundant check-bit is appended to the data. This check-bit is generated in a way that modulo 2 of the sums of the bits must yield zero. After detecting the error, the *Word Masking* technique, like the one proposed in [89], would be applied. Azizimazreah et al. [102] proposed a novel SRAM cell architecture. The Zero-Biased MNU-Aware cell consists of redundant storage cores capable of recovering any node cells that have corrupted soft errors on the DNN accelerators. On the software side, Koppula *et al.*[104] proposed an iterative and progressive error injection retraining process to boost the DNN model's tolerance against errors. The models would *learn* these error patterns through a noise classification training step, avoiding drops in the classification accuracy.

Guan et al. [101] introduced a novel in-place zero-space memory ECC. Through an innovative and enhanced training step entitled Weight-distribution Oriented Training (WOT), the existing *non-informative bits* in DNN's parameters are leveraged to store check bits for an SEC-DED (64, 57, 1) code. In an 8-bit type of model, the technique group seven consecutive weights out of eight that do not require the full 8-bit range and leverages the seven *non-informative bits* for ECC. However, the current technique requires a regularization step so that the spatial distribution of full range weights appears only at specific places.

Targeting DNNs arithmetic units, few prior works have proposed innovative designs for error-resiliency. Zhang *et al.*[105] proposed a novel fault-aware technique for systolic array-based DNN accelerators, which consists of removing faulty MAC units from the systolic grid. This removal step is performed by zeroing out all the weights mapped to the target MAC unit. This approach requires the retraining of the networks considering the new topology (without faulty MACs) and placement. Gambardella et al. [106] proposed a selective channel replication. The technique consists of applying triple-modular redundancy (TMR) over critical channels of the model's network. Through a profiling step, channels are ranked, and the ones more prone to impact the classification accuracy in the presence of faults are triplicated. On the coding side, Feinberg *et al.*[107] proposed an extended version of the ABN correction codes by focusing on a particular type of DNN accelerator, the memristive ones. The proposed solution maps all the possible syndromes into a table that can be further queried for an efficient correction scheme of single and multiple bit errors.

# Chapter 6

# Hardware and Software Co-Design for Resilient Deep Learning Accelerators

The AN-based detection scheme proposed in Chapter 5 has proven to enable error detection at a meager cost. The solution provides a lightweight, high coverage error detection capability by encoding the network's inputs with a pre-defined value $A$. The 16-bit integer-only networks are quantized to 11-bits, making enough room for the encoding data. Detection is performed over the filter granularity by applying a fast and parallel modulo $A$ over each outcome of a MAC operation. Only results that are multiples of $A$ are considered error-free—the word masking technique zeros out the remaining ones (deemed faulty), avoiding further error propagation.

Although the proposed AN-based detection scheme achieves high detection standards at a low cost, it employs severe pre-step restrictions to be deployable. First, the encoding step, where the inputs are pre-multiplied by $A$, adds an overhead. Second, the network must be quantized up to 11 bits to leave sufficient room for the encoding data to be attached. Yet would it be possible to remove the pre-processing step through a software and hardware co-design effort?

The work proposed in this chapter aims to reduce the encoding overhead demanded by the AN-based detection scheme. The model parameters would be adapted, enabling AN-based detection hardware with no further changes needed during the model deployment. The software and hardware co-design approach would adjust the weights based on the deployed AN-based hardware restrictions through a custom DNN quantization aware training process, resulting in an error-checking capability involving practically no overhead.

## 6.1 AN Code-Aware Quantization

Quantization-Aware Training (QaT) is a challenging task that involves the model's training pipeline to achieve enhanced quantization functions. Differently from the Post-Training Quantization (PTQ) employed in Chapters 4 and 5, the QaT must handle the backward propagation of the gradients through quantization blocks in the model network.

$$y = \alpha \left( \sum_{i=1}^{n} s_i \lambda \left( \beta x - b_i \right) - o \right) \tag{6.1}$$

Optimization-based QaT techniques handle the backward propagation problem by employing soft quantizers [6, 49, 50] over the model network. The soft quantizers are learnable differentiable non-linear functions that can be applied to any data of the model (weights and activations). The functions are optimized through several finetune epochs by minimizing the ideal result and the soft quantization function gap. This is a similar process to the training of the network weight, but now the targets are parameters of the quantization function (Equation 6.1).

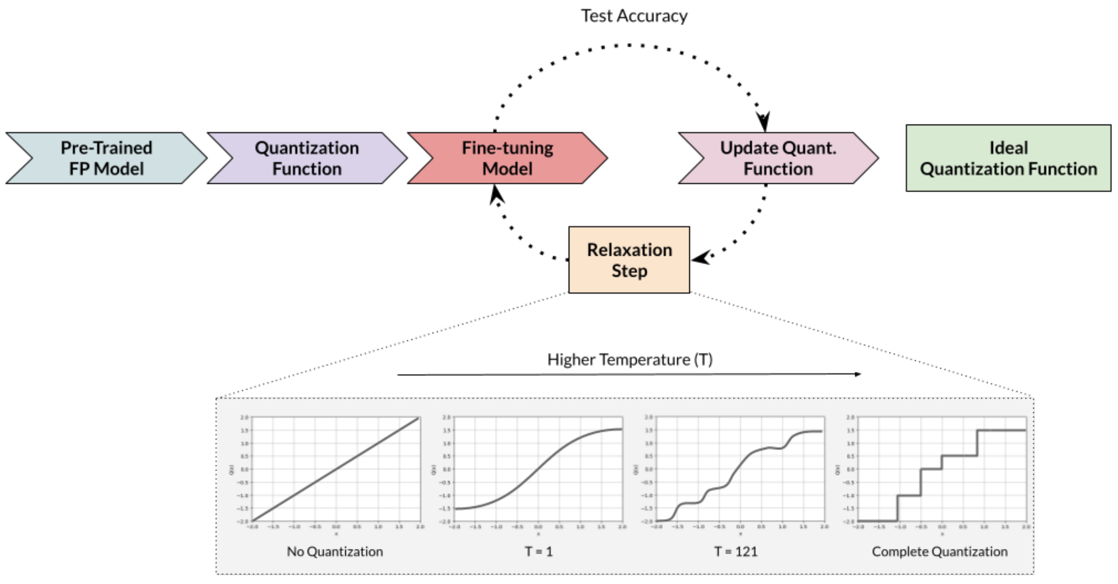


Figure 6.1: Steps for learning a quantization function [6].

The ideal quantization function will be trained/learned during an iterative fine-tuning process with the support of a continuous relaxation method, as illustrated in Figure 6.1, removing the need for a full quantization-aware network training. Since the quantization levels can be pre-determined, a constraint allowing only levels which are multiples of $A$ can be added. If all weights are multiples of $A$, detecting errors becomes a straightforward task by implementing a modulo $A$ checking circuit in the system.

$$\sigma\left(Tx\right) = \frac{1}{1+\exp(-Tx)} \tag{6.2}$$

The proposed AN code-aware quantization process is based on the Quantization Networks method proposed by Yang *et al.*[6]. The method consists of several unit step functions with dedicated learnable parameters. Equation 6.1 shows how the step function $\lambda$ correlates to its learnable parameters $\beta$ (overall scale factor) and $b_i$ (bias of $i$th step) and the remaining variables such as $n$ (number of steps or quantization levels), $o$ (global offset), and $s_i$ (local scale factor). With the ideal parameters, the input data $x$ is then quantized to an integer value $y$. Furthermore, the quantization process is enhanced with a continuous relaxation method, by employing a *temperature factor* $T$ to the step function that increases w.r.t. the training epoch. By considering the Sigmoid function as $\lambda$, the *temperature factor*is employed as seen in Equation 6.2 and Equation 6.1 is updated to Equation 6.3.

$$y = \alpha\left(\sum_{i=1}^{n} s_i\sigma\left(T\left(\beta x - b_i\right)\right) - o\right) \tag{6.3}$$

$$y \subset \Theta \Leftrightarrow y\ mod(A) = 0 \tag{6.4}$$

The method proposed in [6] is modified so that the quantization levels are restricted to multiples of $A$ only. Thus, $y$ must satisfy the relation present in Equation 6.4. Moreover, the number of quantization levels $n$ is reduced since the number of possible values that satisfy the Equation 6.4 does not comprise the full range of integer values. For example, by choosing $A = 7$, the number of quantization levels and their values with 4, 5, 6, and 7 bits are depicted in Table 6.1.

Table 6.1: Comparison between original quantization levels and AN-based one with $A = 7$ and 4, 5, and 6 bits.

| | **# bits** | **Number of Quantization Levels** | **Possible Values** |
|---|---|---|---|
| Original | 4 | 15 | [-7, -6, ···, -1, 0, 1, ···, 6, 7] |
| | 5 | 31 | [-15, -14, ···, -1, 0, 1, ···, 14, 15] |
| | 6 | 63 | [-31, -30, ···, -1, 0, 1, ···, 30, 31] |
| AN-Based | 4 | 3 | [-7, 0, 7] |
| | 5 | 5 | [-14, -7, 0, 7, 14] |
| | 6 | 9 | [-28, -21, -14, -7, 0, 7, 14, 21, 28] |

## 6.2 Experimental Analysis and Results

A set of empirical experiments were performed to evaluate the effectiveness of the proposed AN code-aware quantization process. The experiments consist of training soft quantization functions over a set of pre-trained models. The soft quantization functions are trained considering a subset of $A \in \{3, 7, 31\}$ and the number of bits $b \in \{4, 5, 6, 7, 8\}$. The main objective is to evaluate if the models are capable of converging and reaching the baseline accuracy, where the model has been trained with no quantization (32-bit floating-point data type).

After reaching full convergence and achieving near-optimal soft quantization functions, the model would be attached to a PTQ step to quantize the incoming inputs to $n$-bits integers (where $n \in b$). This process would enforce that all the data flowing into the MAC units of the DNN hardware are $n$-bits integers only, allowing the AN-based detection scheme to work as expected.

### 6.2.1 Models and Datasets

The DNN models used during the AN code-aware quantization experiments are listed in Table 6.2. LeNet-5 is a small-size model with 5 convolutional and fully connected layers. ResNet18, ResNet20, ResNet34, and ResNet50 are considered mid-size and big-size models with 18, 20, 34, and 50 convolutional and fully connected layers.

The 2012 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) validation set was employed as an inference dataset for ResNet18, ResNet34, and ResNet50 models. The ILSVRC consists of 50,000 images with 1,000 unique classes. All the networks were pre-trained over $\approx$ 1.2 million images (training set) of 224x224x3 size.

The handwritten digits dataset MNIST was used to evaluate the LeNet-5 model. The model was pre-trained with 60,000 training samples and evaluated over 10,000 samples. The inputs are small-size grayscale images (28x28 pixel box) with values ranging from 0 to 9.

ResNet20 model was trained over the CIFAR-10 dataset that comprises 60,000 small-size color images (32x32 pixel box) with ten classes. The model is pre-trained over 50,000 images and evaluated over the remaining 10,000 ones.

### 6.2.2 Experimental Setup

The experiments were performed with the support of the PyTorch framework version 1.6 on a workstation equipped with dual Intel(R) Xeon(R) Gold 6246 CPUs @ 3.30GHz, 256GB RAM, and two NVIDIA Quadro RTX 8000 GPUs. The code

Table 6.2: Summary of the five DNN models that compose the experiment set. Accuracy is measured based on top-1 error and considers the original model baseline trained with 32-bit floating-point data type.

| | **Data Type** | **Top-1 acc.** | **Dataset** |
|---|---|---|---|
| LeNet-5 | fp32 | 98.75% | MNIST |
| ResNet20 | fp32 | 91.94% | CIFAR-10 |
| ResNet18 | fp32 | 69.76% | ImageNet |
| ResNet34 | fp32 | 73.31% | ImageNet |
| ResNet50 | fp32 | 76.13% | ImageNet |

repositories in [1] and [2] were leveraged to perform the AN code-aware quantization process.

### 6.2.3 Effectiveness Analysis

The AN code-aware quantization process involves numerous training trials of each model with different configurations. Hyperparameters, such as learning rate, number of epochs, weight decay steps, temperature, and optimization algorithms, must be finetuned to achieve near-optimal results. Therefore, the experiment starts with proof-of-concept models such as LeNet-5 over the MNIST dataset and ResNet20 over the CIFAR10 dataset. After evaluating the best configuration results over the proof-of-concept models, they are used as baseline metrics over the remaining (and bigger) models.

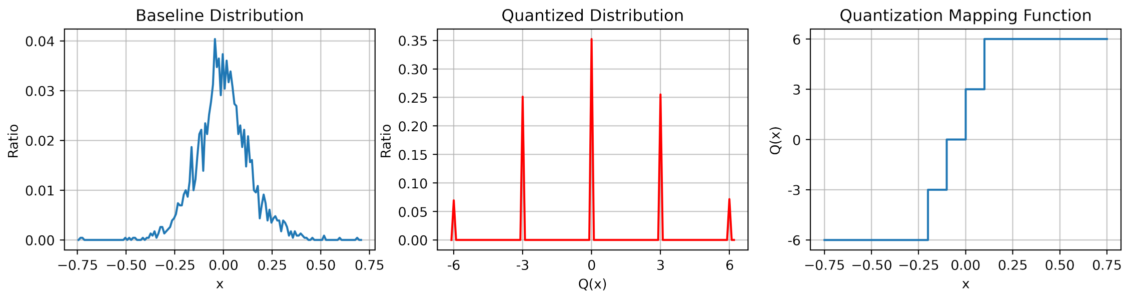


Figure 6.2: Parameter distributions of Baseline (left) and Quantized (middle), and the soft quantization mapping function (right) of the ResNet-20's fourth layer with $A = 3$ and 4 bits.

Figures 6.3, 6.4, and 6.5 show the learning curves for LeNet-5 model with $A$ equal 3, 7 and 31, respectively. For each $A$ configuration, a different set quantization levels is tested, considering the total number of bits $b$. All the configurations converge at

[1]`https://github.com/aliyun/alibabacloud-quantization-networks`

[2]`https://github.com/linkinpark213/quantization-networks-cifar10`

the end of 200 epochs, reaching the top 1 baseline accuracy of the model. The models are update the model by the adaptive learning rate optimization Adam [108]. The initial learning rate is set to $1e^{-3}$ and decayed by 0.1 at epoch 20 with Temperature set to 5.

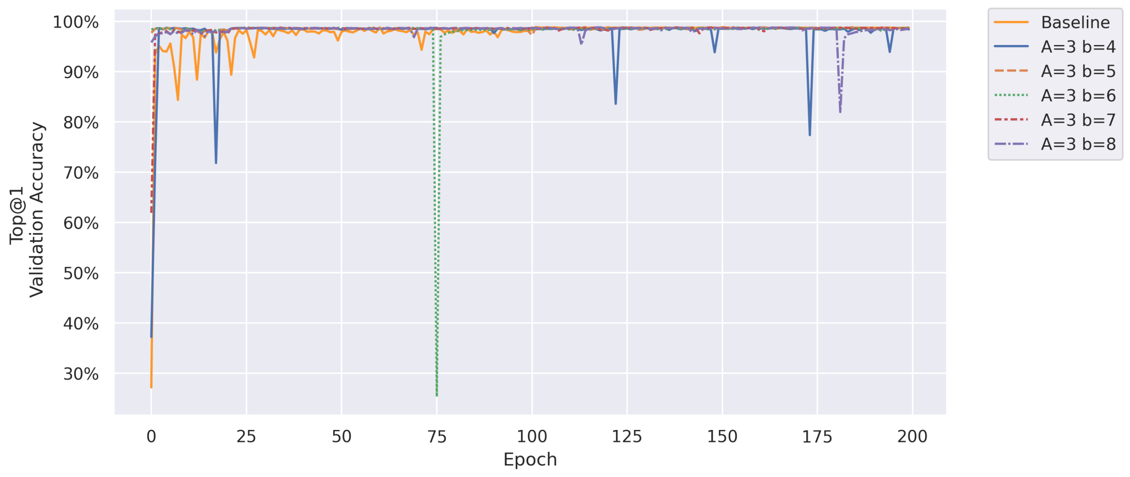


Figure 6.3: Learning curves for LeNet-5 with $A = 3$ and different levels of quantization defined by $b$ bits

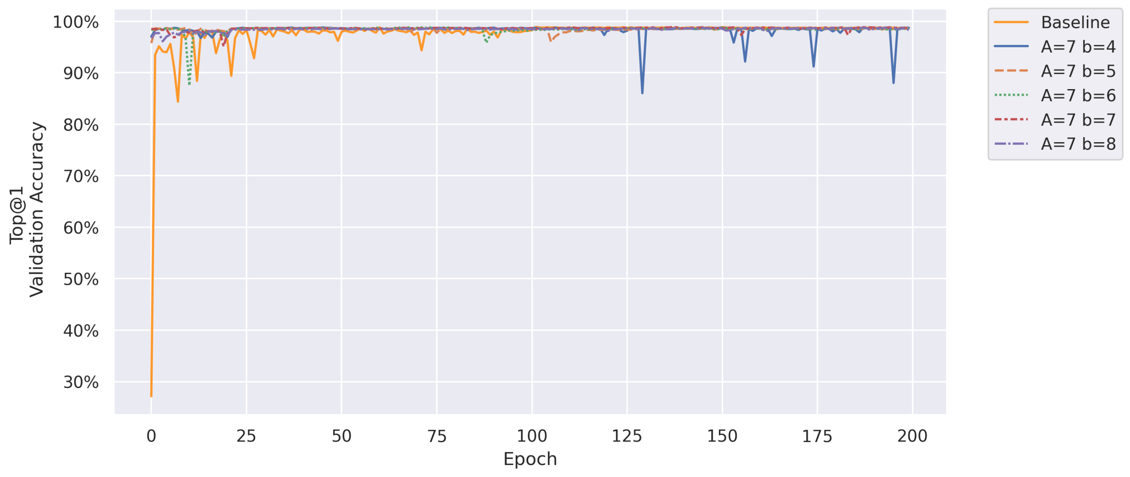


Figure 6.4: Learning curves for LeNet-5 with $A = 7$ and different levels of quantization defined by $b$ bits

For the second proof-of-concept experiment, the ResNet20 model is trained over the CIFAR10 dataset with the same set of configurations for $A$ and $b$. The stochastic gradient descent (SGD) algorithm [109] updates all the model configurations with momentum set to 0.9 and weight decay to $5e^{-4}$. The learning rates are initialized to $1e^{-1}$ and decay by 0.1 at epochs 4, 100, and 150. The model is trained for 200 epochs in total.

Figures 6.6, 6.7, and 6.8 show the resulting learning curves for ResNet20. All the models converge well but only configurations with $A$ equal 3 and 7 can reach

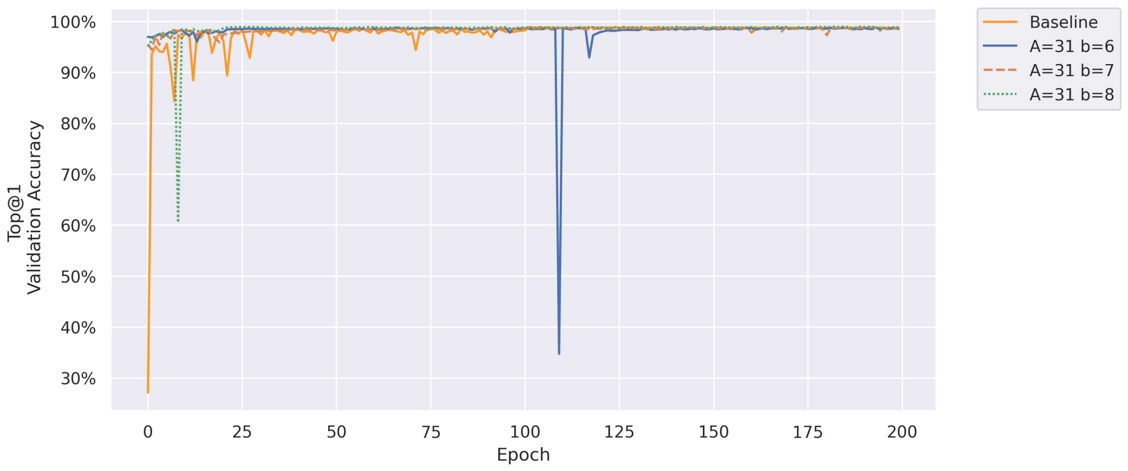


Figure 6.5: Learning curves for LeNet-5 with $A = 31$ and different levels of quantization defined by $b$ bits.

near-baseline accuracies ($\leq 1\%$). ResNet20 with $A = 31$ reaches up to 88.99% out of 91.94%. Still, the results are very promising.

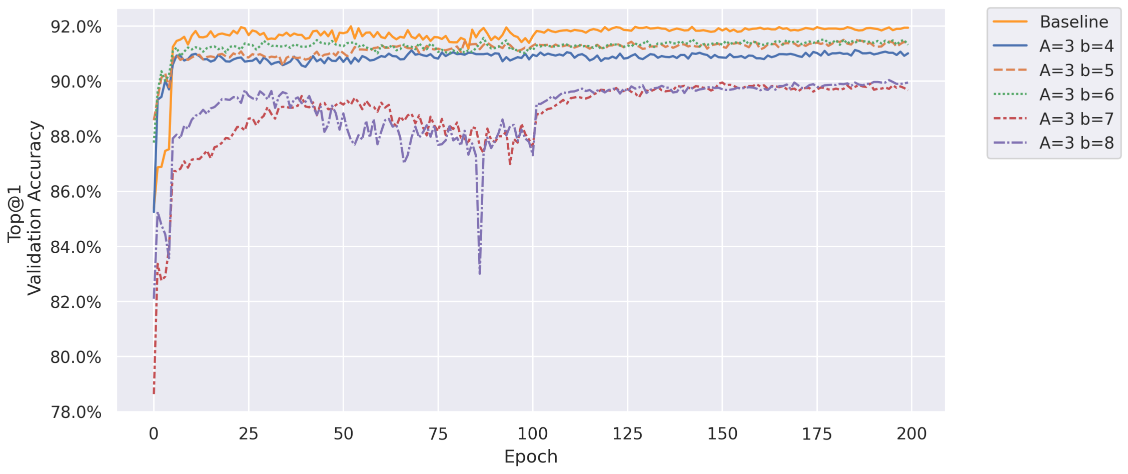


Figure 6.6: Learning curves for ResNet20 with $A = 3$ and different levels of quantization defined by $b$ bits.

Table 6.3 summarizes the best results of experiments considering the whole set of models and configurations. Except for ResNet50, all the models achieved near-baseline accuracies with $\leq 1\%$ of difference. ResNet50 with $A = 7$ and $b = 4$ felt a little behind the baseline (1.72%) but still achieved reasonable results because all the weights are constrained to only 3 levels of quantization ($[-7, 0, 7]$).

Note that higher accuracies are possible to achieve, but due to the hyperparameter search space, time, and the computational cost to run such experiments, this task is out of the scope of this work.

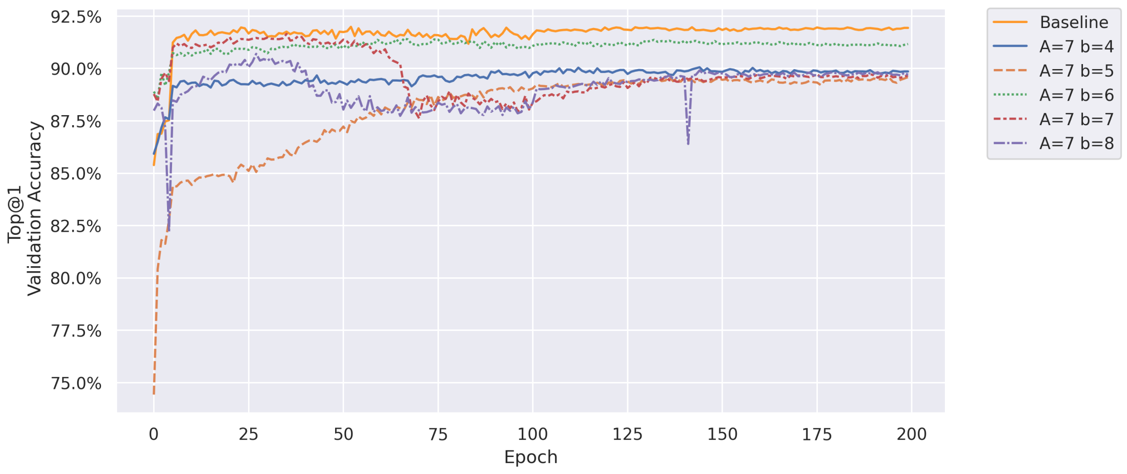


Figure 6.7: Learning curves for ResNet20 with $A = 7$ and different levels of quantization defined by $b$ bits.

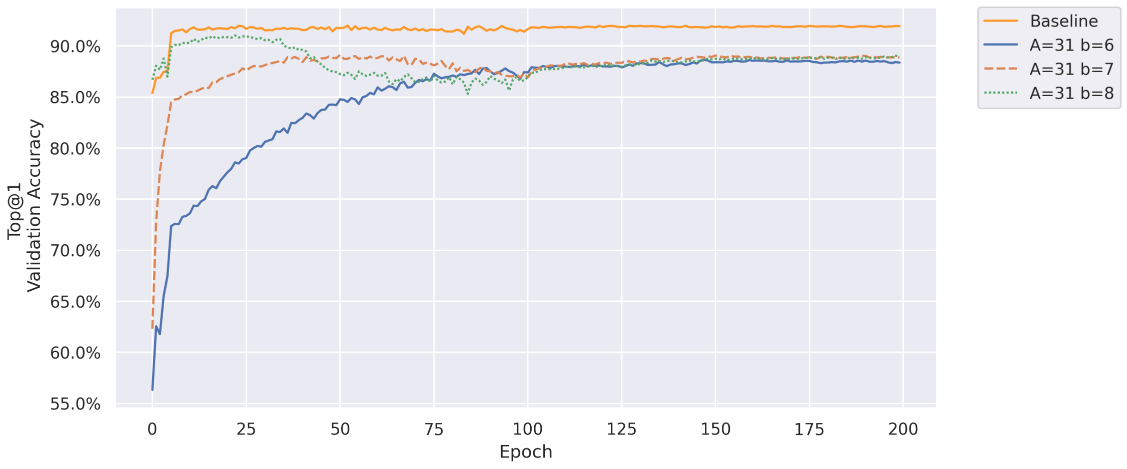


Figure 6.8: Learning curves for ResNet20 with $A = 31$ and different levels of quantization defined by $b$ bits.

Table 6.3: Best top-1 accuracy results with AN code-aware quantization process over the bechmark set.

| **Model** | $A$ | $b$ | **Baseline Acc.** | **Quantized Acc.** |
|---|---|---|---|---|
| LeNet-5 | 3 | 5 | 98.75% | 98.76% |
| | 7 | 4 | 98.75% | 98.74% |
| ResNet20 | 3 | 6 | 91.94% | 91.43% |
| | 7 | 6 | 91.94% | 91.17% |
| ResNet18 | 3 | 5 | 69.76% | 69.84% |
| | 7 | 4 | 69.76% | 68.00% |
| ResNet34 | 3 | 4 | 73.31% | 72.35% |
| ResNet50 | 7 | 4 | 76.13% | 74.41% |

## 6.3 Concluding Remarks

The AN-based detection scheme provides robust error detection at a small cost. However, this technique demands severe pre-step restrictions in order to be a deployable feature. By removing these restrictions through a software and hardware co-design method, the AN error-detection capability would incur practically no overhead over the system. By AN code-aware quantization process with the support of a continuous relaxation method, the weights of a model are constrained to integers multiples of $A$ only. Removing the pre-encoding step of the previous AN detection scheme and drastically reducing its overhead. Empirical experiments with well-known models show that the proposed AN code-aware quantization scheme converges for all the models. For most scenarios, the quantized models can achieve near-optimal accuracies with $\leq 1\%$ of difference from the 32-bit floating-point baseline model, and up to 1.72% variance. This novel technique enhances the AN-based detection scheme, making it feasible for real-world deployment.

# Chapter 7

# Preventing DNN Model IP Theft via Hardware Obfuscation

Deep Neural Networks (DNNs) have attracted considerable attention due to state-of-the-art (SOTA) accuracy in tasks such as image processing (e.g., classification, object detection, tracking, etc.), recommendation, and natural language processing (NLP) [110–113]. However, achieving such levels of accuracy and robustness over DNN models demands a huge effort. This expensive endeavor ranges from the amount of training data provided to the expertise in the domain required by who is training the model. Therefore, DNN models are considered and treated as valuable storehouses of intellectual properties (IP) and must be protected against attacks.

The process of training a deep learning model using the supervised learning method starts from the data collection. More data can roughly translate to a better generalization and robustness of the final model. Data should be carefully curated and manually labeled to provide the best and most precise input for the models. The labeling process incurs a significant amount of human work and access to curated data is a privilege of few in the era where data is a new commodity.

Additionally, a considerable amount of high-performance computing resources coupled with human domain expertise is crucial. SOTA models are typically huge in terms of the number of layers and parameters and the computing resources needed to run these models over a big training set are substantial. The domain expertise comes into action to craft the model and understand its behavior during the training phase. Hyperparameter fine-tuning process is a hard task and must be performed by highly skilled people to avoid future costs (unnecessary resource usage due to exploratory parameter space) and achieve the highest accuracy within the model.

In 2016, Tramèr *et al.*[114] presented a *model extraction* attack at the USENIX security conference. The attack consists of cloning DNN models through simple access to service levels API provided by cloud-based Machine-Learning-as-a-Service (MLaaS) companies such as Amazon AWS and BigML. Without any prior knowledge of the model's parameters or training data, the adversary was able to steal the model and use it privately. This type of attack is known as *model stealing* or *model piracy* attack.

The model stealing/piracy attack scenario gets even worst when the cloud-based provider allows the model (trained over the cloud) to be downloaded to an edge device (e.g., smartphone) for further usage (e.g., face recognition). Through physical access, attackers can steal the model's architecture and its hyperparameters, allowing a near-perfect clone of the original model to be trained [23].

In brief, accurate DNN models are expensive to develop and hold valuable information. As intellectual properties, protecting such models against stealing/piracy attacks is of paramount importance. Both hardware and software layers must provide ways to secure such IP.

## 7.1 Neural Model Obfuscation

Neural model obfuscation is a protection scheme that adds a security layer on top of the original neural network model to protect it against structure and parameter piracy. The security layer locks the model such that only authorized users can achieve the expected high prediction accuracy. On the other hand, non-authorized users would be unable to *unlock* the model, resulting in significant penalties over the model's accuracy. The obfuscation can be performed over various techniques that can range from key-dependent training processes to full cryptographic encryption schemes.

Xu *et al.*[23] proposed the first neural model obfuscation technique in 2018. The main idea is to achieve structural obfuscation through a *joint training* process where the original network would act as a *"teacher"* to a smaller *"student"* model. The *Teacher-Student* approach achieves an effective structural obfuscation but still leaves the *"student"* model unprotected. Non-legitimate users who access it can still run the model without penalties (e.g., accuracy drop) [1].

Abhishek *et al.*[7] proposed a novel *key-dependent* training obfuscation technique that protects the model by employing an enhanced backpropagation algorithm during the model's training phase. The method consists of a random set of selected neuron nodes that are *locked*. A key is generated and associated with the selected nodes, which only provide the correct results with the given right key. Unauthorized parties, which do not possess the valid key, are not able to *unlock* the neuron nodes, and the model behaves poorly during the prediction (i.e., low accuracy). A

[1]Although the *Teacher-Student* approach does not provide the necessary protection of the model from non-authorized users, it has been important for model compression area. Neural Architectural Search (NAS) techniques employ a *Teacher-Student* approach to create a reduced version of the model with the same top1 and top5 accuracies.

hardware root-of-trust is demanded to enable such a protection scheme to act as a trusted key source.

Secure cryptographic techniques, such as Homomorphic encryption [115], can also be employed as a protection layer over the models. These techniques consist of fully encrypting the models before delivering them to the end-users. Only authorized users would be able to decrypt them or execute the encrypted model. However, these techniques incur significant overheads since models must be decrypted on-the-fly while loaded from memory to the execution unit or run over the encrypted space to assure that no side-channel attack would occur.

The neural model obfuscation proposed in this work does not prevent the attacker from stealing the model, but it avoids the expected behavior of the model without a proper key. In other words, non-authorized access to the model will degrade its accuracy, making the stolen data useless. This work restricts the attack model to a man-at-the-end (MATE) attack type. In this scenario, the attacker has complete control with physical access to the edge device and can run DNN models over its Neural Processing Unit (NPU) [116]. End-user devices securely acquire the models through trusted parties, entitled Model Providers. By reverse-engineering the end-user device, the attacker can access the whole model, such as its topology and parameters, but **not** the mapping key (more details in Section 7.2).

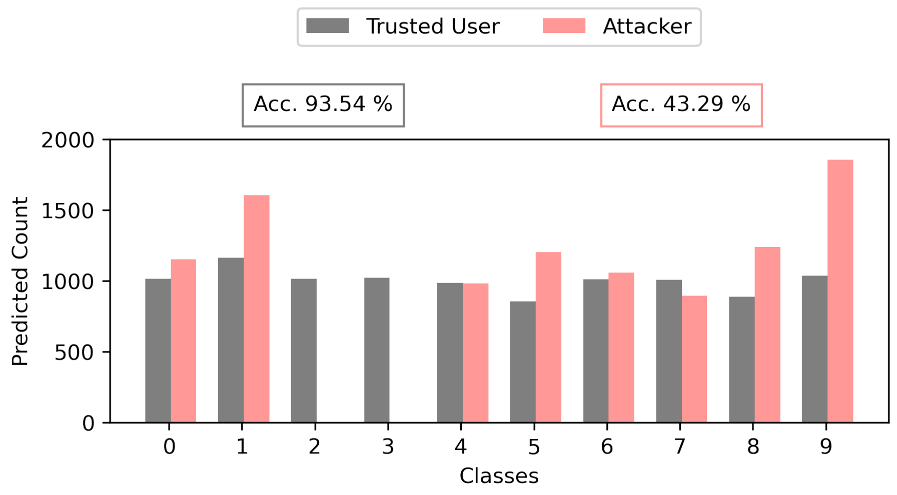


Figure 7.1: Output class distribution of HPNN framwork's key-dependent backpropagation [7] using LeNet-4 model over MNIST dataset.

The attacker's primary goal is to steal the model and reuse it privately, adding to its portfolio or for illegal distribution. Therefore, the model obfuscation technique must provide layers of security to avoid the expected use of such models by non-authorized parties. Moreover, information leakage, presented in other obfuscation techniques [7] (Figure 7.1), must be contained by the proposed obfuscation technique, avoiding revealing that the stolen model has been locked.

## 7.2 Swap-based Model Obfuscation

Training-based techniques can achieve robust obfuscation over DNN model parameters. However, the training process incurs a costly (e.g., energy and computation resources) and time-consuming step to guarantee an additional layer of security. By adopting such techniques, model providers must re-train their entire portfolio of DNN models from scratch for each unique key generated, turning training-based approaches not scalable.

Additionally, empirical analysis (Figure 7.1) show that key-depended backpropagation algorithm [7] leaks crucial information that could indicate that the target model has been locked. The attacker can infer if the model was compromised by running an output class distribution analysis. Figure 7.1 shows a strong disturbance over the LeNet-4 model output distribution after applying the HPNN framework's key-dependent backpropagation algorithm [7]. The main objective of the proposed model obfuscation is to preserve as much as possible the original output distribution though making the classification as erroneous as possible for non-authorized users (i.e., wrong keys).

The proposed swap-based model obfuscation idea focuses on the basic building block of every deep convolutional neural network: the convolutional filters. As shown in Figure 7.2 and described in Section 2.1.2, DNNs can be seen as a stack of layers, with each layer holding a set of learnable weights (a.k.a parameters). The weights are spread over filters on convolutional layers, responsible for extracting the low and high-level features from the input image. They are the heart of convolutional-based DNNs, and their weights have local and global relevance over each input image provided during the training phase. Filters must have a fixed placement and a fixed sequence of execution (defined by the layers) to provide an accurate classification during the inference phase. Therefore, having the model data (i.e., architecture and weights) on hand does not guarantee that the model will perform as expected.

The proposed swap-based model obfuscation takes advantage of the unique filter placement to deliver a strong and lightweight model obfuscation technique. It consists of three possible approaches that provide different obfuscation effectiveness and impact on the outputs class distribution. The three approaches entitled *(a)* full filter swaps, *(b)* row/column swaps, and *(c)* hybrid swaps, are depicted in Figure 7.2.B. The full filter swap consists of swapping entire pairs of filters within the same layer. The row/column approach works at fine-grain scope by swapping rows or columns of different filters within the same layer. The last is a mix of filters and rows/columns swaps, creating a hybrid approach. The proposed technique can achieve high entropy by iterating the swap process over the whole set of layers. The incorrect placement within a layer will generate noise classification that will increase

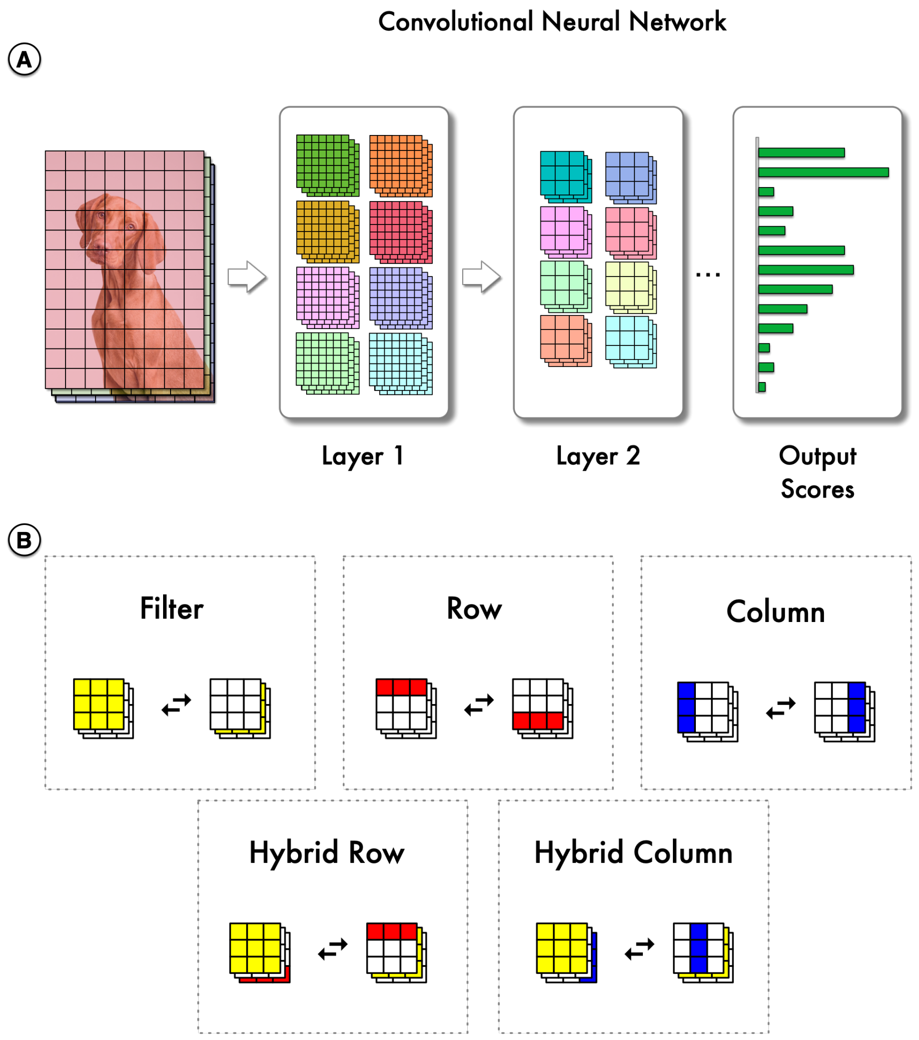


Figure 7.2: (A) Convolutional Neural Network layer's structure. (B) Model obfuscation techniques (filter, row, column and hybrid swaps).

further as the swaps occur in the remaining layers. This noise directly impacts the model's classification accuracy and is dictated by which swap-based approach is applied over the layers and the number of swaps.

The proposed swap-based obfuscation scheme can be further divided into two modes: stealth and full-defense. The stealth mode provides obfuscation but considers the impact of the swap noise over the final class distribution. The final accuracy penalty might not be as substantial as the full-defense mode. Still, it minimizes the information leakage, covering any traces that the model has been compromised by preserving the output class distribution. The full-defense mode ignores the extra information that a non-authorized user can obtain and locks the model by dropping its accuracy by significant percentages. This high level of protection has the side effect of producing similar disturbance over the output class distribution as seen with the training-based schemes. The final decision of which obfuscation mode should be applied is tied to the final user objectives (i.e., model providers).

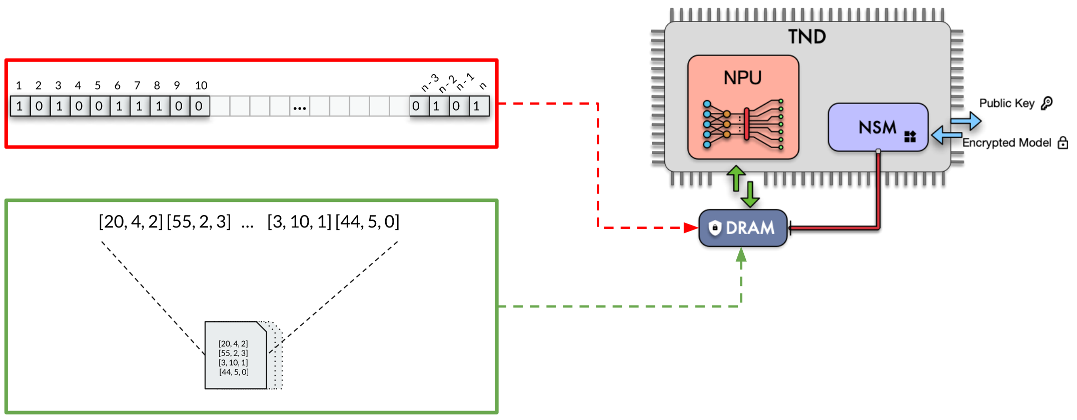


Figure 7.3: Necessary metadata to be safely delivered to the NPU. The secret key must be kept safe while the mapping control file can be disclosed without compromising the obfuscation security.

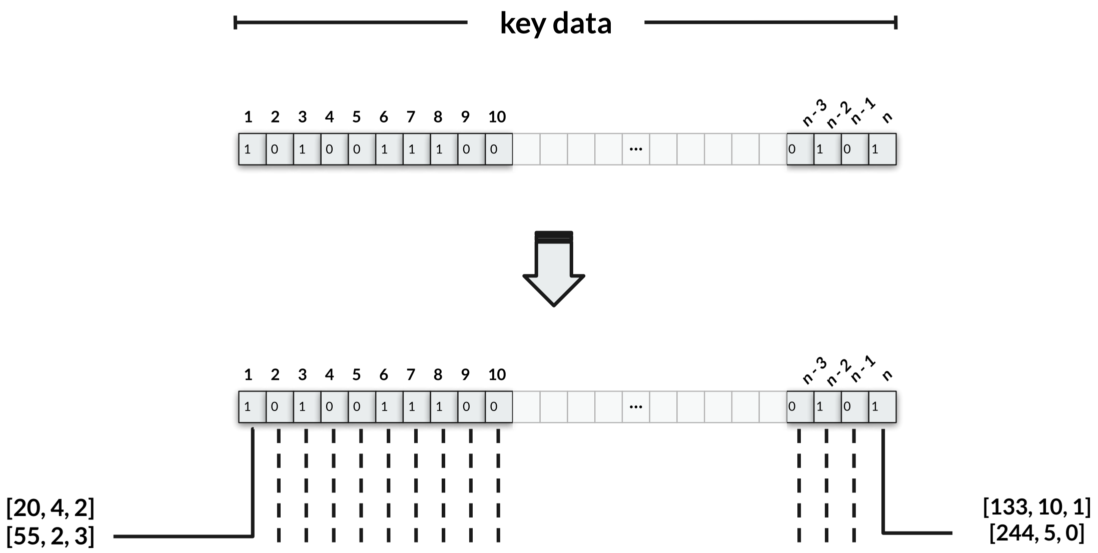


Figure 7.4: Example of secret key and mapping control file relation. Each key corresponds to a pair of tuples with indexes from the structure to be swapped. An active bit in the key means that the structures must be swapped.

The metadata required by the swap-base obfuscation is composed of a binary secret key (e.g., 128-bit key) and the mapping control file. The mapping control file holds the information on which pairs of structures might be swapped (i.e., tuples of indices). The binary secret key informs which of these pairs of structures must be swapped, where a swap is indicated by an active bit (i.e., 1). Figures 7.3 and 7.4 shows an example of mapping control file, a secret key, and how they are related.

Model providers can randomly generate the secret key during the pre-processing step of the swap-base obfuscation scheme. Only end-users holding the correct secret key and the mapping control file associated with it can unlock the model's inference. If a potential attacker owns the topology and parameters of the model but not the metadata, the model will misbehave, resulting in accuracy penalties.

The proposed swap-based obfuscation is not only a robust solution but also scalable. It only requires a pre-processing step where the swaps are performed, and the model's accuracy and output class distribution are evaluated with inference only passes. The proposed solution avoids re-training the whole model portfolio from scratch, enabling a straightforward production of obfuscated models with unique mapping keys for each end-user device.

### 7.2.1 Hardware Support

State-of-the-art NPUs implement a parallel spatial architecture through the support of a systolic array-based design where a grid of connected processing elements (PEs) fetches data from buffers and performs multiply and accumulate (MAC) operations in a pre-defined sequence. Commercial examples of such NPUs are the Google TPU, the ARM Ethos, and the Samsung Exynos [117–119].

Through the support of a weight decoder unit, the ARM Ethos [118] chip manages the DMA controller's weight stream. This unit decompresses the weight on the fly by reordering, padding, aligning them into blocks, and further storing them into double-buffered registers that will feed the MAC units. The Samsung Exynos [119] is equipped with a double data-staging unit (DSU) where the weight stream decompression process occurs. A dispatcher unit receives the decompressed data and forwards it to the MAC arrays within their respective feature map data. In the Google TPU [117], a central unit, called weight fetcher, is responsible for fetching the weights directly from the DRAM. It reorders the data and feeds the massive matrix multiplication unit (MxM).

To implement the proposed swap-based obfuscation technique, a custom preloading step is required. The current weight decoders presented in the state-of-the-art NPUs can be leveraged to perform the necessary steps. First, a dedicated NPU register must be available to store the binary secret key securely. This key is shared between layers, and it is a single-time fetch. Second, as the weights are loaded and decoded, pairs of indices from the mapping control file will also be fetched. The information on the secret key plus the pairs of indices will dictate how the weight decoder unit should proceed (i.e., swap or not swap). Without the proper binary secret key and the mapping control data, the weight decoder unit will swap wrong structures, dispatching incorrect weights into the PEs, causing the model to misbehave. This scheme also creates an *walled garden ecosystem*, forcing the NPU to work only with models from verified parties, closing the edge-device platform to only trusted neural devices (TND). [2]

### 7.2.2 Model Obfuscation Overhead

The swap-based model obfuscation is not an overhead-free technique. As mentioned earlier, a secret key and a mapping control file must be generated during the pre-processing step. Swaps are then performed, and the model's accuracy and output class distribution are evaluated through a set of inference calls (no backpropagation required). These steps can be performed iteratively until the outputs meet the model provider criteria.

The secret key is an $n$-bit key where each bit represents one swap. The total of swaps is tied to $n$ and the number of convolutional layers in the network since layers share the same key. The key should be safely delivered and stored in the NPU register. The mapping control file is a list of pairs of tuples (i.e., indices) that are strongly tied to the mapping key. Only part of the tuple pairs is swapped (i.e., bit key equals 1). Therefore, the mapping control content can be disclosed with no harm to the obfuscation scheme.

## 7.3 Experimental Analysis and Results

A set of empirical experiments were performed to evaluate the effectiveness of the proposed swap-based obfuscation scheme. First, the obfuscation performance is assessed by analyzing each swap's technique (filter, row, column, and hybrid swaps) and their impact on the model accuracy. This composes one of the pre-processing steps that a model provider must perform. The second part is the information leakage analysis, where the perturbation over the output class distribution is quantified. A new perturbation metric is proposed, and each swap's technique is evaluated considering the magnitude of this metric. The third set of experiments performs a security analysis of the swap-based obfuscation scheme. This analysis simulates two types of attacks over the obfuscated models. A brute-force attack with random walks tries to unlock the models through a sequence of bit-flips over randomly generated *initial key.* The main objective is to guess the secret key and unlocks the baseline accuracy of the model. The second type is a genetic algorithm (GA) attack. Through

[2]Design details of Ethos's weight decoder, Exynos weight decompressor/dispatcher, and TPU weight fetcher are not publicly available. Therefore, only the design principle is described and not the detailed implementation.

the support of a GA-based attack, the attacker will try to predict the secret key. An initial population of keys will be evaluated through a set of inference calls. Later, the best ones are selected and passed through a crossover and mutation process, generating new attacker samples. This iterative process is repeated until the correct key is found or the maximum iteration is reached. Both attacks are evaluated over 2 million inference trials. Lastly, a security enhancement analysis is provided with the support of pruning. Pruned models keep only the *relevant* weights, and the remaining ones are set to zero. By removing the redundant weights, pruning should significantly impact the proposed obfuscation technique performance. The brute-force and genetic algorithm attacks are performed over the pruned and obfuscated models.

The following sections will detail the models, dataset, the experimental setup, and the results of each experimental category.

### 7.3.1 Models and Dataset

To evaluate our proposed model obfuscation technique, we selected three well-known CNN models with different numbers of layers, filters per layer, and filter sizes (Table 7.1). The models were pre-trained on the ImageNet ILSVRC-2012 dataset [11] for the image classification task.

Table 7.1: Summary of the three DNN models that compose the benchmark set. Number of weights is displayed in mega (M) unit.

| **Model** | **Number of Convolutional Layers** | **Filter Sizes** | **Number of Parameters** | **Top-1 Accuracy** |
|---|---|---|---|---|
| AlexNet | 5 | [11x11], [5x5], [3x3] | 61M | 56.52% |
| ResNet18 | 17 | [3x3], [7x7] | 11.7M | 69.76% |
| ResNet50 | 49 | [1x1], [3x3], [7x7] | 25.5M | 76.13% |

### 7.3.2 Experimental Setup

The experiments were performed with the support of the PyTorch framework version 1.6 on a workstation equipped with dual Intel(R) Xeon(R) Gold 6246 CPUs @ 3.30GHz, 256GB RAM, and two NVIDIA Quadro RTX 8000 GPUs.

To simulate the proposed swap-based obfuscation mechanism, random secret keys and mapping control files holding how filters, rows, or columns should be swapped must be generated. This step is tightly connected to the model under evaluation since the random mapping control file generator needs prior knowledge about the model structure, such as the number of layers, filters per layer, and filter sizes. After loading the pre-trained model, the structures are swapped based on the mapping control file and the secret key. An attacker in possession of the model's parameters and structure would be able to run inferences passes but with no indication that the model has been obfuscated.

### 7.3.3 Performance Analysis

The first experiment analyzes the performance of each structure swap (filter, row, and column swap) and their impact on the model accuracy when executed by untrusted parties. Three obfuscated model accuracies are compared to the original unlocked model (Baseline) with different secret key sizes: 32, 64, and 128 bits.

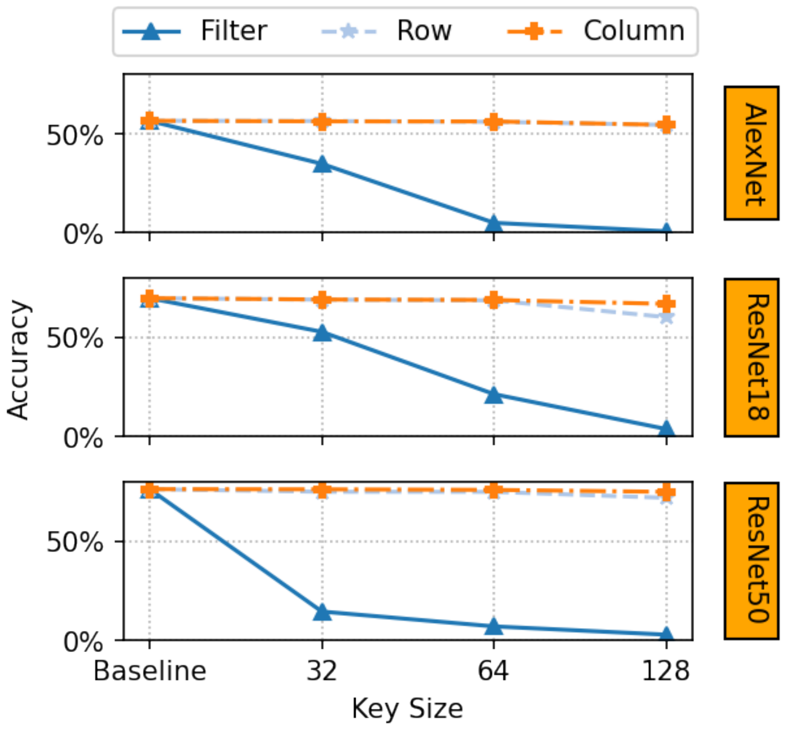


Figure 7.5: Effect of the key sizes for AlexNet, ResNet18 and ResNet50 top 1 accuracy under the three proposed model obfuscation techniques.

Figure 7.5 shows that filter swaps have a significant impact on the model's overall accuracy, providing the strongest protection. When considering only 32 bits for filter swaps, the accuracy drops by 21.68%, 16.96%, and 61.64% for AlexNet, ResNet18, and ResNet50 models. As for row and column swaps, they provide negligible protection, with an impact of $\leq 2\%$ on the model's accuracy for up to 64-bit key sizes. Moving up to a 128-bit key size, rows, and columns swap provides slightly enhanced protection with a 10% drop in the overall accuracy.

This behavior is expected since in a larger group of weights (e.g., filters), there is a higher probability of swapping relevant weights with non-relevant ones [51]. Therefore, swapping over the filter level is the best way to achieve high standard obfuscation goals.

### 7.3.4 Information Leakage Analysis

By solely considering accuracy penalties as a metric for evaluating the obfuscation technique, the defender could go straight and pick filter swap as the best option. As seen from the performance evaluation results, swapping many filters drastically impacts the model's accuracy, resulting in a high level of protection. However, this level of protection comes at the cost of *considerable perturbation* in the output class distribution. This is the side effect of uncontrollable swaps between key structures that holds many relevant weights. By analyzing the outputs of sequential inferences runs, a potential attacker can distinguish between an obfuscated and non-obfuscated model.

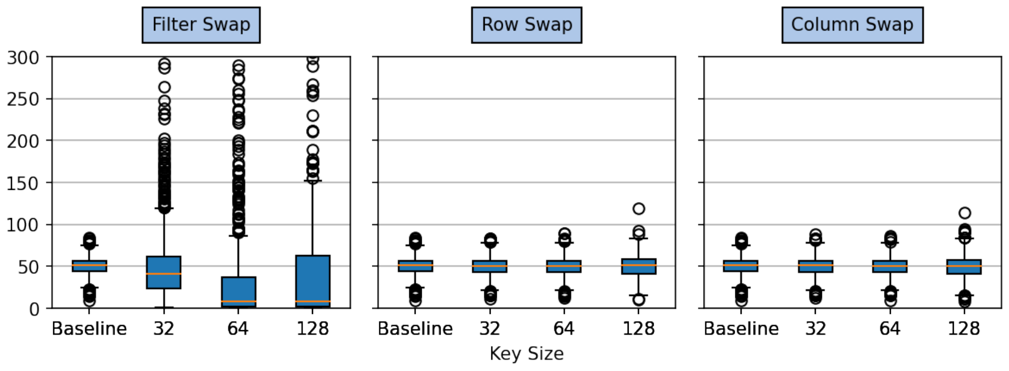


Figure 7.6: Impact of model obfuscation techniques over AlexNet output class distribution.

It is hard to quantify perturbation over the output class distribution without a proper metric, as can be seen in Figure 7.6. Therefore, a new metric ($pr$) is proposed in this work. The perturbation is described by Equation 7.1 which considers an optimal distribution ($optimal\_count_i$) for each class in the dataset and a bad distribution ($bad\_count_i$) that arises from an obfuscated model for all available classes in the dataset ($N$). A high $pr$ means the proposed obfuscation scheme incurs a high perturbation over the final distribution.

$$[!h]pr = \frac{\sqrt{\sum_{i=0}^{N}(optimal\_count_i - bad\_count_i)^2}}{total\_count} \tag{7.1}$$

Figure 7.6 and Table 7.2 show the output class distribution of AlexNet during the previous model performance experiment in Section 7.3.3. Filter swap obfuscation clearly incurs high perturbation rates, even for the smallest secret key size ($pr >$

Table 7.2: Perturbation rate on models output distribution using different types of obfuscation mode and key size.

| Swap Location | Perturbation Rate | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | AlexNet | | | ResNet18 | | | ResNet50 | | |
| | 32-bit key | 64-bit key | 128-bit key | 32-bit key | 64-bit key | 128-bit key | 32-bit key | 64-bit key | 128-bit key |
| Filter | 2.43% | 22.28% | 31% | 1.92% | 15.1% | 19.3% | 52.96% | 70.58% | 34.97% |
| Row | 0.16% | 0.27% | 0.43% | 0.24% | 0.29% | 1.67% | 0.24% | 0.25% | 0.53% |
| Column | 0.19% | 0.21% | 0.44% | 0.23% | 0.26% | 0.49% | 0.08% | 0.17% | 0.28% |

22%). Row and column swap obfuscation preserve the distribution ($pr < 1.7\%$), with a marginal *pr* increase due to few outliers.

The experiment results have shown that filter swap obfuscation delivers a high level of protection by drastically dropping the accuracy but harms the output class distribution and that row/column swaps provide marginal protection while keeping the class distribution stable. Is there a way to balance these two strategies and gets the best of both worlds? A fourth structure swap is proposed. The hybrid obfuscation scheme merges filter and row/column swaps, where few filters are swapped, and the remaining key bits are reserved for rows or columns swap.

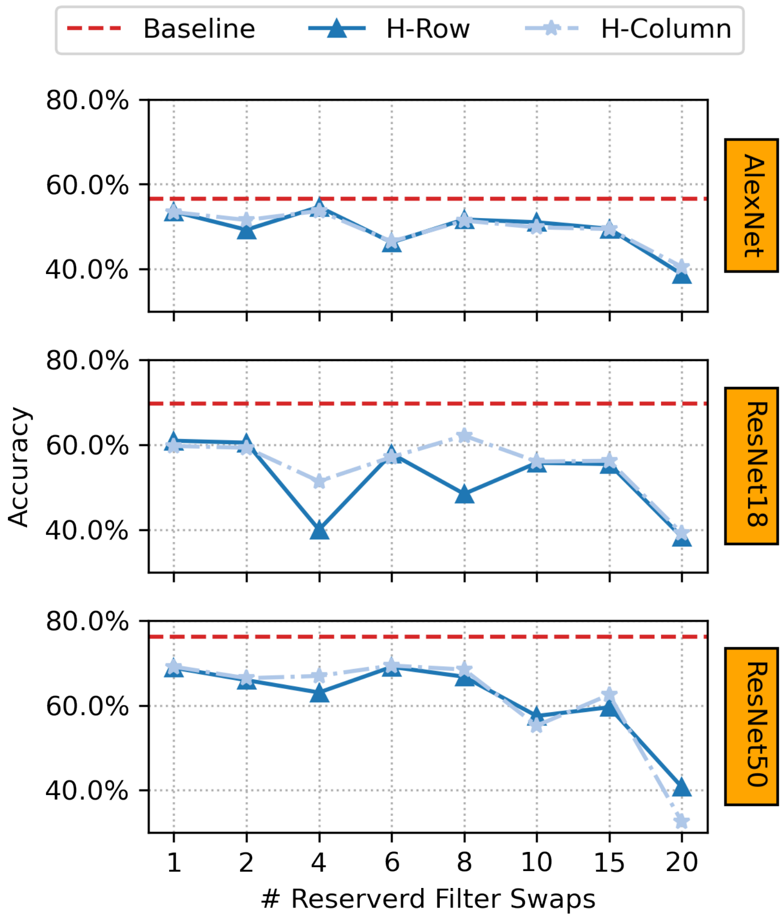


Figure 7.7: Impact on models accuracy using different types of configurations for hybrid row (H-Row) and hybrid column (H-Column) modes using 128-bit key.

Table 7.3: Perturbation rate on models output distribution using different types of configurations for hybrid row and hybrid column modes using 128-bit key.

| # Reserved Filter Swaps | Perturbation Rate 128-bit key | | | | | |
|---|---|---|---|---|---|---|
| | AlexNet | | ResNet18 | | ResNet50 | |
| | Hybrid Row | Hybrid Column | Hybrid Row | Hybrid Column | Hybrid Row | Hybrid Column |
| 1 | 0.56% | 0.59% | 1.21% | 1.47% | 0.76% | 0.77% |
| 2 | 0.94% | 0.71% | 1.26% | 1.51% | 1.10% | 1.01% |
| 4 | 0.42% | 0.51% | 10.67% | 3.46% | 1.47% | 1.06% |
| 6 | 1.30% | 1.31% | 1.29% | 1.34% | 0.69% | 0.68% |
| 8 | 0.69% | 0.70% | 4.24% | 0.79% | 1.10% | 1.90% |
| 10 | 0.84% | 0.90% | 1.52% | 1.55% | 6.78% | 7.45% |
| 15 | 0.87% | 0.90% | 2.15% | 2.27% | 2.46% | 1.92% |
| 20 | **1.95%** | **1.79%** | **6.1%** | **5.58%** | **18.48%** | **26.41%** |

Figure 7.7 and Table 7.3 show the impact on the accuracy and output class distribution of different types of hybrid swap configurations for the three models. By reserving 20 bits of the secret key for filter swap, the accuracy penalties are substantial, and the distribution perturbation ($pr$) achieves an acceptable rate. For example, Hybrid Row over AlexNet and ResNet18 with 128-bit key reaches 17.73% and 31.47% of accuracy penalties while keeping $pr \leq 7\%$. As a baseline for comparison, the key-dependent backpropagation algorithm [7] illustrated in Figure 7.1 incurs a $pr$ of 18% over a small four-layers deep network (LeNet-4) evaluated with a ten-classes dataset (MNIST).

Table 7.4: Summary of the obfuscation results over the DCNN models that compose the benchmark set [8–10]. Accuracy is measured based on top-1 error on ImageNet [11] dataset. The results consider an 128-bit key for filters, rows, and columns swaps. The hybrid mode is composed of 20 filters and 108 rows/columns swaps.

| Model | Original Accuracy | Obfuscated Model 128-bit key | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Filter | | Row | | Column | | Hybrid Row | | Hybrid Column | |
| | | accuracy | % drop | accuracy | % drop | accuracy | % drop | accuracy | % drop | accuracy | % drop |
| AlexNet | 56.52% | 0.85% | **55.67%** | 54.45% | **2.07%** | 54.40% | **2.12%** | 38.80% | **17.73%** | 40.54% | **15.98%** |
| ResNet18 | 69.76% | 3.84% | **65.92%** | 60.21% | **9.55%** | 66.88% | **2.88%** | 38.29% | **31.47%** | 39.14% | **30.64%** |
| ResNet50 | 76.13% | 2.85% | **73.28%** | 71.75% | **4.38%** | 74.77% | **1.36%** | 40.75% | **35.38%** | 32.56% | **43.57%** |

Table 7.4 shows the summary of the swap-base obfuscation results over the three DNN models under the four proposed structure swap configurations. The hybrid row and column approaches consider a reserve of 20 bits from the secret key for filter swaps. With a 128-bit key on AlexNet, ResNet18, and ResNet50, the hybrid obfuscation can preserve the output distribution (up to 18.48% considering the Hybrid Row scheme) and reduce the overall accuracy at acceptable rates (more than 35.38% in best cases), providing the best trade-off between accuracy penalties and output class perturbation.

Experiments have shown that increasing the filter swap reserved bit keys by more than twenty increases output class distribution perturbation. This fact occurs mainly because the first convolutional layer is more sensitive and less redundant [3], and a simple additional filter swap over the first layer impacts the model's overall accuracy.

The perturbation metric coupled with the accuracy penalties can help Model Providers to achieve an optimal (or close to optimal) obfuscation scheme. Since the obfuscation process comprises random selections of filters, rows, and columns, proper use of an optimization process can deliver even better results than the ones presented in Table 7.4.

### 7.3.5 Security Analysis

A significant challenge for any obfuscation technique is to provide a high level of security, keeping its model misbehaving under unauthorized access of unknown entities. The swap-based obfuscation allows any entity access to the model structure, parameters, and mapping control list. Even with such an amount of information, a potential attacker would not be able to achieve the same accuracy as trusted parties since it does possess the correct secret key. To analyze the security properties of the proposed approach, two types of attacks on the obfuscated models are simulated in an effort to discover the correct key.

#### Brute-Force Attack

The first type of attack is a brute-force mode with random walks. The target model is obfuscated with the swap-base scheme using a 128-bit key and the filter and the hybrid structure approach to simulate the brute-force type of attack. Then, an *initial key* is randomly generated, resulting in a starting-point accuracy. Considering the starting-point scenario, the attacker performs random walks over individual bits of the initial key, flipping one at a time and checking the new accuracy with a known test set. If the accuracy improves, the attacker keeps that bit and considers it a hit by removing the index from the random search. If the accuracy drops, the bit is reverted to its original value, and the random walks are performed over the remaining indices. This iterative process is repeated until the attack reaches the target accuracy (i.e., baseline) or after the expiry of some number of iterations (1 000 in this case). For the obfuscation to be considered secure, the technique must not allow the attack to reach the target accuracy by random walking over the secret key. Moreover, a secure obfuscation technique must keep the accuracy far below the target by considerable margins.

Figure 7.8 shows the success potential of an attacker with brute-force mode over a 128-bit key for each network during 1 000 trials. Each trial corresponds to a random walk step, where one of the key's bit is flipped. During each trial, a **full prediction**

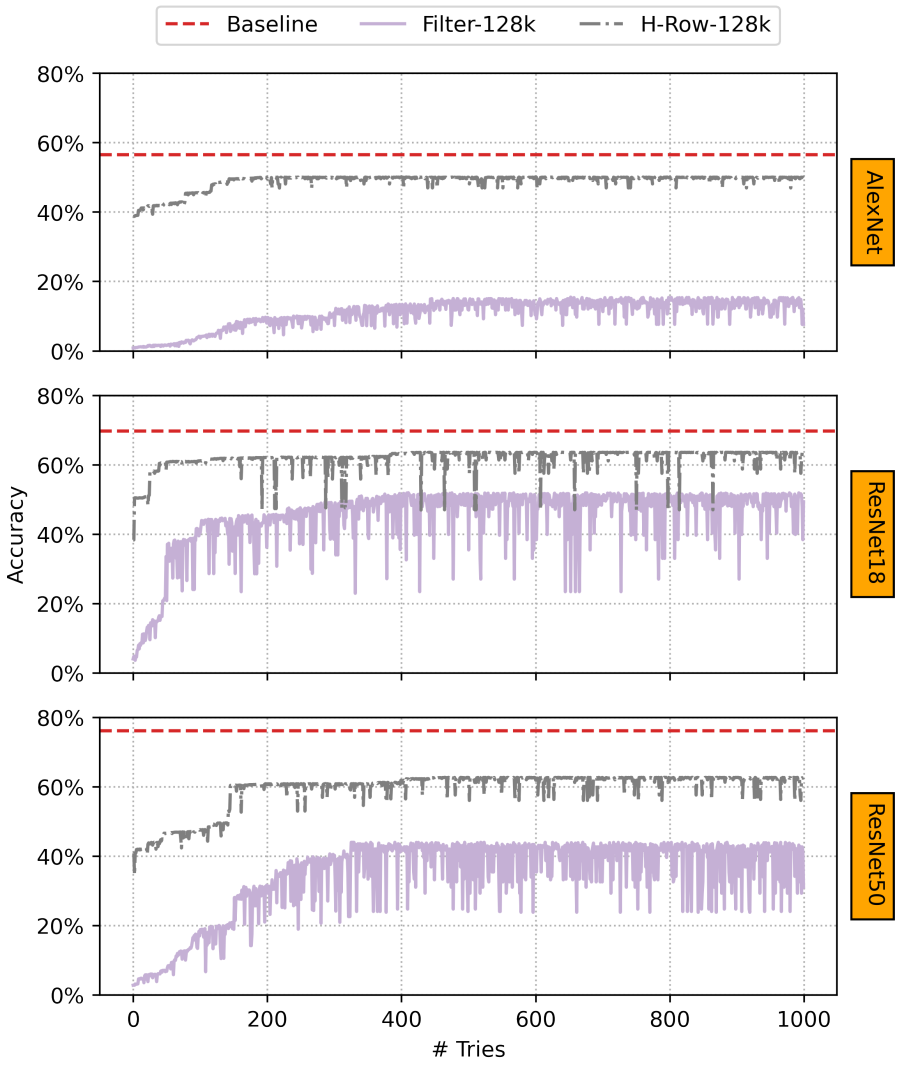


Figure 7.8: Security analysis of the swap-based obfuscation technique using filter (Filter-128k) and hybrid row (H-Row-128k) modes with 128-bit key against brute-force attack.

**over the Imagenet validation set is performed**. For the three models, the swap-based obfuscation approach can sustain a substantial accuracy penalty with the filters swap approach. Considering 20 bits for filter swaps out of 128, the hybrid row approach maintains the accuracy penalties up to 8.8% for ResNet18.

The attack achieves low accuracies at the initial random walks due to the randomness of the secret key starting point. The accuracy rises to a plateau as the attack moves forward over the trials. After reaching the plateau, the attack accuracies of the remaining trials get stagnant, not exceeding a specific threshold. The non-uniqueness nature of the swaps causes this effect over the brute-force attacks with random walks. Structures, such as filters, are first swapped considering all the available filters/rows/columns within a specific layer. Then, the upcoming swaps have the remaining structures in the swap pool and the structures from the previous swaps, creating a relation between the bit keys. This correlation allows model providers to determine a specific order in which the bit keys should be flipped to achieve the correct placement of the structures.

### Genetic Algorithm Attack

The second experiment is a genetic algorithm (GA) based type of attack. This attack consists of applying GA, where the initial genomes population is composed of possible secret keys to unlock the obfuscated model (Figure 7.9). Each genome is evaluated through a fitness function that consists of **a complete set of predictions over the whole validation set**. Then, the genomes are ranked based on their accuracy (i.e., fitness function output), which expresses how well and close that genome got to unlocking the obfuscated model. This process, entitled generation, is repeated but now considers only the most relevant genomes of the rank. A crossover function is applied over this new subset by randomly selecting pairs of genomes and combining their genetic information (secret key bits). The combination process creates new pairs of genomes denominated offsprings. Then, offsprings are conducted to a mutation process, where single or multiple bits from their secret key can be flipped. Finally, the generation process can restart with a new genome population (i.e., secret key candidates).

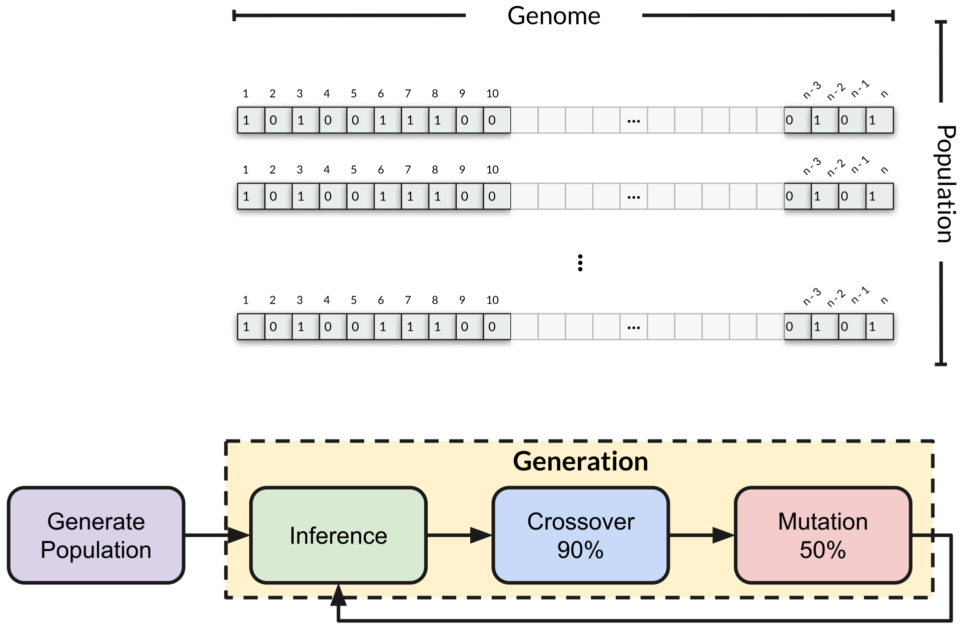


Figure 7.9: Steps performed during the Genetic Algorithm attack.

GA is a highly parametrizable algorithm requiring several combinations and experimental runs to converge appropriately. Population size, number of generations, crossover function, crossover, and mutation rates are the parameters that must be fine-tuned to achieve a proper behavior of GA. Moreover, these parameters are tightly coupled to the problem. In the GA-based attack, fine-tuning the parameters is not different from the original algorithm. It requires several combinations and experimental runs until a proper convergence. Yet, the GA-based attack fitness function comprises a thousand DNN model inferences, demanding a humongous amount of time and computational resources. Therefore, searching for an optimal set of parameters is out of the scope of this work. During the GA-base attack experiments, the parameters are set with the following values:

- Genome size: 128
- Population size: 10
- Number of Generations: 200
- Crossover function: Single-point Crossover
- Crossover rate: 0.9
- Mutation rate: 0.5

Figure 7.10 shows the success potential of the GA-based attack on a 128-bit key for each obfuscated model over 200 generations. Each point in the plot corresponds to the best top-1 genome accuracy of a generation. Similar to the brute-force attack with random walks, the three obfuscated models can preserve a substantial accuracy penalty when employing the swap-based obfuscation over the filter structure. For the hybrid-row approach with 20 bits reserved for filter swaps, a slight margin of 4% accuracy penalty is sustained over AlexNet after 100 million inference trials [3].

It is essential to highlight that both attacks (brute-force and GA-based) have considered predictions on the whole validation dataset for simplicity of coding. In a real-world scenario, a potential attacker **would not** have this amount of data available to evaluate each candidate key. Instead, the attacker would have to make several predictions using an reduced portion of the validation set ($< 10\%$). Thus, the security experiments consider unusual cases where attackers would possess the same amount of validation data as the model provider. Finally, the security experiments demonstrate that *seeking monotonic accuracy improvement via random walk or GA-based algorithms will not disclose the secret key* when obfuscating the DNN models with the proposed approach.

[3]# trials = 200 generations $\times$ 10 genomes $\times$ 50,000 images

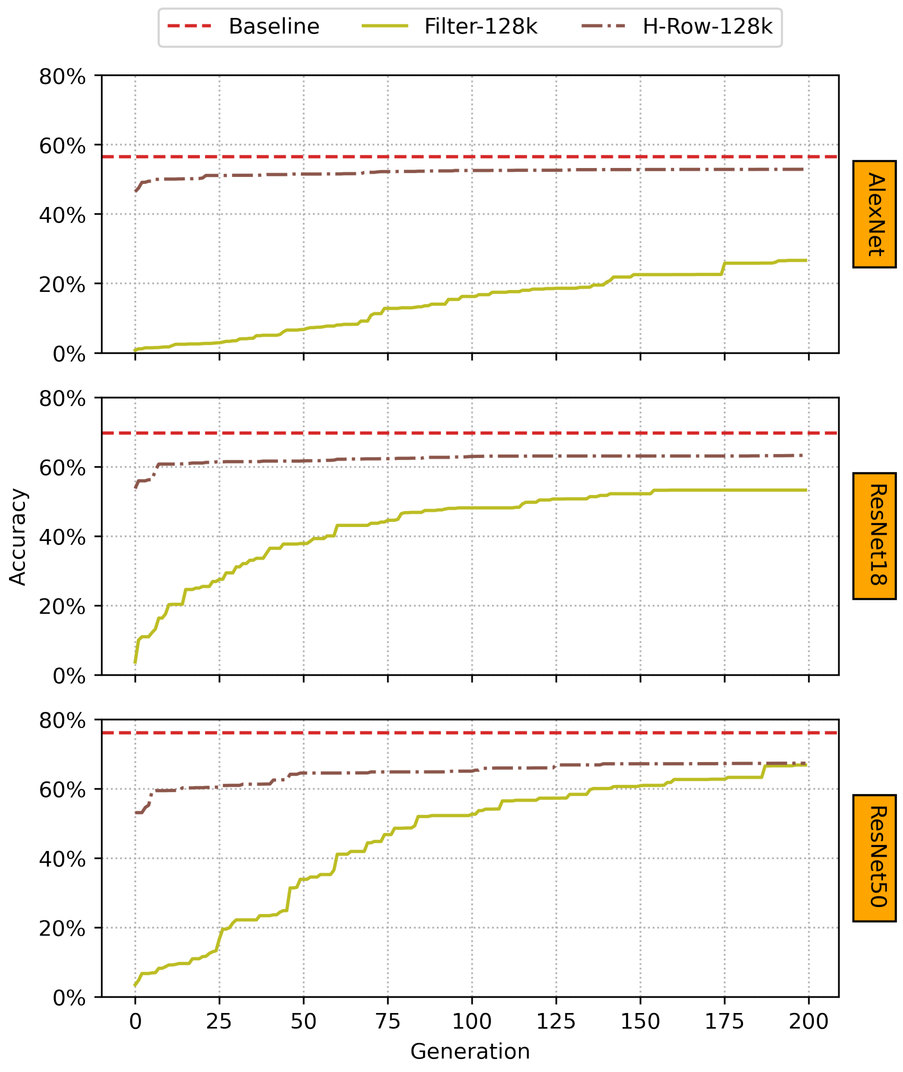


Figure 7.10: Security analysis of the swap-based obfuscation technique using filter (Filter-128k) and hybrid row (H-Row-128k) modes with 128-bit key against GA-based attack.

### 7.3.6 Enhancing Security of Swap-based Obfuscation

The security of the swap-based obfuscation technique can be further enhanced by considering pruned models instead of dense ones. DNN models have natural redundancy, and a considerable portion of its parameters have little or no relevance over the final accuracy [51]. Pruning [3, 52] is a compression technique that iteratively removes the redundant parameters of the models based on their impact on the classification accuracy. It keeps only the relevant weights, making the model extremally sparse (more details in Section 2.1.5). Since the swap-based obfuscation heavily relies on the weights structure, pruning should significantly impact the proposed technique's performance against brute-force and GA-based attacks.

The security experiments performed in Section 7.3.5 with the brute-force attack were re-executed but now considering the pruned version of the obfuscated models

Table 7.5: DNNs top-1 accuracy under model compression. Pruning (P) removes more than 59% of the parameters by dropping less than 1% on accuracy.

| Model | Model Type | Data Type | Pruning Rate | Top-1 acc. |
|---|---|---|---|---|
| AlexNet | Original (D) | fp32 | - | 56.52% |
| | Pruned (P) | fp32 | 88.31% | 56.61% |
| ResNet18 | Original | fp32 | - | 69.76% |
| | Pruned | fp32 | 59.92% | 69.87% |
| ResNet50 | Original | fp32 | - | 76.13% |
| | Pruned | fp32 | 84.57% | 75.52% |

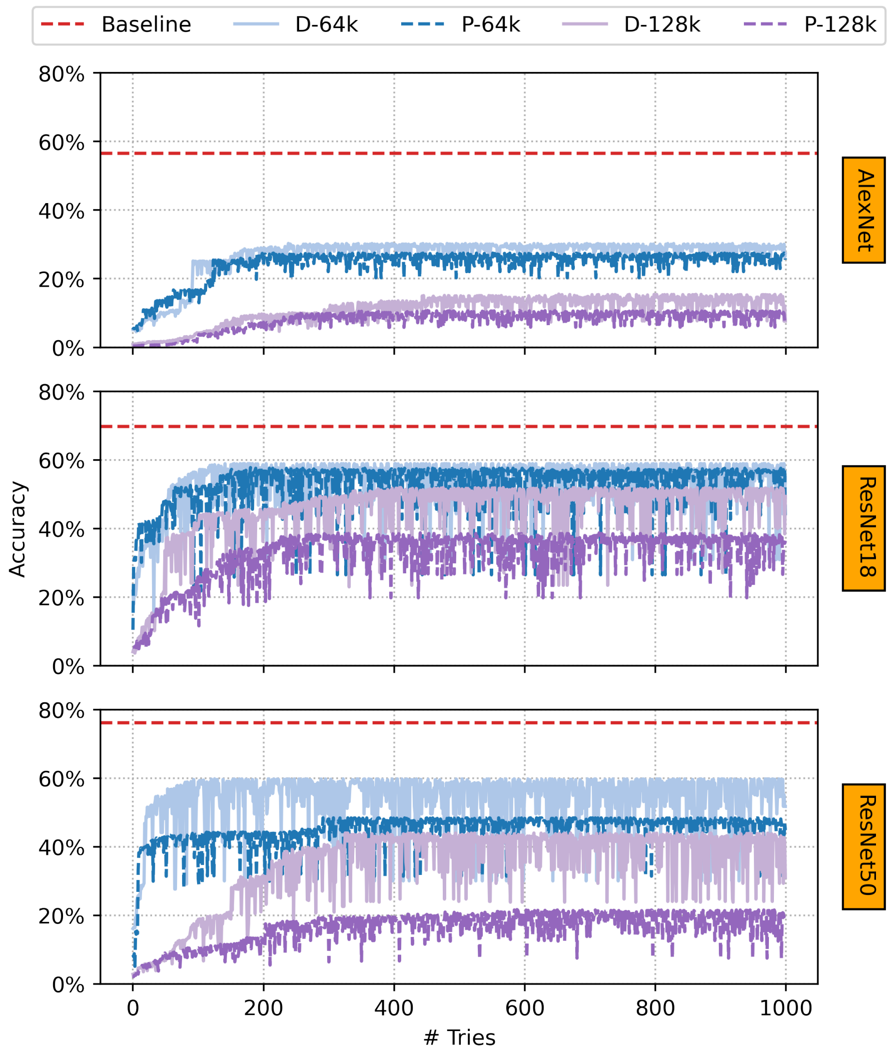


Figure 7.11: Security analysis of the swap-based obfuscation scheme over Dense (D) and Pruned (P) models of AlexNet, ResNet18, and ResNet50 using 64-bit and 128-bit keys against brute-force attack.

in Table 7.5. Figure 7.11 shows the secure potential of 64-bit and 128-bit keys filters swaps over pruned and dense models. Pruned models outperform the dense version, especially with a 128-bit key size over ResNet18 and ResNet50. This set of experiments indicates that pruning the model before obfuscating it can be a prominent way to enhance swap-based obfuscation security.

## 7.4 Concluding Remarks

Training state-of-the-art deep neural networks is an expensive endeavor. It requires a massive amount of training data that is further hand-labeled, good expertise in the domain, and a considerable number of high-performance computation resources to train a model. Due to these factors, DNN models are considered and treated as valuable storehouses of intellectual properties (IP) and must be protected against attacks.

Model piracy attacks have become a significant concern over the past years where attackers are capable of stealing (or cloning) the model for illegal usage or distribution. DNN models are easy targets without proper protection, especially if the attacker has physical access to the device where the models have been deployed. By stealing the model's architecture and its hyperparameters, attackers can achieve a near-perfect clone of the original model. Therefore, models must be shielded before being delivered to end-users.

Neural model obfuscation protects the original model against structure and parameter piracy. It adds a security layer that locks the model such that only authorized parties can achieve the expected behavior. Non-authorized users are unable to *unlock* the model, resulting in significant penalties over its accuracy, making the model useless for illegal use or distribution.

The proposed swap-based obfuscation encompasses a novel, lightweight, scalable technique that *does not require a specialized training process*. It conceals the original model's internal structure by swapping rows, columns, and full filters based on a secret key and a mapping control file. The proposed swap-based obfuscation effectiveness is demonstrated across different DNN architectures, with varying numbers of layers, filters per layer, and filter sizes. Four different structure swapping schemes are proposed and evaluated, showing that a hybrid approach can provide a stealth obfuscation by avoiding information leakage by preserving the output class distribution. The filter option provides a full-protection scheme by applying considerable penalties over the model's accuracy. Additionally, the proposed obfuscation scheme shows strong resistance against security attacks such as brute-force and GA-based ones.

# Chapter 8

# Conclusion

Reliability and security solutions applied to computer systems have been widely explored and still play a critical role, particularly in the artificial intelligence era. The transistors shrinking to atomic limits and the adoption of machine learning-based solutions over safety-critical systems have brought the spotlight back to these research areas. Notably, this thesis explores solutions to enhance the reliability and security of deep neural network-based microarchitectures.

Compression techniques, such as pruning and quantization, are promising solutions for deploying deep neural networks over resource-constrained devices. However, it is unknown how such techniques can impact the reliability of DNN-based systems. This thesis evaluates pruning and quantization over a large set of real-world DNN models, showing that compression can improve the system's reliability by reducing the error propagation over the models. Furthermore, when combining pruning and quantization solutions, the overall reliability is enhanced by 108.7x.

Safety-critical systems demand high dependability standards. To be considered safe for end-users, DNN-based automotive systems must provide solutions that can detect more than 99% of the faults in any component. This thesis proposes a robust, low-cost AN-based solution to ensure the reliable execution of DNN-based accelerators. The AN-based detection scheme achieves a minimum fault coverage of 99.6% for up to $10^{-3}$ bit error rate (BER) scenario. Moreover, the proposed detection scheme, coupled with the *Word Masking* technique, can preserve the correct execution without compromising the original model accuracy even at unusually high bit error rates such as $10^{-2}$ BER.

The proposed AN-based detection scheme has limitations that can jeopardize its deployment in real-world applications. This thesis provides a solution that makes the AN error-detection capability incur practically no overhead over the system by a software and hardware co-design method. The AN code-aware quantization technique removes the pre-encoding step of the previous AN detection scheme by constraining to integers multiples of $A$ only. The AN code-aware quantization can achieve near-optimal accuracies with $\leq 1\%$ of difference from the original model.

Because training DNN models requires a humungous effort with high costs, they are considered and treated as valuable storehouses of intellectual properties (IP). Therefore, the models must be protected against attacks, such as piracy. Without proper protection, non-authorized users can steal the DNN model for illegal use or distribution, resulting in considerable losses for model providers (i.e., ML vendors). This thesis proposes a lightweight, scalable, and robust technique to protect DNN models from man-at-the-end (MATE) attack types. By obfuscating the model's parameters, the swap-based solution can thwart the illegal use of the model by significantly decreasing the original model's accuracy without leaving traces that the model has been obfuscated.

Steadily, safety-critical systems incorporate DNNs to perform additional tasks or even entirely replace the need for human interaction, increasing the degree of safety, integrity, and security demanded. In the future, the AN-based detection solution can be enhanced by quantizing all the inputs of the networks with the AN code-aware quantization, making every data that flows into the model easy to be checked. In the context of security, the swap-based obfuscation can be further enhanced through an optimization-based technique that will select which structures give the best benefits in terms of protection and information leakage costs.

## 8.1 Contribution

This thesis proposes novel techniques and improvements over the state-of-the-art related to model obfuscation and error detection on deep neural networks microarchitectures. Additionally, a custom open-source DNN fault injection framework was created and shared through a GitHub repository during the research development. The main contributions are listed below:

- Extensive reliability analysis of compressed DNN models under the presence of transient faults. It is demonstrated that compression, in the mean of data quantization and model pruning, can dramatically increase the system's overall resiliency under a faulty condition (e.g., cosmic radiation).

- TorchFI, a custom DNN fault injection framework created on top of the classic and highly adopted DNN framework PyTorch. TorchFI can simulate transient, permanent, and intermittent faults over computational units and memory subsystems.

- Custom DNN error detection based on arithmetic error codes (AN code) that can detect more than 99% of errors over DNN MAC units. When coupled with a word-masking error correction scheme, the technique allows the DNN system to operate with no accuracy loss, even under high bit error rate scenarios.

- Novel AN code-aware quantization process that enhances the custom DNN error detection by removing the necessity of pre-multiplying one of the operands (e.g., input features or model parameters) by a constant A; further reducing the cost of an HW detection scheme through a co-design scheme (HW and SW error detection cooperation).
- Novel lightweight DNN model obfuscation technique that does not require a specialized training process to protect the model's IP while providing oblivious output class distribution.

# References

[1] SZE, V., CHEN, Y.-H., YANG, T.-J., et al. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey", *Proceedings of the IEEE*, v. 105, n. 12, pp. 2295–2329, 2017. doi: 10.1109/JPROC.2017.2761740.

[2] HOROWITZ, M. "1.1 Computing's energy problem (and what we can do about it)". In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014. doi: 10.1109/ISSCC.2014.6757323.

[3] HAN, S., POOL, J., TRAN, J., et al. "Learning Both Weights and Connections for Efficient Neural Networks". In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, p. 1135–1143, Cambridge, MA, USA, 2015. MIT Press.

[4] CHEN, Y. H., EMER, J., SZE, V. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks", *Proc. - 2016 43rd Int. Symp. Comput. Archit. ISCA 2016*, pp. 367–379, 2016. ISSN: 19374143. doi: 10.1109/ISCA.2016.40.

[5] CHEN, Y.-H., OTHERS. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks", *IEEE J. Solid-State Circuits*, 2017. doi: 10.1109/JSSC.2016.2616357.

[6] YANG, J., SHEN, X., XING, J., et al. "Quantization Networks". In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7300–7308, 2019. doi: 10.1109/CVPR.2019.00748.

[7] CHAKRABORTY, A., MONDAI, A., SRIVASTAVA, A. "Hardware-Assisted Intellectual Property Protection of Deep Learning Models". In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020. doi: 10.1109/DAC18072.2020.9218651.

[8] KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E. "ImageNet Classification with Deep Convolutional Neural Networks". In: Pereira,

F., Burges, C., Bottou, L., et al. (Eds.), *Advances in Neural Information Processing Systems*, v. 25. Curran Associates, Inc., 2012. Available: <`https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf`>.

[9] SIMONYAN, K., ZISSERMAN, A. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: Bengio, Y., LeCun, Y. (Eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. Available: <`http://arxiv.org/abs/1409.1556`>.

[10] HE, K., ZHANG, X., REN, S., et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

[11] DENG, J., DONG, W., SOCHER, R., et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*, 2009.

[12] AVIZIENIS, A., LAPRIE, J.-C., RANDELL, B., et al. "Basic concepts and taxonomy of dependable and secure computing", *IEEE Transactions on Dependable and Secure Computing*, v. 1, n. 1, pp. 11–33, 2004. doi: 10.1109/TDSC.2004.2.

[13] HOCHSCHILD, P. H., TURNER, P., MOGUL, J. C., et al. "Cores That Don't Count". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, p. 9–16, New York, NY, USA, 2021. Association for Computing Machinery. ISBN: 9781450384384. doi: 10.1145/3458336.3465297. Available: <`https://doi.org/10.1145/3458336.3465297`>.

[14] DIXIT, H. D., PENDHARKAR, S., BEADON, M., et al. "Silent Data Corruptions at Scale". In: *arXiv preprint*, 2021. doi: 10.48550/arXiv.2102.11245. Available: <`https://doi.org/10.48550/arXiv.2102.11245`>.

[15] KIM, Y., DALY, R., KIM, J., et al. "Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors", *SIGARCH Comput. Archit. News*, v. 42, n. 3, pp. 361–372, jun 2014. ISSN: 0163-5964. doi: 10.1145/2678373.2665726. Available: <`https://doi.org/10.1145/2678373.2665726`>.

[16] KOCHER, P., HORN, J., FOGH, A., et al. "Spectre Attacks: Exploiting Speculative Execution". In: *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[17] LIPP, M., SCHWARZ, M., GRUSS, D., et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[18] KOREN, I., KRISHNA, C. M. *Fault-Tolerant Systems*. 1st ed. San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2007. ISBN: 0120885255.

[19] ARM. “ARM Security Technology - Building a Secure System using TrustZone Technology”. Apr 2009. Available: <`https://documentation-service.arm.com/static/5f212796500e883ab8e74531?token=`>. Visited on: 06-19-22.

[20] NILSSON, A., BIDEH, P. N., BRORSSON, J. “A Survey of Published Attacks on Intel SGX”. 2020.

[21] DOWLIN, N., GILAD-BACHRACH, R., LAINE, K., et al. “CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy”. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML’16, p. 201–210. JMLR.org, 2016.

[22] LIU, B., DING, M., SHAHAM, S., et al. “When Machine Learning Meets Privacy: A Survey and Outlook”, v. 54, n. 2, mar 2021. ISSN: 0360-0300. doi: 10.1145/3436755. Available: <`https://doi.org/10.1145/3436755`>.

[23] XU, H., SU, Y., ZHAO, Z., et al. “DeepObfuscation: Securing the Structure of Convolutional Neural Networks via Knowledge Distillation”. 2018.

[24] ISAKOV, M., BU, L., CHENG, H., et al. “Preventing Neural Network Model Exfiltration in Machine Learning Hardware Accelerators”. In: *2018 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 62–67, 2018. doi: 10.1109/AsianHOST.2018.8607161.

[25] ALAM, M., SAHA, S., MUKHOPADHYAY, D., et al. “<i>NN-Lock</i>: A Lightweight Authorization to Prevent IP Threats of Deep Learning Models”, *J. Emerg. Technol. Comput. Syst.*, dec 2021. ISSN: 1550-4832. doi: 10.1145/3505634. Available: <`https://doi.org/10.1145/3505634`>. Just Accepted.

[26] WELLS, P. M., CHAKRABORTY, K., SOHI, G. S. “Adapting to Intermittent Faults in Future Multicore Systems”. In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pp. 431–431, 2007. doi: 10.1109/PACT.2007.4336259.

[27] RASHID, L., PATTABIRAMAN, K., GOPALAKRISHNAN, S. "Characterizing the Impact of Intermittent Hardware Faults on Programs", *IEEE Transactions on Reliability*, v. 64, n. 1, pp. 297–310, 2015. doi: 10.1109/TR.2014.2363152.

[28] CONSTANTINESCU, C. "Dependability benchmarking using environmental test tools". In: *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pp. 567–571, 2005. doi: 10.1109/RAMS.2005.1408423.

[29] CONSTANTINESCU, C. "Intermittent faults and effects on reliability of integrated circuits". In: *2008 Annual Reliability and Maintainability Symposium*, pp. 370–374, 2008. doi: 10.1109/RAMS.2008.4925824.

[30] BORKAR, S. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation", *IEEE Micro*, v. 25, n. 6, pp. 10–16, 2005. doi: 10.1109/MM.2005.110.

[31] GOLDSTEIN, B. F., FERREIRA, V. C., SRINIVASAN, S., et al. "A Lightweight Error-Resiliency Mechanism for Deep Neural Networks". In: *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pp. 311–316, 2021. doi: 10.1109/ISQED51717.2021.9424287.

[32] PAN, A., TSCHANZ, J. W., KUNDU, S. "A Low Cost Scheme for Reducing Silent Data Corruption in Large Arithmetic Circuits". In: *2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, pp. 343–351, 2008. doi: 10.1109/DFT.2008.42.

[33] MCCULLOCH, W. S., PITTS, W. "A Logical Calculus of Ideas Immanent in Nervous Activity", *Bulletin of Mathematical Biophysics*, v. 5, 1943.

[34] HEBB, D. O. *The organization of behavior: A neuropsychological theory.* New York, Wiley, jun. 1949. ISBN: 0-8058-4300-0.

[35] ROSENBLATT, F. *Principles of Neurodynamics.* Arlington, VA, USA, Spartan Books, 1959.

[36] MINSKY, M., PAPERT, S. *Perceptrons: An Introduction to Computational Geometry.* Cambridge, MA, USA, MIT Press, 1969.

[37] VEEN, F. V., LEIJNEN, S. "The Neural Network Zoo". Available: <`https://www.asimovinstitute.org/neural-network-zoo`>.

[38] LEIJNEN, S., VEEN, F. V. "The Neural Network Zoo", *Proceedings*, v. 47, n. 1, 2020. ISSN: 2504-3900. doi: 10.3390/proceedings2020047009. Available: <`https://www.mdpi.com/2504-3900/47/1/9`>.

[39] LECUN, Y., BOSER, B., DENKER, J. S., et al. "Backpropagation Applied to Handwritten Zip Code Recognition", *Neural Computation*, v. 1, n. 4, pp. 541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

[40] CUN, Y. L., BOSER, B., DENKER, J. S., et al. "Handwritten Digit Recognition with a Back-Propagation Network". In: *Advances in Neural Information Processing Systems 2*, p. 396–404, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 1990. ISBN: 1558601007.

[41] SZEGEDY, C., LIU, W., JIA, Y., et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015. doi: 10.1109/CVPR.2015.7298594.

[42] HORNIK, K., STINCHCOMBE, M., WHITE, H. "Multilayer feedforward networks are universal approximators", *Neural Networks*, v. 2, n. 5, pp. 359–366, 1989. ISSN: 0893-6080. doi: https://doi.org/10.1016/0893-6080(89)90020-8. Available: <https://www.sciencedirect.com/science/article/pii/0893608089900208>.

[43] CORPORATION, I. *BFLOAT16 - Hardware Numerics Definition.* Relatório técnico, Intel Corporation, 2018. Available: <https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf>. Visited on: 04-09-22.

[44] JOUPPI, N. P., YOUNG, C., PATIL, N., et al. "In-datacenter performance analysis of a Tensor Processing Unit", abr. 2017.

[45] JACOB, B., KLIGYS, S., CHEN, B., et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[46] JAIN, S., GURAL, A., WU, M., et al. "Trained Quantization Thresholds for Accurate and Efficient Fixed-Point Inference of Deep Neural Networks". In: Dhillon, I., Papailiopoulos, D., Sze, V. (Eds.), *Proceedings of Machine Learning and Systems*, v. 2, pp. 112–128, 2020. Available: <https://proceedings.mlsys.org/paper/2020/file/e2c420d928d4bf8ce0ff2ec19b371514-Paper.pdf>.

[47] ESSER, S. K., MCKINSTRY, J. L., BABLANI, D., et al. "Learned Step Size Quantization". In: *International Conference on Learning Representations*, 2020. Available: <https://openreview.net/forum?id=rkgO66VKDS>.

[48] BHALGAT, Y., LEE, J., NAGEL, M., et al. "LSQ+: Improving low-bit quantization through learnable offsets and better initialization". In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 2978–2985, 2020. doi: 10.1109/CVPRW50498.2020.00356.

[49] KIM, D., LEE, J., HAM, B. "Distance-aware Quantization". In: *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 5251–5260, 2021. doi: 10.1109/ICCV48922.2021.00522.

[50] GONG, R., LIU, X., JIANG, S., et al. "Differentiable Soft Quantization: Bridging Full-Precision and Low-Bit Neural Networks". In: *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 4851–4860, Los Alamitos, CA, USA, nov 2019. IEEE Computer Society. doi: 10.1109/ICCV.2019.00495. Available: <`https://doi.ieeecomputersociety.org/10.1109/ICCV.2019.00495`>.

[51] CUN, Y. L., DENKER, J. S., SOLLA, S. A. "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems*, pp. 598–605. Morgan Kaufmann, 1990.

[52] HAN, S., MAO, H., DALLY, W. J. "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding". In: Bengio, Y., LeCun, Y. (Eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. Available: <`http://arxiv.org/abs/1510.00149`>.

[53] ZHU, M., GUPTA, S. "To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression". In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*, 2018.

[54] MUKHERJEE, S. *Architecture Design for Soft Errors.* San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2008. ISBN: 9780080558325.

[55] CONSTANTINESCU, C. "Intermittent faults and effects on reliability of integrated circuits". In: *2008 Annual Reliability and Maintainability Symposium*, 2008.

[56] CONSTANTINESCU, C. "Dependability benchmarking using environmental test tools". In: *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, 2005.

[57] MAY, T., WOODS, M. “Alpha-particle-induced soft errors in dynamic memories”, *IEEE Transactions on Electron Devices*, v. 26, n. 1, pp. 2–9, 1979. doi: 10.1109/T-ED.1979.19370.

[58] MULLER, D., LEBEDEV, P. “The Universe is Hostile to Computers”. Available: <`https://www.youtube.com/watch?v=AaZ_RSt0KP8`>.

[59] ZIEGLER, J. F., LANFORD, W. A. “Effect of Cosmic Rays on Computer Memories”, *Science*, v. 206, n. 4420, pp. 776–788, 1979. doi: 10.1126/science.206.4420.776. Available: <`https://www.science.org/doi/abs/10.1126/science.206.4420.776`>.

[60] QUINN, H., GRAHAM, P. “Terrestrial-based radiation upsets: a cautionary tale”. In: *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’05)*, pp. 193–202, 2005. doi: 10.1109/FCCM.2005.61.

[61] BINDER, D., SMITH, E. C., HOLMAN, A. B. “Satellite Anomalies from Galactic Cosmic Rays”, *IEEE Transactions on Nuclear Science*, v. 22, n. 6, pp. 2675–2680, 1975. doi: 10.1109/TNS.1975.4328188.

[62] ZIEGLER, J. F., CURTIS, H. W., MUHLFELD, H. P., et al. “IBM experiments in soft fails in computer electronics (1978–1994)”, *IBM Journal of Research and Development*, v. 40, n. 1, pp. 3–18, 1996. doi: 10.1147/rd.401.0003.

[63] SEGURA, J., HAWKINS, C. F. “Failure Mechanisms in CMOS IC Materials”. In: *CMOS Electronics: How It Works, How It Fails*, 2004. doi: 10.1002/0471728527.

[64] KEANE, J., KIM, C. H. “Transistor Aging”. Apr 2011. Available: <`https://spectrum.ieee.org/transistor-aging`>. Visited on: 06-19-22.

[65] SLEGEL, T., AVERILL, R., CHECK, M., et al. “IBM’s S/390 G5 microprocessor design”, *IEEE Micro*, v. 19, n. 2, pp. 12–23, 1999. doi: 10.1109/40.755464.

[66] BANNON, P., VENKATARAMANAN, G., SARMA, D. D., et al. “Computer and Redundancy Solution for the Full Self-Driving Computer”. In: *2019 IEEE Hot Chips 31 Symposium (HCS)*, pp. 1–22, 2019. doi: 10.1109/HOTCHIPS.2019.8875645.

[67] MUKHERJEE, S., KONTZ, M., REINHARDT, S. “Detailed design and evaluation of redundant multi-threading alternatives”. In: *Proceedings 29th*

*Annual International Symposium on Computer Architecture*, pp. 99–110, 2002. doi: 10.1109/ISCA.2002.1003566.

[68] DÖBEL, B., HÄRTIG, H., ENGEL, M. "Operating System Support for Redundant Multithreading". In: *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, p. 83–92, New York, NY, USA, 2012. Association for Computing Machinery. ISBN: 9781450314251. doi: 10.1145/2380356.2380375. Available: <`https://doi.org/10.1145/2380356.2380375`>.

[69] PATEL, J. H., FUNG, L. Y. "Concurrent error detection in ALUS by recomputing with shifted operands", *IEEE Trans. Comput.; (United States)*, v. 7, 7 1982. doi: 10.1109/TC.1982.1676055.

[70] NICOLAIDIS, M. "Time redundancy based soft-error tolerance to rescue nanometer technologies". In: *Proceedings 17th IEEE VLSI Test Symposium (Cat. No.PR00146)*, pp. 86–94, 1999. doi: 10.1109/VTEST.1999.766651.

[71] ARLAT, J., AGUERA, M., AMAT, L., et al. "Fault injection for dependability validation: a methodology and some applications", *IEEE Transactions on Software Engineering*, v. 16, n. 2, pp. 166–182, 1990. doi: 10.1109/32.44380.

[72] CHO, H., MIRKHANI, S., CHER, C.-Y., et al. "Quantitative evaluation of soft error injection techniques for robust system design". In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–10, 2013.

[73] LI, G., PATTABIRAMAN, K., CHER, C.-Y., et al. "Understanding Error Propagation in GPGPU Applications". In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 240–251, 2016. doi: 10.1109/SC.2016.20.

[74] LI, G., PATTABIRAMAN, K., HARI, S. K. S., et al. "Modeling Soft-Error Propagation in Programs". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 27–38, 2018. doi: 10.1109/DSN.2018.00016.

[75] MAHMOUD, A., AGGARWAL, N., NOBBE, A., et al. "PyTorchFI: A Runtime Perturbation Tool for DNNs". In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 25–31, 2020.

[76] CHEN, Z., NARAYANAN, N., FANG, B., et al. “TensorFI: A Flexible Fault Injection Framework for TensorFlow Applications”. In: Vieira, M., Madeira, H., Antunes, N., et al. (Eds.), *31st IEEE International Symposium on Software Reliability Engineering, ISSRE 2020, Coimbra, Portugal, October 12-15, 2020*, pp. 426–435. IEEE, 2020. doi: 10.1109/ISSRE5003.2020.00047. Available: <`https://doi.org/10.1109/ISSRE5003.2020.00047`>.

[77] GOLDSTEIN, B. F., SRINIVASAN, S., DAS, D., et al. “Reliability Evaluation of Compressed Deep Learning Models”. In: *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, pp. 1–5, 2020. doi: 10.1109/LASCAS45839.2020.9069026.

[78] CHATZIDIMITRIOU, A., BODMANN, P., PAPADIMITRIOU, G., et al. “Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 26–38, 2019. doi: 10.1109/DSN.2019.00018.

[79] PASZKE, A., GROSS, S., MASSA, F., et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: Wallach, H., Larochelle, H., Beygelzimer, A., et al. (Eds.), *Advances in Neural Information Processing Systems 32*, Curran Associates, Inc., pp. 8024–8035, 2019.

[80] YANG, T.-J., CHEN, Y.-H., SZE, V. “Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6071–6079, 2017. doi: 10.1109/CVPR.2017.643.

[81] ZMORA, N., JACOB, G., NOVIK, G. “Neural Network Distiller”. jun. 2018. Available: <`https://doi.org/10.5281/zenodo.1297430`>.

[82] LU, Q., FARAHANI, M., WEI, J., et al. “LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults”. In: *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 11–16, 2015. doi: 10.1109/QRS.2015.13.

[83] ABADI, M., AGARWAL, A., BARHAM, P., et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems”. 2015. Available: <`https://www.tensorflow.org/`>. Software available from tensorflow.org.

[84] LI, G., HARI, S. K. S., SULLIVAN, M., et al. “Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Ap-

plications". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN: 9781450351140. doi: 10.1145/3126908.3126964. Available: <`https://doi.org/10.1145/3126908.3126964`>.

[85] SHI, Q., OMAR, H., KHAN, O. "Exploiting the Tradeoff between Program Accuracy and Soft-error Resiliency Overhead for Machine Learning Workloads", *arXiv Prepr.*, 2017.

[86] REAGEN, B., GUPTA, U., PENTECOST, L., et al. "Ares: A framework for quantifying the resilience of deep neural networks". In: *2018 55th ACM/ESDA/IEEE Des. Autom. Conf.*, pp. 1–6. IEEE, jun 2018. ISBN: 978-1-5386-4114-9. doi: 10.1109/DAC.2018.8465834.

[87] SALAMI, B., UNSAL, O. S., KESTELMAN, A. C. "On the Resilience of RTL NN Accelerators: Fault Characterization and Mitigation". In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 322–329, Sep. 2018. doi: 10.1109/CAHPC.2018.8645906.

[88] PIURI, V. "Analysis of Fault Tolerance in Artificial Neural Networks", *Journal of Parallel and Distributed Computing*, v. 61, n. 1, pp. 18 – 48, 2001. ISSN: 0743-7315. doi: https://doi.org/10.1006/jpdc.2000.1663. Available: <`http://www.sciencedirect.com/science/article/pii/S0743731500916630`>.

[89] REAGEN, B., WHATMOUGH, P., ADOLF, R., et al. "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, June 2016. doi: 10.1109/ISCA.2016.32.

[90] LINDSAY, G. W. "Convolutional Neural Networks as a Model of the Visual System: Past, Present, and Future", *Journal of Cognitive Neuroscience*, v. 33, n. 10, pp. 2017–2031, 09 2021. ISSN: 0898-929X. doi: 10.1162/jocn_a_01544. Available: <`https://doi.org/10.1162/jocn_a_01544`>.

[91] MASI, I., WU, Y., HASSNER, T., et al. "Deep Face Recognition: A Survey". In: *2018 31st SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, pp. 471–478, 2018. doi: 10.1109/SIBGRAPI.2018.00067.

[92] WOLF, T., DEBUT, L., SANH, V., et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". 2019. Available: <`https://arxiv.org/abs/1910.03771`>.

[93] GOOGLE-WAYMO. "Introducing Waymo's suite of custom-built, self-driving hardware". 2017. Available: <`https://blog.waymo.com/2019/08/introducing-waymos-suite-of-custom.html`>.

[94] TESLA. "All Tesla Cars Being Produced Now Have Full Self-Driving Hardware". 2016. Available: <`https://www.tesla.com/blog/all-tesla-cars-being-produced-now-have-full-self-driving-hardware`>.

[95] TALPES, E., OTHERS. "Compute Solution for Tesla's Full Self-Driving Computer", *IEEE Micro*, v. 40, n. 2, pp. 25–35, 2020.

[96] J. DONGARRA, H. M., STROHMAIER, E. "ISO 26262-1:2018 Road vehicles–Functional safety". 2015.

[97] FETZER, C., OTHERS. "AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware". In: *Computer Safety, Reliability, and Security*, 2009.

[98] CHEN, Y.-H., EMER, J., SZE, V. "Eyeriss v2: A Flexible and High-Performance Accelerator for Emerging Deep Neural Networks", v. 3, 2018.

[99] WELLS, P. M., OTHERS. "Adapting to Intermittent Faults in Multicore Systems". In: *ASPLOS'08*, 2008. doi: 10.1145/1346281.1346314.

[100] RASHID, L., PATTABIRAMAN, K., GOPALAKRISHNAN, S. "Characterizing the Impact of Intermittent Hardware Faults on Programs", *IEEE Transactions on Reliability*, v. 64, n. 1, pp. 297–310, 2015.

[101] GUAN, H., OTHERS. "In-Place Zero-Space Memory Protection for CNN". In: *NeurIPS*, 2019.

[102] AZIZIMAZREAH, A., GU, Y., GU, X., et al. "Tolerating Soft Errors in Deep Learning Accelerators with Reliable On-Chip Memory Designs", *2018 IEEE Int. Conf. Networking, Archit. Storage*, 2018. doi: 10.1109/NAS.2018.8515692.

[103] QIN, M., SUN, C., VUCINIC, D. "Robustness of Neural Networks against Storage Media Errors". In: *arXiv preprint arXiv:1709.06173*, 2017.

[104] KOPPULA, S., OTHERS. "EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM". In: *MICRO'52*, 2019.

[105] ZHANG, J. J., OTHERS. "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator". In: *IEEE 36th VLSI Test Symposium (VTS)*, 2018.

[106] GAMBARDELLA, G., OTHERS. "Efficient Error-Tolerant Quantized Neural Network Accelerators". In: *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, 2019.

[107] FEINBERG, B., WANG, S., IPEK, E. "Making Memristive Neural Network Accelerators Reliable". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 52–65, 2018.

[108] KINGMA, D. P., BA, J. "Adam: A Method for Stochastic Optimization". 2014. Available: <`https://arxiv.org/abs/1412.6980`>.

[109] KIEFER, J., WOLFOWITZ, J. "Stochastic Estimation of the Maximum of a Regression Function", *Annals of Mathematical Statistics*, v. 23, pp. 462–466, 1952.

[110] HE, K., ZHANG, X., REN, S., et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 770–778. IEEE, jun 2016. ISBN: 978-1-4673-8851-1. doi: 10.1109/CVPR.2016.90.

[111] XU, Y., OSEP, A., BAN, Y., et al. "How to Train Your Deep Multi-Object Tracker". In: *Proceedings of CVPR'20*, June 2020.

[112] NAUMOV, M., MUDIGERE, D., SHI, H.-J. M., et al. "Deep Learning Recommendation Model for Personalization and Recommendation Systems". 2019. Available: <`https://arxiv.org/abs/1906.00091`>.

[113] BROWN, T. B., OTHERS. "Language Models are Few-Shot Learners". 2020.

[114] TRAMÈR, F., ZHANG, F., JUELS, A., et al. "Stealing Machine Learning Models via Prediction APIs". In: *25th USENIX Security Symposium (USENIX Security 16)*, pp. 601–618, Austin, TX, ago. 2016. USENIX Association. ISBN: 978-1-931971-32-4. Available: <`https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer`>.

[115] GILAD-BACHRACH, R., DOWLIN, N., LAINE, K., et al. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy". In: Balcan, M. F., Weinberger, K. Q. (Eds.), *Proceedings of The 33rd International Conference on Machine Learning*, v. 48, *Proceedings of Machine Learning Research*, pp. 201–210, New York, New York, USA, 20–22 Jun 2016. PMLR. Available: <`https://proceedings.mlr.press/v48/gilad-bachrach16.html`>.

[116] ARM. *Arm® Ethos™-U NPU Application development overview.* Relatório técnico, Arm, 2020. Available: <`https://documentation-service.arm.com/static/5fae5cefca04df4095c1ca31?token=`>. Visited on: 05-12-22.

[117] JOUPPI, N. P., OTHERS. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, p. 1–12, New York, NY, USA, 2017. Association for Computing Machinery. ISBN: 9781450348928. doi: 10.1145/3079856.3080246. Available: <`https://doi.org/10.1145/3079856.3080246`>.

[118] SKILLMAN, A., EDSO, T. "A technical overview of the Arm Cortex-M55 and Ethos-U55: ARM's most capable processors for endpoint AI". In: *Hot Chips 32: A Symposium on High Performance Chips*, 2020.

[119] SONG, J., CHO, Y., PARK, J., et al. "7.1 An 11.5TOPS/W 1024-MAC Butterfly Structure Dual-Core Sparsity-Aware Neural Processing Unit in 8nm Flagship Mobile SoC". In: *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*, pp. 130–132, 2019. doi: 10.1109/ISSCC.2019.8662476.

# Appendix A

# List of Publications

1. Goldstein, B. F., Srinivasan, S., Das, D., Banerjee, K., Santiago, L., Ferreira, V. C., Nery, A. S., Kundu, S., França, F. M. G. (2020). "Reliability Evaluation of Compressed Deep Learning Models". *2020 IEEE 11th Latin American Symposium on Circuits Systems (LASCAS)*, 1–5. doi:10.1109/LASCAS45839.2020.9069026

2. Goldstein, B. F., Ferreira, V. C., Srinivasan, S., Das, D., Nery, A. S., Kundu, S., França, F. M. G. (2021). "A Lightweight Error-Resiliency Mechanism for Deep Neural Networks." *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 311–316. doi:10.1109/ISQED51717.2021.9424287

3. Goldstein, B. F., Patil, V. C., Ferreira, V. C., Nery, A. S., França, F. M. G., Kundu, S. (2021). "Preventing DNN Model IP Theft via Hardware Obfuscation." *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 11(2), 267–277. doi:10.1109/JETCAS.2021.3076151

4. Bacellar, A. T. L., Goldstein, B. F., da Cruz Ferreira, V., Santiago, L., Lima, P. M. V., França, F. M. G. (2020). "Fast Deep Neural Networks Convergence using a Weightless Neural Model." *28th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning – ESANN 2020*, Bruges, Belgium, October 2-4, 2020, 211–216.

5. Do, J., Ferreira, V. C., Bobarshad, H., Torabzadehkashi, M., Rezaei, S., Heydarigorji, A., Souza, D., Goldstein, B., ... Alves, V. (2020). "Cost-Effective, Energy-Efficient, and Scalable Storage Computing for Large-Scale AI Applications.", *ACM Transactions on Storage*, V.16(4). doi:10.1145/3415580