



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dept. of Computer Engineering

Protecting the Parameters of Floating-Point-Based
Convolutional Neural Networks Against Accidental and
Malicious Faults without Increasing Their Memory Footprint

Master's Thesis

Master's Degree in Computer and Network Engineering

AUTHOR: Silin, Andrei

Tutor: Saiz Adalid, Luis José

Cotutor: Andrés Martínez, David de

ACADEMIC YEAR: 2023/2024

Resumen

Las redes neuronales convolucionales (CNN) son, de hecho, el método estándar para clasificación de imágenes en diversos dominios, incluyendo reconocimiento facial automático en sistemas de protección de fronteras, conducción autónoma en vehículos, sanidad, etc. Desplegar una CNN requiere encontrar un equilibrio entre objetivos contrapuestos, como productividad, precisión y consumo de energía. En entornos críticos, asegurar un nivel aceptable de robustez contra fallos es de vital importancia. Millones de parámetros, cargados desde memoria principal a los buffers de los aceleradores de las CNN, son usados repetidamente en el proceso de inferencia. Bit-flips accidentales o maliciosos en esos buffers pueden afectar negativamente a la precisión de la red. Las soluciones tradicionales, basadas en redundancia, pueden aportar una elevada cobertura de errores, pero con una elevada sobrecarga, en ocasiones inasumible, especialmente en soluciones con recursos limitados.

Este trabajo propone una nueva metodología para localizar bits en los parámetros en coma flotante de una CNN que no son necesarios (por ser irrelevantes o invariantes). Por una parte, la representación en memoria de valores en coma flotante (frecuentemente usando el estándar IEEE-754 de 32 bits) tiene una precisión muy elevada, innecesaria para el proceso de inferencia de una CNN. Así pues, se puede utilizar la inyección de fallos para determinar los bits que son irrelevantes para el proceso de inferencia (generalmente los bits menos significativos). Por otra parte, la mayoría de los parámetros están incluidos en un pequeño rango de valores y, por tanto, sus representaciones tienen exponentes similares. Analizando estos valores, es posible encontrar bits invariantes (que tienen el mismo valor en todos los parámetros). Incluso se pueden realizar ligeras modificaciones a determinados parámetros para incrementar el número de bits invariantes.

Los bits invariantes y los irrelevantes no afectan al proceso de inferencia, así que se pueden utilizar como bits de paridad para códigos correctores de errores. Es importante remarcar que esta metodología mantiene la precisión de la CNN y su huella en memoria, y puede desplegarse sin reentrenar la red. Se ha aplicado a diferentes CNN de PyTorch entrenadas previamente para demostrar la validez y aplicabilidad general de la propuesta.

Palabras clave: Redes Neuronales Convolucionales, Confiabilidad, Inyección de fallos, Coma flotante, Redundancia, Optimización a nivel de bits, Códigos de corrección de errores.

Abstract

Convolutional Neural Networks (CNNs) are the de facto standard method for image classification in various domains, including automatic face recognition in border-protection systems, autonomous driving in vehicles, health care, etc. Deploying CNNs requires balancing conflicting goals, like throughput, accuracy, and power consumption. In safety-critical environments, ensuring acceptable levels of robustness against faults is also of utmost importance. Millions of parameters, loaded from main memory into the internal buffers of CNN accelerators, are repeatedly used in the inference process. Accidental and malicious bit-flips targeting these buffers may negatively impact CNN's accuracy. Traditional redundancy-based solutions provide high error coverage at the cost of high, and sometimes unaffordable, overheads, especially for resource-constrained solutions.

This work proposes a novel methodology to locate bits in the floating-point (FP) parameters of a CNN that are not necessary (irrelevant or invariant). On the one hand, the memory representation of FP values (frequently 32-bit IEEE-754 standard) has a very high precision, unnecessary for the CNN inference process. Thus, fault injection can be used to determine the bits that are irrelevant to the inference process (usually the least significant bits). On the other hand, most of the parameters are within a short range of values and, therefore, their representations have similar exponents. By analyzing these values, it is possible to find invariant bits, i.e. bits that have always the same value in all parameters. Even more, slight modifications can be applied to selected parameters to increase the number of invariant bits.

Irrelevant and invariant bits do not affect the inference process, so they can be used as parity bits for error correction codes (ECCs). It is important to note that this methodology preserves the CNN accuracy and its memory footprint, and it can be deployed without retraining the network. It has been applied to different PyTorch pre-trained CNNs to demonstrate the validity and general applicability of the approach.

Keywords: Convolutional Neural Networks (CNN), Dependability, Fault Injection, Floating Point, Redundancy, Bit-Level Optimization, Error Correcting Codes.

Table of Contents

| | | |
|-------|--|----|
| 1. | Introduction | 6 |
| 1.1 | Context and Motivation | 6 |
| 1.2 | Problem Statement..... | 7 |
| 1.3 | Proposed Solution..... | 8 |
| 1.4 | Objectives | 9 |
| 1.5 | Document Structure | 10 |
| 2. | Background and Related Work | 12 |
| 2.1 | Development of Convolutional Neural Networks (CNNs)..... | 12 |
| 2.2 | CNNs Utilizing Float32 and the Shift Toward Quantized Networks | 12 |
| 2.3 | Errors in CNNs: Causes, Effects, and Mitigation Techniques..... | 13 |
| 2.3.1 | Impact of Errors on CNNs..... | 13 |
| 2.3.2 | CNN Robustness and Fault Tolerance..... | 14 |
| 2.3.3 | Error Mitigation Strategies in CNNs..... | 14 |
| 2.4 | Existing Research on Bit-Level Error Correction in CNNs..... | 15 |
| 3. | Methodology | 16 |
| 3.1 | Overview of Neural Networks and Parameter Roles..... | 16 |
| 3.2 | Representation of float32 According to IEEE 754 | 19 |
| 3.3 | Golden run | 19 |
| 3.3.1 | Main code blocks..... | 20 |
| 3.3.2 | Output example | 23 |
| 3.3.3 | Analysis of the Golden Run | 23 |
| 3.4 | Identification of Significant Bits..... | 23 |
| 3.4.1 | Algorithm Overview..... | 24 |
| 3.4.2 | Example Output..... | 25 |
| 3.4.3 | Analysis of Significant Bits..... | 25 |
| 3.5 | Analysis of the Sign Bit and Exponent Values..... | 27 |
| 3.5.1 | Preliminary Analysis of Bit Values | 28 |
| 3.5.2 | Example Output..... | 28 |
| 3.5.3 | Analysis of Invariants Bits | 28 |
| 3.6 | Analysis of Parameter Values Across Network Layers..... | 31 |
| 3.6.1 | Exponent Distribution Analysis | 32 |
| 3.6.2 | Parameter Categorization and Exponent Range Analysis | 34 |

| | | |
|-------|---|----|
| 3.7 | Rounding Experiments on Exponent Bits..... | 36 |
| 3.7.1 | Experimental Goal..... | 36 |
| 3.7.2 | Experimental Procedure | 36 |
| 3.7.3 | Key Observations: | 38 |
| 3.8 | Summary of Results and Final Analysis..... | 40 |
| 3.8.1 | Categories of values of bits. | 40 |
| 3.8.2 | Bits to Protect: | 41 |
| 3.8.3 | Final Result and Potential Application of Error-Correcting Codes (ECC): | 41 |
| 3.8.4 | Potential Application of Error-Correcting Codes (ECC): | 42 |
| 3.9 | Summary of the Methodology | 43 |
| 4. | Experiments and Results | 45 |
| 4.1 | Golden Run Results | 45 |
| 4.2 | Identification of Least Significant Bits (LSBs) | 47 |
| 4.3 | Analysis of Invariant Bits | 48 |
| 4.4 | Assessment of Parameter Magnitudes | 49 |
| 4.5 | Rounding and Bit-Fixing Analysis | 50 |
| 4.6 | Evaluation of Available Parity Bits and Data Bits for ECC | 51 |
| 4.6.1 | Analysis Summary: | 51 |
| 4.6.2 | Overview of Results: | 52 |
| 4.6.3 | Parity Bits and Protection Requirements:..... | 53 |
| 5. | Conclusions and future work..... | 56 |
| 5.1 | Key Contributions..... | 56 |
| 5.2 | Limitations..... | 56 |
| 5.3 | Future Work..... | 57 |
| 5.4 | Conclusion | 58 |
| 6. | References | 59 |
| 7. | Annexes..... | 61 |
| 7.1 | Model Wrapper..... | 61 |
| 7.1.1 | Model Wrapper for On-the-Fly Parameter Correction | 61 |
| 7.1.2 | Example of Parameter and Buffer Correction | 62 |
| 7.1.3 | Application of the Wrapper | 62 |
| 7.1.4 | Future Work on Wrapper Extensions | 62 |
| 7.2 | Assessment of Parameter Magnitudes | 63 |
| 7.3 | Rounding and Bit-Fixing Analysis | 65 |

1. Introduction

This chapter provides an overview of the background and motivation for this thesis, focusing on the challenges and solutions in ensuring the fault tolerance of Convolutional Neural Networks (CNNs). It outlines the key issues related to computational and memory constraints in CNNs, particularly in the context of deployment on resource-limited devices. The chapter also discusses how quantization techniques help address these challenges but introduce new problems related to network robustness and fault tolerance. The subsequent sections will delve into the context and motivation behind this research, explore related work in the field, and present the research objectives and contributions of this thesis.

1.1 Context and Motivation

In recent years, deep learning has become a cornerstone of various fields, from computer vision to natural language processing, and Convolutional Neural Networks (CNNs) have played a pivotal role in this success [1]. CNNs are particularly well-suited for tasks involving image recognition, object detection, and other forms of visual data processing due to their ability to automatically learn spatial hierarchies of features from input data [2]. However, as the complexity and size of CNN models have increased, so too have their computational and memory requirements, which presents significant challenges when deploying these models on resource-constrained devices such as mobile phones, embedded systems, and edge devices [3].

To address these challenges, quantization techniques have been developed to reduce the precision of the network's parameters, such as weights and activations, thereby decreasing the model's memory footprint and computational cost [4]. For example, instead of using 32-bit floating-point representations, parameters can be quantized to 16-bit or even 8-bit integers with minimal loss in accuracy [5]. This reduction in precision not only lowers memory usage but also enables faster inference and lower power consumption, making quantization a highly effective approach for deploying CNNs in real-time and energy-efficient applications [6].

However, quantization introduces new challenges, particularly in terms of model robustness and fault tolerance. In critical applications, where the reliability of neural networks is paramount, even minor errors in model parameters can lead to significant degradation in performance. This is especially true in quantized models, where the reduced precision means that there is less redundancy to absorb errors, making the network more susceptible to faults [7]. For instance, bit flips caused by cosmic rays or other sources of noise can have a disproportionate impact on the performance of quantized CNNs compared to their full-precision counterparts.

The need for fault tolerance in neural networks has led to the exploration of various techniques, such as error-correcting codes (ECC) and redundant computations [8]. However, these methods typically require additional memory or computational resources, which can negate the benefits gained from quantization. As a result, there is a growing interest in developing novel approaches that can provide fault tolerance without significantly increasing the memory or computational burden of the network. This thesis explores one such approach: leveraging the "insignificant" bits within 32-bit floating-point parameters of CNNs for fault tolerance.

Recent research has shown that not all bits in a floating-point representation contribute equally to the accuracy or performance of a CNN [9]. Some bits, particularly those representing less significant decimal places, may have little to no impact on the model's overall performance [10]. By identifying and repurposing these insignificant bits, it may be possible to embed parity information directly into the model's parameters, thereby enabling fault detection and correction without the need for additional memory. This approach has the potential to enhance the robustness of quantized CNNs while preserving their memory efficiency, making it a promising avenue for further investigation [11].

1.2 Problem Statement

As Convolutional Neural Networks (CNNs) continue to evolve and find applications in increasingly critical and resource-constrained environments, ensuring their robustness and reliability becomes a priority. CNNs are often implemented using 32-bit floating-point representations for their parameters, not only because it provides a good balance between precision and computational efficiency but also because FP32 is the standard floating-point format used in most computers, with processors optimized to accelerate arithmetic operations in this format. However, these networks remain vulnerable to errors that can arise from various sources, such as hardware faults, environmental disturbances, or cosmic radiation. Such errors can manifest as bit flips or other forms of data corruption in the model's parameters, leading to potentially severe degradation in network performance.

Traditional approaches to fault tolerance in neural networks typically involve the introduction of redundancy, either in the form of additional parity bits or through error-correcting codes (ECC). Parity bits can be used to detect and sometimes correct errors in the data, while ECCs can provide more robust error correction capabilities. However, these techniques generally require extra storage space to accommodate the additional information needed for error detection and correction. In the context of CNNs with 32-bit floating-point parameters, this additional storage can become significant, as the number of parameters may reach several millions, counteracting efforts to optimize the network's memory usage.

The problem is particularly acute because the primary motivation for many CNN applications is to maximize accuracy and performance while minimizing resource consumption. For instance, in embedded systems or edge devices where memory is limited, increasing the memory footprint to incorporate fault tolerance can undermine the efficiency gains achieved through careful network design and parameter optimization. This creates a fundamental tension between the need for robustness and the imperative to conserve memory resources.

Moreover, the precise nature of 32-bit floating-point representation adds another layer of complexity to this problem. In a floating-point number, certain bits contribute more significantly to the numerical value than others. Traditional fault tolerance methods do not differentiate between these bits, treating all of them as equally important. This approach can be overly conservative, leading to unnecessary memory overhead. Therefore, a more nuanced method of protecting CNNs is required—one that recognizes and leverages the varying significance of different bits within the 32-bit floating-point representation.

The key challenge, then, is to develop a methodology that allows for the protection of CNNs against errors without the need for additional memory. This thesis proposes to explore

whether certain bits in the 32-bit floating-point representation—those that contribute minimally to the overall accuracy of the network—can be repurposed for fault tolerance. The central research question is: How can we identify these less significant bits and use them to embed error-detection and correction information without increasing the overall memory footprint of the network?

By addressing this question, the thesis aims to contribute to the development of more efficient and robust CNNs, suitable for deployment in memory-constrained environments where reliability is critical. This approach, if successful, could provide a new pathway for balancing the competing demands of memory efficiency and fault tolerance in modern deep learning applications.

1.3 Proposed Solution

To address the challenge of protecting Convolutional Neural Networks (CNNs) that use 32-bit floating-point representations from faults without increasing their memory footprint, this thesis proposes a novel approach that leverages the existing bits within the floating-point parameters. Specifically, the solution involves identifying and utilizing the “insignificant” bits in the mantissa—those that have minimal impact on the network’s accuracy—alongside an analysis of the exponent bits, to embed fault-tolerance mechanisms directly within the network parameters.

Key Aspects of the Proposed Solution:

1. Utilization of Insignificant Bits in the Mantissa:

- **Concept:** In a 32-bit floating-point representation, the mantissa (or significand) contains the most detailed part of the number. However, not all bits within the mantissa are equally important for maintaining the accuracy of the CNN. The idea is to identify those bits that, when altered, do not significantly affect the network's accuracy. These insignificant bits can then be repurposed to store parity information or other forms of error-correction data, effectively adding fault tolerance without requiring additional memory.

2. Analysis and Adjustment of Exponent Bits:

- **Concept:** The exponent in a floating-point number determines the scale of the value. This thesis proposes to analyze the sensitivity of the network to changes in the exponent bits across different types of parameters, such as weights, biases, running mean, and running variance. The goal is to identify invariant bits (pure invariants) and less critical exponent values, which can be rounded with minimal impact on precision. By doing so, it may be possible to modify the floating-point representation in a way that reduces the impact on accuracy, thus freeing up space within the existing bit structure for embedding fault-tolerance mechanisms.

3. Integration of Fault-Tolerance Mechanisms:

- **Concept:** Based on the analyses of the mantissa and exponent bits, the thesis will develop a methodology to integrate fault-tolerance mechanisms directly into the CNN’s parameters. This approach will aim to protect the network from common types of errors, such as bit flips, without increasing the overall memory usage.

The fault-tolerance information will be strategically embedded in the insignificant bits identified during the analysis phase.

Expected Benefits:

- **Memory Efficiency:** By embedding fault-tolerance mechanisms within the existing bits of the 32-bit floating-point representation, the need for additional memory is eliminated, preserving the compactness and efficiency of the CNN.
- **Robustness:** The proposed solution enhances the fault tolerance of CNN, making it more resilient to errors without compromising on accuracy.
- **Scalability:** This approach is adaptable to various types of CNN architectures and can be applied to different components of the network, such as weights and biases, making it a versatile solution for a wide range of applications.

This thesis seeks to demonstrate that by carefully analyzing and repurposing certain bits within the 32-bit floating-point parameters, it is possible to achieve a balance between robustness and memory efficiency in CNNs, offering a practical solution for deployment in environments where both reliability and resource constraints are critical.

1.4 Objectives

The primary objective of this thesis is to develop and validate a methodology for enhancing the fault tolerance of Convolutional Neural Networks (CNNs) that utilize 32-bit floating-point representations, without increasing their memory footprint. This will be achieved through a detailed analysis of the bits within the floating-point representation, with a particular focus on identifying and exploiting bits that have minimal impact on the network's accuracy. The specific objectives of this research are as follows:

1. Analysis of Less Significant Bits in the Mantissa:

- **Objective:** To systematically identify the bits within the mantissa of 32-bit floating-point numbers that are less significant, meaning that their alteration has minimal impact on the accuracy of the CNN.
- **Approach:** This involves evaluating how changes to specific bits in the mantissa affect the accuracy of the network. By analyzing the sensitivity of the CNN to these changes, it will be possible to determine which bits can be safely modified or repurposed for storing error-correction information. The focus will be on identifying the least significant bits that, when altered, result in negligible changes to the network's outputs.

2. Analysis of Exponent Bits:

- **Objective:** To investigate the role of the exponent bits in the floating-point representation and determine the impact of fixing certain bits on the network's accuracy. This analysis will be conducted across different components of the CNN, including weights, biases, running mean, and running variance.

- **Approach:** The study will involve selectively fixing or altering specific bits and values in the exponent to assess how these changes influence the overall performance of the network. This will help identify which exponent bits are critical to maintaining accuracy and which can be altered with minimal impact. The analysis will differentiate between the effects on various parameters (weights, biases, `running_mean`, `running_var`) to understand how the significance of bits may vary across different parts of the network.

3. Development of a Methodology for Fault Tolerance:

- **Objective:** To propose a methodology that utilizes the identified insignificant bits for embedding fault-tolerant information, such as parity bits or other forms of error detection and correction codes, without increasing the memory footprint.
- **Approach:** Based on the findings from the analysis of mantissa and exponent bits, a systematic approach will be developed to incorporate fault-tolerance mechanisms directly within the existing 32-bit floating-point representation. This methodology will aim to maximize the use of available bits without compromising the network's accuracy or memory efficiency.

4. Evaluation and Validation:

- **Objective:** To rigorously evaluate the proposed fault-tolerance methodology in terms of its effectiveness in protecting CNNs against errors, while maintaining their accuracy and efficiency.
- **Approach:** The evaluation will involve implementing the proposed method in practical CNN models and conducting experiments to assess its impact on both fault tolerance and network accuracy and performance. The results will be compared to traditional fault-tolerance approaches to highlight the benefits and potential trade-offs of the new method.

By achieving these objectives, the thesis aims to provide a novel approach to fault tolerance in CNNs that preserves memory efficiency and could be particularly valuable for deployment in resource-constrained environments. This research will contribute to the broader understanding of how to balance the competing demands of accuracy, reliability, and memory usage in deep learning applications.

1.5 Document Structure

The document is structured as follows:

- Chapter 2: Background and Related Work - This chapter provides an overview of CNN quantization, fault tolerance in neural networks, and related work in the field.
- Chapter 3: Methodology - Here, we describe the proposed methodology for identifying insignificant bits and embedding parity information.
- Chapter 4: Experiments and Results - This chapter presents the experiments conducted to validate the proposed methodology and discusses the results.

- Chapter 5: Conclusion and Future Work - The final chapter summarizes the findings of the thesis and suggests directions for future research.

2. Background and Related Work

This chapter provides an overview of the development of Convolutional Neural Networks (CNNs), their applications, and the evolution of techniques used to improve their efficiency and robustness. We will explore key milestones in the progression of CNN architectures, examine the impact of float32 representations and quantization techniques, and discuss existing research on error correction methods and fault tolerance in CNNs.

2.1 Development of Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) have become a cornerstone in the field of deep learning, particularly in areas such as image recognition, natural language processing, and various other pattern recognition tasks. The origin of CNNs can be traced back to the work of LeCun et al. (1998) [12] on the LeNet architecture, which was primarily designed for digit recognition tasks. This architecture introduced the concept of convolutional layers combined with subsampling (pooling) layers, enabling the network to extract hierarchical features from input images.

Over the years, CNNs have evolved significantly, with major milestones including the introduction of the AlexNet architecture by Krizhevsky et al. (2012) [1], which won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) and demonstrated the power of deep convolutional networks for large-scale image classification. Subsequent architectures like VGGNet, GoogLeNet, and ResNet further refined the depth and complexity of CNNs, introducing new techniques such as very deep networks and residual connections that helped mitigate the vanishing gradient problem.

These developments have led to CNNs becoming the de facto standard for many vision-based tasks, and their performance has continued to improve with advances in hardware and optimization techniques. However, the high computational and memory demands of CNNs, particularly those using 32-bit floating-point (float32) representations, have driven research towards more efficient implementations, including model compression and quantization. [13][2]

2.2 CNNs Utilizing Float32 and the Shift Toward Quantized Networks

The widespread use of 32-bit floating-point representations in CNNs is largely due to their balance between precision and computational efficiency. Float32 allows networks to capture a wide dynamic range of values, which is crucial for training deep models with large datasets. However, as CNNs have grown in size and complexity, the limitations of float32 in terms of memory footprint and power consumption have become increasingly apparent.

To address these challenges, researchers have explored various quantization techniques aimed at reducing the bit-width of CNN parameters, thereby lowering memory usage and computational demands without significantly compromising model accuracy. Courbariaux, M., Bengio, Y., & David, J. P. (2015) [14] were among the pioneers in this area, introducing

BinaryConnect, which uses binary weights during the forward and backward passes of training while retaining float32 for parameter updates. This approach was followed by more sophisticated methods like Quantized Neural Networks (QNNs) [5], which reduce weights and activations to lower bit-widths such as 8-bit integers.

Quantization has become a standard technique for deploying CNNs on resource-constrained devices, such as mobile phones and embedded systems, where power efficiency and low memory usage are critical. Despite the advantages of quantization, float32 representations are still widely used, particularly in high-performance applications where accuracy is paramount, necessitating continued research into optimizing float32 CNNs.

2.3 Errors in CNNs: Causes, Effects, and Mitigation Techniques

Errors in CNNs can arise from various sources, such as hardware faults, environmental influences like cosmic radiation, and quantization effects. These errors can manifest as bit-flips, leading to incorrect outputs or degraded performance. In safety-critical applications like autonomous driving or medical diagnosis, fault tolerance is essential. Here, we expand on the effects of these errors and the robustness of CNNs in handling them.

2.3.1 Impact of Errors on CNNs

Soft errors, often caused by high-energy particles, can significantly affect CNNs deployed in safety-critical environments. For instance, as demonstrated in [15], soft errors in CNN accelerators can lead to Silent Data Corruptions (SDCs). These errors propagate through the network and may cause incorrect predictions, such as misclassifying a truck as a bird in autonomous vehicle systems. The consequences of such misclassification can be catastrophic, especially in time-sensitive applications like self-driving cars, where misinterpretation of critical objects could lead to fatal accidents. As shown in the example in **Figure 1**, the truck is misidentified as a bird and brakes may not be applied.

Figure 1. Example of SDC that could lead to collision in self-driving cars due to soft errors: (left) Fault-free execution and (right) SDC.



The resilience of CNNs against errors depends on several factors: the network topology, data types used, layer sensitivity, and the hardware implementation. Different layers in a network exhibit varying sensitivities to errors. For instance, normalization layers can mitigate the impact

of errors by averaging faulty values with adjacent correct values, while fully connected layers, due to their direct impact on output predictions, are more vulnerable to errors.

2.3.2 CNN Robustness and Fault Tolerance

Recent studies have explored different methods to improve the fault tolerance of CNNs. One common approach is using Triple Modular Redundancy (TMR), which replicates hardware components to detect and correct errors. However, such methods incur high overheads in terms of energy and hardware costs, making them challenging for deployment in low-power, real-time applications like autonomous systems.

Several more lightweight approaches have been proposed. Selective hardening techniques focus on protecting the most vulnerable bits or components, thereby reducing the overhead. For example, [16] highlights that selective latch hardening applied to the most sensitive bits in a CNN accelerator can significantly reduce the failure rate with minimal area overhead. Similarly, symptom-based detectors can be employed to identify abnormal behaviors in CNN outputs, thereby detecting and correcting faults with high precision.

2.3.3 Error Mitigation Strategies in CNNs

To mitigate the effects of soft errors, CNNs can benefit from a combination of hardware and software approaches:

1. **Error Detection and Correction (EDAC):** Parity bits and Hamming codes can be used to protect critical data in CNNs, ensuring that any errors are detected and corrected before they propagate. More advanced ECC codes, such as **SEC-DED (Single Error Correction-Double Error Detection)**, can further improve reliability by detecting multiple-bit errors and correcting single-bit errors.
2. **Quantization-Aware Training:** As neural networks become quantized for efficiency, maintaining precision becomes critical. Techniques like quantization-aware training can ensure that models remain robust against the reduced precision and potential errors introduced by quantization.
3. **Data Type Optimization:** According to [17], selecting data types with just enough dynamic value range and precision can dramatically reduce error vulnerability. For instance, floating-point representations with larger dynamic ranges are more prone to error propagation than fixed-point alternatives.

The ongoing development of **reliable CNN systems** involves balancing performance, energy efficiency, and fault tolerance. As mentioned in [18], future research could focus on optimizing ECC techniques for CNN accelerators, combining them with selective hardening and detector systems

One common approach to mitigating these errors is through the use of error detection and correction codes, such as Hamming codes [19]. Hamming codes are capable of detecting and correcting single-bit errors in binary data, making them a suitable choice for enhancing the robustness of CNNs. In the context of neural networks, Hamming codes can be embedded into

the weights and biases of the network, providing a layer of protection against faults without requiring significant changes to the network architecture.

2.4 Existing Research on Bit-Level Error Correction in CNNs

The identification of insignificant bits within the parameters of CNNs has garnered increasing interest, as it presents opportunities for embedding error correction codes without significantly impacting model performance. Research by Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. (2015) [9] demonstrated that reducing the precision of weights and activations in CNNs can lead to certain bits becoming redundant. These redundant bits can then be repurposed to store additional information, such as error correction codes, without increasing the memory footprint. This approach not only minimizes memory usage but also enhances the network's fault tolerance.

Another study by Zhang, D., Yang, J., Ye, D., & Shi, Y. (2018) [10] explored the use of low-bit quantization in CNNs and found that even at reduced bit-widths, specific bits in the network's parameters can be modified to embed error detection codes without significantly degrading accuracy. This method is particularly beneficial for deploying neural networks in resource-constrained environments, where both accuracy and reliability are critical.

In more recent work, Ruiz, J. C., de Andrés, D., Saiz-Adalid, L. J., & Gracia-Morán, J. (2024) [8] introduced the concept of “Zero-Space In-Weight and In-Bias Protection for Floating-Point-based CNNs.” This study proposed a novel method for embedding fault-tolerant information directly into the floating-point representations used in CNNs. By analyzing the sensitivity of different bits in the network's parameters, the authors identified bits that could be utilized for error correction without compromising the network's accuracy or efficiency. This approach is particularly valuable for maintaining the reliability of CNNs in environments where memory resources are limited, and fault tolerance is essential.

Overall, the existing literature highlights a growing interest in enhancing the fault tolerance of CNNs through bit-level modifications, particularly in float32 representations. These studies provide a foundation for the development of new methodologies that leverage insignificant bits for error correction, as proposed in this thesis.

3. Methodology

This chapter outlines the methodology used in this thesis to enhance the fault tolerance of Convolutional Neural Networks (CNNs) that utilize 32-bit floating-point (float32) representations, without increasing their memory footprint. The approach involves analyzing the significance of individual bits within the floating-point representation of various CNN parameters and identifying opportunities to embed fault-tolerance mechanisms.

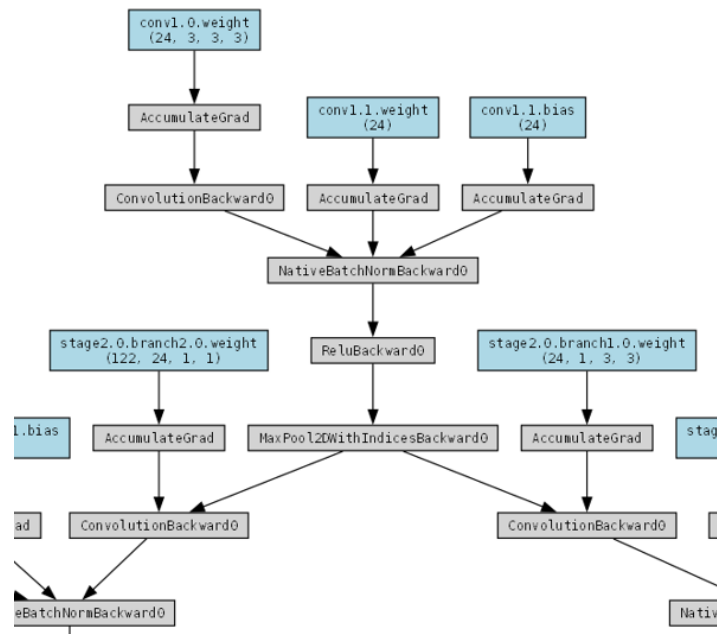
3.1 Overview of Neural Networks and Parameter Roles

Convolutional Neural Networks (CNNs) are composed of multiple layers, each with a specific function, such as convolution, pooling, or fully connected operations. The key parameters that drive the functionality of these layers include:

- **Weights:** These are the primary parameters learned during training, responsible for capturing the features in the input data. Weights are applied to the input during the forward pass to produce the feature maps.
- **Biases:** Biases are added to the output of the weighted sum in each neuron, allowing the network to shift the activation function and better model the data. They are also learned during the training process.
- **Running Mean and Running Variance:** In layers such as Batch Normalization, the running mean and variance are used to normalize the output of the previous layer. These parameters help the network maintain stability during training by ensuring that the activations are well-scaled.

Figure 2 illustrates the interaction between various layers in a ShuffleNet Convolutional Neural Network (CNN). The diagram shows how parameters such as weights and biases are utilized in different stages of the network. Each layer in the network, represented by boxes, interacts with others through forward and backward propagation. For instance, weights from the convolutional layers (*conv1.0.weight*, *stage2.0.branch2.0.weight*) undergo accumulation of gradients during the backward pass, which is essential for the learning process. Operations like *ConvolutionBackward0*, *ReluBackward0*, and *NativeBatchNormBackward0* depict how gradients are propagated back through the network, adjusting the weights and biases to minimize the loss function. This figure highlights the complexity of parameter interactions and the importance of each component in the overall functionality of CNN.

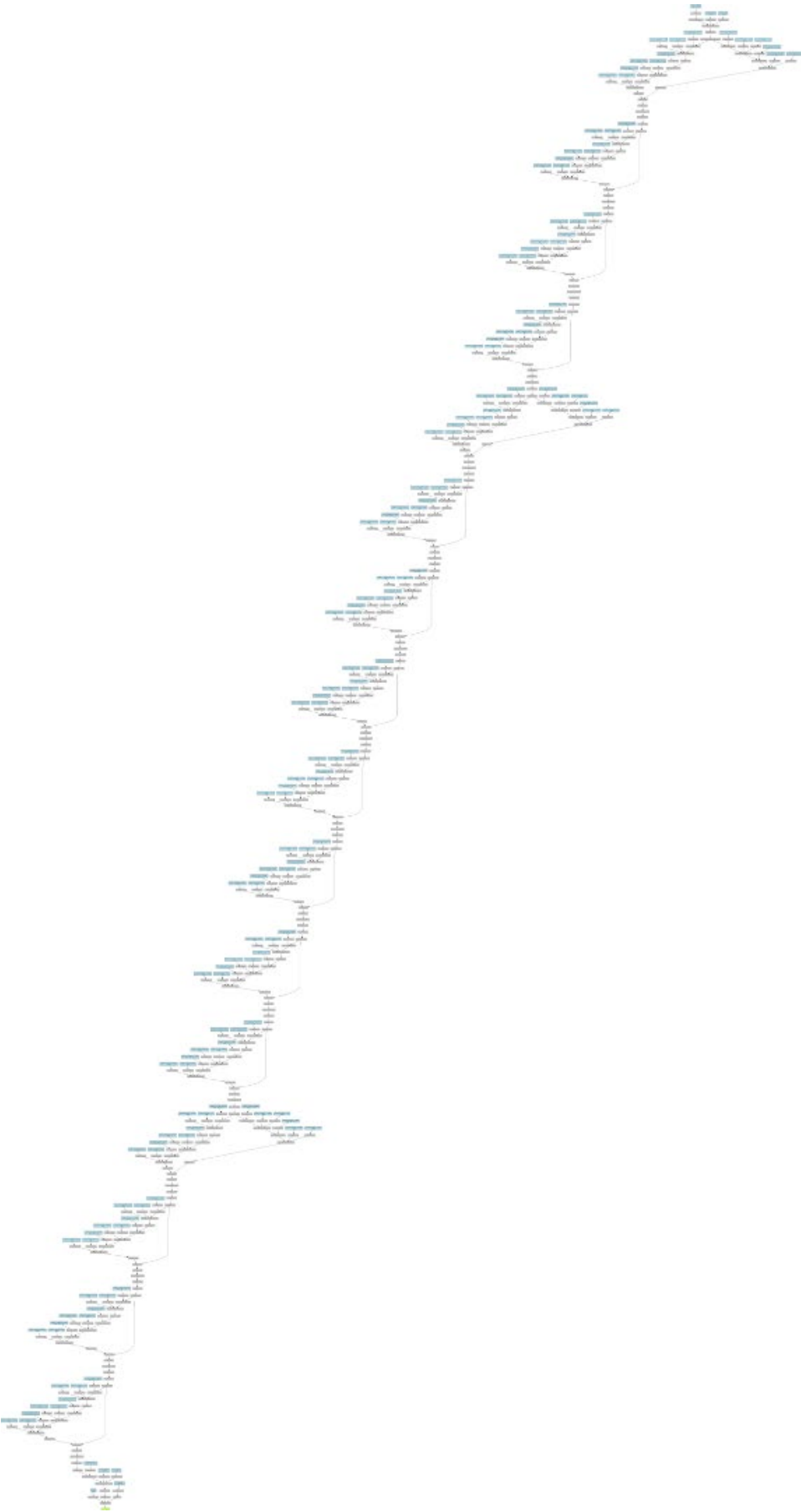
Figure 2. The interaction between various layers in a ShuffleNet CNN



The number of layers and parameters in ShuffleNet is extensive, making the full visualization complex and comprehensive. **Figure 3** provides a complete diagram of these interactions, demonstrating the intricate structure and the vast number of connections that exist within the network.

The values of these parameters typically vary across different layers and functions within the network. For example, weights might range widely depending on the features they represent, while biases tend to have smaller magnitudes.

Figure 3. The full visualization of a ShuffleNet CNN

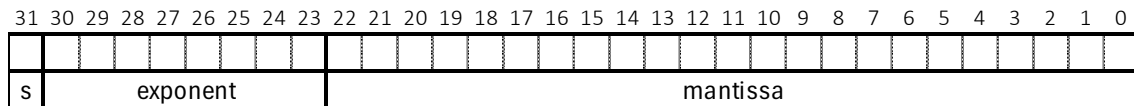


3.2 Representation of float32 According to IEEE 754

The 32-bit floating-point representation, commonly referred to as float32, is defined by the IEEE 754 standard [20]. **Figure 4** shows the three main parts of the 32-bit floating-point representation:

- Sign Bit (1 bit): Determines whether the number is positive (0) or negative (1).
- Exponent (8 bits): Represents the scale of the number by adjusting the position of the decimal point. It uses a biased representation, meaning that the exponent is stored as an unsigned integer with a bias (127 for float32) subtracted to obtain the actual exponent value.
- Mantissa or Significand (23 bits): Contains the significant digits of the number. In normalized form, the most significant bit (MSB) is implicitly 1 and not stored, allowing for a 24-bit precision.

Figure 4. The 32-bit floating-point representation

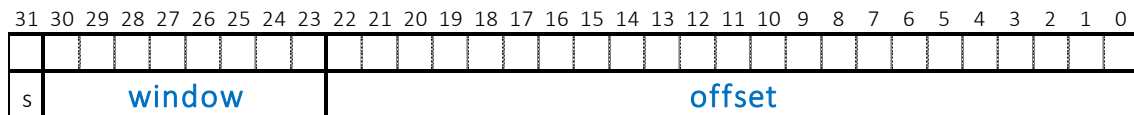


Thus, the final value of the number is determined by the product of the sign, the power of 2 defined by the exponent, and the value defined by the mantissa. **The equation (1)** allows for efficient representation of both very large and very small numbers within 32 bits.

$$value = (-1)^{sign} \times 2^{exponent-127} \times (1 + mantissa) \quad (1)$$

The exponent can be interpreted as a **window** indicating between which powers of two the number is located, for example, between [1,2[or [2,4[, and the mantissa as **an offset** within this window, dividing it into segments (see **Figure 5**).

Figure 5. Simplified 32-bit floating point representation.



3.3 Golden run

Python is the most widely used programming language for developing Convolutional Neural Networks (CNNs) due to its extensive support for machine learning libraries and ease of use. In this thesis, we work with Python along with its powerful libraries, particularly PyTorch [21], which provides robust tools for building and deploying deep learning models. To explain and validate our methodology, we utilize a pre-trained ShuffleNet model, sourced from the PyTorch library.

As part of the analysis, we perform a "golden run" by evaluating the network on a test set of 50,000 images. This large-scale test provides a baseline of the model's accuracy under normal conditions, without any bit-level alterations or faults introduced. The results from this golden run, including the network's accuracy and prediction consistency, serve as a critical reference point for comparing the impact of subsequent bit-level modifications and fault injections.

3.3.1 Main code blocks

The code includes the following main blocks:

1. Imports and Settings:

The Code Fragment 1 imports the necessary libraries, including PyTorch, torchvision, and PIL for image processing. It also imports custom settings such as the pre-trained CNN model, input labels, and various paths for input and output files from the settings file.

Code Fragment 1

```
from torchvision import transforms
from PIL import Image

import torch
import numpy as np
import sys
import os
import time

sys.path.append('../')
from settings import CNN_MODEL, MAX_IMAGES, INPUT_IMAGES_LABELS,
IMAGE_PREFIX, OUTPUT_DIR_GR, OUTPUT_GR
```

2. Print Configuration:

Configures the print settings for NumPy and PyTorch to ensure that large arrays can be fully displayed, and floating-point numbers are printed with high precision, making debugging and output analysis easier as shown at **Code Fragment 2**.

Code Fragment 2

```
np.set_printoptions(threshold=1000000, suppress=True)
torch.set_printoptions(threshold=sys.maxsize, precision=16,
sci_mode=False)
```

3. Loading Labels:

Reads the image labels from a file and stores them in a list. Each label is extracted and cleaned for use in evaluating the model's predictions (**Code Fragment 3**).

Code Fragment 3

```
with open(INPUT_IMAGES_LABELS) as f:
    labels = [line.strip().split(";", 1)[1] for line in f.readlines()]
```

4. Model Initialization:

Loads the pre-trained CNN model specified in the settings and sets it to evaluation mode. Evaluation mode ensures that certain layers like dropout and batch normalization behave appropriately during inference, providing consistent results (see **Code Fragment 4**).

Code Fragment 4

```
current_model = CNN_MODEL
# CNN_MODEL =
models.shufflenet_v2_x2_0(weights='ShuffleNet_V2_X2_0_Weights.DEFAULT')
current_model.eval()
```

5. Image Transformation Pipeline:

As shown in **Code Fragment 5**, the sequence of transformations applied to each input image before it is fed into the model includes resizing, cropping, converting to a tensor, and normalizing the pixel values to match the model's training conditions.

Code Fragment 5

```
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225]
    )
])
```

6. Image Processing and Inference:

The algorithm iterates over all images specified by `MAX_IMAGES` (50000), as shown in **Code Fragment 6**. For each image:

- The image is loaded, converted to RGB, and transformed according to the pipeline defined earlier.

- The transformed image is fed into the CNN to get the model's prediction.
- The predicted label is compared with the true label, and the result (including whether the prediction was correct) is written to the output file.

Code Fragment 6

```
for num_image in range(1, MAX_IMAGES+1):
    # Get image path
    image_path = IMAGE_PREFIX + format(str(num_image), "0>8") + ".JPEG"
    img = Image.open(image_path)
    img_rgb = img.convert("RGB")
    # Apply the required transformation
    img_t = transform(img_rgb)
    batch_t = torch.unsqueeze(img_t, 0)
    # Run the inference
    out = current_model(batch_t)
    # Get the selected category
    _, index = torch.max(out, 1)
    golden_run_output.write(f"{num_image};{index[0].item()};{labels[num_i
image - 1]};{index[0].item() == labels[num_image - 1]}\n")
```

7. Timing and Output File Management:

These blocks handle the preparation and management of the output file where the results of the golden run will be stored. The first part ensures that the output directory exists and creates it if necessary. Then, the output file is opened in write mode, and a header row is written to the file to structure the results (see **Code Fragment 7**).

Code Fragment 7

```
# Open the output file in write mode
if not os.path.exists(OUTPUT_DIR_GR):
    os.makedirs(OUTPUT_DIR_GR)
golden_run_output = open(OUTPUT_GR, 'w')

# Start timestamp
start = time.time()

...

# End timestamp
end = time.time()

# Close the output file
golden_run_output.close()

print("Execution time: " + str(end - start) + " seconds")
print(f"Output file: {OUTPUT_GR}")
```

Additionally, a timer is started to track the total execution time of an experiment. Once the entire process is completed, the timer is stopped, and the total execution time is printed to the console along with the location of the output file.

3.3.2 Output example

The results of the network's accuracy with either unmodified or adjusted parameters are summarized in the table. The algorithm generates a CSV output that records the impact of each bit modification on the CNN's accuracy. An example of this output is presented in **Table 1** and includes columns such as:

Table 1. An example of Golden Run output

| IMGID | PRED | LABEL | HIT |
|-------|------|-------|-------|
| 1 | 65 | 65 | TRUE |
| 2 | 795 | 970 | FALSE |
| 3 | 230 | 230 | TRUE |
| 4 | 809 | 809 | TRUE |
| ... | ... | ... | ... |

Here, IMGID represents the image ID, PRED is the predicted class, LABEL is the actual class label, and HIT indicates whether the prediction was correct or not.

3.3.3 Analysis of the Golden Run

The results from the golden run show that the ShuffleNet model achieved an accuracy of **76.19%** on the test set of 50,000 images (see **Table 2**). This means that the model correctly predicted the labels for 76.19% of the images, while the remaining 23.81% were misclassified.

Table 2. The Golden Run prediction result

| | TRUE | FALSE | TRUE | FALSE |
|------------|----------|--------|---------|---------|
| Golden Run | • 38,097 | 11,903 | 76.194% | 23.806% |

This accuracy serves as a baseline for the model's performance under normal conditions, without any modifications or faults injected into the parameters.

3.4 Identification of Significant Bits

The mantissa in float32 provides the detailed precision of the number, but not all 23 bits contribute equally to the represented value and, therefore, to the network's accuracy. The next step is to identify the significant bits—those that, when altered, lead to noticeable changes in the network's output.

- 1. Bit-Level Sensitivity Analysis:** For each layer's parameters (weights, biases, running means, and variances), we will systematically alter individual bits in the mantissa, starting from the least significant bit (LSB) to the most significant bit (MSB), and observe the impact on network accuracy.

2. Threshold Determination: Establishing a threshold below which changes to bits are considered insignificant, meaning they do not meaningfully degrade the network's performance. This threshold will be determined empirically based on the observed effects of bit alterations during the sensitivity analysis.

3. Insignificant Bits Identification: Based on the sensitivity analysis, identify the bits that can be altered or repurposed without substantially affecting network accuracy. These bits will be candidates for storing fault-tolerance information.

In this section, we describe the algorithm developed to identify significant bits within the 32-bit floating-point (float32) values used in CNN. The goal is to analyze how changes to individual bits in the mantissa affect the network's accuracy, thereby determining which bits are critical for maintaining accuracy and which can be altered without significant impact. This process is crucial for identifying potential bits that can be repurposed for fault-tolerance mechanisms without increasing memory usage.

3.4.1 Algorithm Overview

The algorithm operates by systematically injecting faults into specific bits of the mantissa across all parameters in the network (such as weights, biases, running means, and variances) and then measuring the impact of these modifications on the network's output accuracy. The key steps of the algorithm are as follows:

1. Setup and Initialization:

- The algorithm begins by setting up the environment, including loading the pre-trained CNN model and preparing the input data. The model is set to evaluation mode to ensure consistency in output during testing.
- A series of transformations is applied to the input images to prepare them for the CNN, ensuring they are resized, cropped, and normalized according to the model's requirements.

2. Fault Injection in Mantissa Bits:

- The algorithm iterates through each parameter tensor in the CNN. A tensor is a multi-dimensional array, similar to a matrix, which holds numerical data, which in this case represents the parameters (weights, biases, etc.) of the network. To simplify the process of manipulating individual values, the data in each tensor is flattened into a one-dimensional array.
- For each value in the tensor, the algorithm interprets the 32-bit float as an integer, allowing direct manipulation of specific bits.
- The algorithm then modifies a specific bit in the mantissa. Depending on whether the bit is originally set to 1 or 0, the algorithm either zeros out the less significant bits that follow or sets them all to 1. This process is used to study the impact of altering that particular bit and the subsequent less significant bits on the network's accuracy, rather than injecting a fault.

3. Inference and Accuracy Measurement:

- After modifying the bits in the model's parameters, the algorithm processes a set of labeled images through the modified CNN. For each image, the output prediction is compared to the ground truth label.
- The results, including the image ID, predicted class, actual label, and whether the prediction was correct, are recorded in a CSV file for further analysis.

4. Iteration Across All Bits:

- This process is repeated for each of the 32 bits in the mantissa, starting from the least significant bit (LSB) to the most significant bit (MSB). By iterating over all possible bit positions, the algorithm identifies which bits, when altered, lead to noticeable degradation in network accuracy.

5. Logging and Output:

- Throughout the process, detailed logs are maintained, capturing the status of each run and the time taken to complete the analysis for each bit. The final results are saved in a CSV file, providing a comprehensive view of the network's sensitivity to changes in specific bits of the mantissa.

3.4.2 Example Output

The algorithm generates a CSV output that records the impact of each bit modification on the CNN's performance. An example of this output is presented in **Table 3** and includes the same columns as a Golden run:

Table 3. An example of a modified CNN output

| IMGID | PRED | LABEL | HIT |
|-------|------|-------|-------|
| 1 | 65 | 65 | TRUE |
| 2 | 795 | 970 | FALSE |
| 3 | 230 | 230 | TRUE |
| 4 | 809 | 809 | TRUE |
| ... | ... | ... | ... |

Here, IMGID represents the image ID, PRED is the predicted class, LABEL is the actual class label, and HIT indicates whether the prediction was correct.

3.4.3 Analysis of Significant Bits

The results of the bit-level experiments are summarized in **Table 4**. The table shows the number of correct predictions (TRUE) and incorrect predictions (FALSE) for each bit in the mantissa after modifying that bit and all previous bits across all network parameters. Additionally, the percentage of correct (TRUE %) and incorrect (FALSE %) predictions are provided to highlight the impact of each bit modification on the network's accuracy.

Table 4. The Result of Analysis of Significant Bits

| BIT | TRUE | FALSE | TRUE | FALSE | |
|--------|-------|-------|--------|--------|----------|
| bit_31 | 50 | 49950 | 0.10% | 99.90% | sign |
| bit_30 | 50 | 49950 | 0.10% | 99.90% | |
| bit_29 | 50 | 49950 | 0.10% | 99.90% | |
| bit_28 | 50 | 49950 | 0.10% | 99.90% | exp |
| bit_27 | 50 | 49950 | 0.10% | 99.90% | |
| bit_26 | 50 | 49950 | 0.10% | 99.90% | |
| bit_25 | 50 | 49950 | 0.10% | 99.90% | |
| bit_24 | 50 | 49950 | 0.10% | 99.90% | |
| bit_23 | 50 | 49950 | 0.10% | 99.90% | |
| bit_22 | 50 | 49950 | 0.10% | 99.90% | |
| bit_21 | 50 | 49950 | 0.10% | 99.90% | |
| bit_20 | 133 | 49867 | 0.27% | 99.73% | mantissa |
| bit_19 | 11995 | 38005 | 23.99% | 76.01% | |
| bit_18 | 26508 | 23492 | 53.02% | 46.98% | |
| bit_17 | 35803 | 14197 | 71.61% | 28.39% | |
| bit_16 | 37629 | 12371 | 75.26% | 24.74% | |
| bit_15 | 37909 | 12091 | 75.82% | 24.18% | |
| bit_14 | 38108 | 11892 | 76.22% | 23.78% | |
| bit_13 | 38106 | 11894 | 76.21% | 23.79% | |
| bit_12 | 38097 | 11903 | 76.19% | 23.81% | |
| bit_11 | 38103 | 11897 | 76.21% | 23.79% | |
| bit_10 | 38101 | 11899 | 76.20% | 23.80% | |
| bit_9 | 38093 | 11907 | 76.19% | 23.81% | |
| bit_8 | 38095 | 11905 | 76.19% | 23.81% | |
| bit_7 | 38097 | 11903 | 76.19% | 23.81% | |
| bit_6 | 38097 | 11903 | 76.19% | 23.81% | |
| bit_5 | 38097 | 11903 | 76.19% | 23.81% | |
| bit_4 | 38097 | 11903 | 76.19% | 23.81% | |
| bit_3 | 38098 | 11902 | 76.20% | 23.80% | |
| bit_2 | 38098 | 11902 | 76.20% | 23.80% | |
| bit_1 | 38097 | 11903 | 76.19% | 23.81% | |
| bit_0 | 38097 | 11903 | 76.19% | 23.81% | |

Key Observations:

1. Bits 31 to 21 (Most Significant Bits):

- The network's accuracy is significantly affected when these bits are altered, becoming so low that its inference results are useless.. The TRUE % remains at a minimal 0.10%, while the FALSE % is consistently at 99.90%. This indicates that these bits in the mantissa are highly significant for maintaining the accuracy of the network. Any alteration in these bits leads to almost total degradation in performance.

2. Bits 20 to 17:

- There is a slight improvement in accuracy as we move from bit 20 to bit 17. For example, by bit 19, the TRUE % increases to 23.99%, indicating that some

resilience to errors begins to appear. However, the FALSE % remains dominant, highlighting that these bits are still largely significant but less so than the bits above them.

3. Bits 16 to 10:

- A noticeable shift occurs starting from bit 16, where the TRUE % jumps to over 75%. This trend continues down to bit 10, with the network maintaining a TRUE % of approximately 76.19%. This suggests that bits 16 through 10 are less critical, and the network can tolerate changes in these bits without substantial loss of accuracy.

4. Bits 9 to 0 (Least Significant Bits):

- In the range from bit 9 down to bit 0, the network shows a steady TRUE % of around 76.19% and a FALSE % of about 23.81%. These bits, particularly the least significant ones (bit 0 to bit 9), have the least impact on the network's overall accuracy, indicating that they are less significant. Alterations in these bits lead to some incorrect predictions, but the majority of the network's predictions remain accurate.

Conclusion:

The experiment demonstrates that the significance of mantissa bits in float32 representation diminishes as we move from the most significant bit (bit 31) to the least significant bit (bit 0). The network's accuracy is highly sensitive to changes in the higher-order bits (bits 31 to 21), where even minor alterations lead to a drastic reduction in accuracy. However, starting from bit 16, the network shows a much greater tolerance to bit alterations, with significant resilience observed in bits 16 to 0.

3.5 Analysis of the Sign Bit and Exponent Values

The exponent in float32 determines the scale of the number and can have a significant impact on the parameter values. However, like the mantissa, not all values in the exponent contribute equally to the network's accuracy.

1. Fixed Bit Analysis: Fix specific bits within the exponent and assess the impact on the accuracy of the CNN. This will help identify which bits are critical for maintaining numerical stability and which can be fixed without major consequences.

2. Component-Specific Analysis: Conduct this analysis separately for weights, biases, running means, and running variances to determine if the significance of exponent bits varies across these different components. For instance, running means and variances might tolerate more aggressive bit modifications than weights or biases.

3. Impact Assessment: Evaluate how the alterations in exponent bits affect the overall performance of the network and identify any trade-offs between accuracy and fault tolerance.

3.5.1 Preliminary Analysis of Bit Values

To begin the fixed bit analysis, we first conducted a thorough examination of the bit values for each parameter across all layers of the Convolutional Neural Network (CNN). The purpose of this analysis is to identify any bits in the 32-bit floating-point (float32) representation of the parameters that consistently hold the same value (either 0 or 1) across the entire network.

To achieve this, we developed a custom script that systematically iterates through all the parameters (such as weights, biases, running means, and variances) within the network. The script converts each parameter value from its float32 representation to its corresponding 32-bit integer format, allowing us to inspect and count the occurrences of 0s and 1s at each bit position.

The script accumulates these counts across all parameters, producing a detailed distribution of 0s and 1s for every bit position from 0 (the least significant bit) to 31 (the most significant bit). The results are then saved in a CSV file, providing a clear overview of which bits may be invariant.

3.5.2 Example Output

The algorithm generates a CSV output that records the value of each bit. An example of this output is presented in **Table 5** and includes columns such as:

Table 5. The Result of Analysis of Bit Values

| BIT | #0 | #1 |
|-----|---------|---------|
| 0 | 3712086 | 3715626 |
| 1 | 3713645 | 3714067 |
| 2 | 3715036 | 3712676 |
| 3 | 3713481 | 3714231 |
| 4 | 3713519 | 3714193 |
| ... | ... | ... |

In this file BIT represents the bit position (0 being the least significant bit and 31 the most significant bit), #0 is the count of times that bit was 0 across all parameter values, #1 is the count of times that bit was 1.

3.5.3 Analysis of Invariants Bits

The analysis of the bit-level distribution across all parameters in the network reveals significant insights into the potential for fixed bit optimization in the CNN. **Tables 6, 7** summarize the count and percentage of 0s and 1s for each bit in the 32-bit floating-point representation, segmented into the sign bit, exponent bits, and mantissa bits.

Table 6. The Result of Analysis of Bit Values (all tensors, weights)

| | | all param | 282 | tensors | 7427712 | weight | 113 | tensors | 7,376,138 |
|----------|---------|-----------|---------|---------|---------|---------|---------|---------|-----------|
| BIT | | | | | | #0 | #1 | #0 | #1 |
| sign | bit_31 | 3502522 | 3925190 | 47.15% | 52.85% | 3472343 | 3903795 | 47.08% | 52.92% |
| | bit_30 | 7412408 | 15304 | 99.79% | 0.21% | 7371121 | 5017 | 99.93% | 0.07% |
| exponent | bit_29 | 15384 | 7412328 | 0.21% | 99.79% | 5025 | 7371113 | 0.07% | 99.93% |
| | bit_28 | 24204 | 7403508 | 0.33% | 99.67% | 13838 | 7362300 | 0.19% | 99.81% |
| | bit_27 | 28069 | 7399643 | 0.38% | 99.62% | 15357 | 7360781 | 0.21% | 99.79% |
| | bit_26 | 1425016 | 6002696 | 19.19% | 80.81% | 1406111 | 5970027 | 19.06% | 80.94% |
| | bit_25 | 5911097 | 1516615 | 79.58% | 20.42% | 5884520 | 1491618 | 79.78% | 20.22% |
| | bit_24 | 3619056 | 3808656 | 48.72% | 51.28% | 3595716 | 3780422 | 48.75% | 51.25% |
| | bit_23 | 3742675 | 3685037 | 50.39% | 49.61% | 3717564 | 3658574 | 50.40% | 49.60% |
| | bit_22 | 4342547 | 3085165 | 58.46% | 41.54% | 4312747 | 3063391 | 58.47% | 41.53% |
| | bit_21 | 4044562 | 3383150 | 54.45% | 45.55% | 4016451 | 3359687 | 54.45% | 45.55% |
| | bit_20 | 3878117 | 3549595 | 52.21% | 47.79% | 3851142 | 3524996 | 52.21% | 47.79% |
| mantissa | bit_19 | 3797640 | 3630072 | 51.13% | 48.87% | 3771289 | 3604849 | 51.13% | 48.87% |
| | bit_18 | 3754779 | 3672933 | 50.55% | 49.45% | 3728893 | 3647245 | 50.55% | 49.45% |
| | bit_17 | 3733555 | 3694157 | 50.27% | 49.73% | 3707791 | 3668347 | 50.27% | 49.73% |
| | bit_16 | 3723735 | 3703977 | 50.13% | 49.87% | 3698051 | 3678087 | 50.14% | 49.86% |
| | bit_15 | 3719163 | 3708549 | 50.07% | 49.93% | 3693305 | 3682833 | 50.07% | 49.93% |
| | bit_14 | 3717680 | 3710032 | 50.05% | 49.95% | 3691765 | 3684373 | 50.05% | 49.95% |
| | bit_13 | 3714230 | 3713482 | 50.01% | 49.99% | 3688544 | 3687594 | 50.01% | 49.99% |
| | bit_12 | 3714703 | 3713009 | 50.01% | 49.99% | 3688995 | 3687143 | 50.01% | 49.99% |
| | bit_11 | 3715662 | 3712050 | 50.02% | 49.98% | 3689881 | 3686257 | 50.02% | 49.98% |
| | bit_10 | 3715302 | 3712410 | 50.02% | 49.98% | 3689443 | 3686695 | 50.02% | 49.98% |
| | bit_9 | 3713186 | 3714526 | 49.99% | 50.01% | 3687280 | 3688858 | 49.99% | 50.01% |
| | bit_8 | 3715241 | 3712471 | 50.02% | 49.98% | 3689551 | 3686587 | 50.02% | 49.98% |
| | bit_7 | 3714074 | 3713638 | 50.00% | 50.00% | 3688062 | 3688076 | 50.00% | 50.00% |
| | bit_6 | 3714539 | 3713173 | 50.01% | 49.99% | 3688824 | 3687314 | 50.01% | 49.99% |
| | bit_5 | 3713336 | 3714376 | 49.99% | 50.01% | 3687374 | 3688764 | 49.99% | 50.01% |
| | bit_4 | 3713519 | 3714193 | 50.00% | 50.00% | 3687636 | 3688502 | 49.99% | 50.01% |
| | bit_3 | 3713481 | 3714231 | 49.99% | 50.01% | 3687546 | 3688592 | 49.99% | 50.01% |
| | bit_2 | 3715036 | 3712676 | 50.02% | 49.98% | 3689450 | 3686688 | 50.02% | 49.98% |
| bit_1 | 3713645 | 3714067 | 50.00% | 50.00% | 3687907 | 3688231 | 50.00% | 50.00% | |
| bit_0 | 3712086 | 3715626 | 49.98% | 50.02% | 3686440 | 3689698 | 49.98% | 50.02% | |

Protecting FP-based CNNs Against Faults without Increasing Their Memory Footprint

Table 7. The Result of Analysis of Bit Values (biases, running means, running vars)

| | BIT | bias | | | | mean | | | | var | | | |
|----------|--------|-------|--------|--------|--------|-------|--------|--------|--------|-------|--------|--------|--------|
| | | #0 | #1 | #0 | #1 | #0 | #1 | #0 | #1 | #0 | #1 | #0 | #1 |
| sign | bit 31 | 5958 | 11900 | 33.36% | 66.64% | 7363 | 9495 | 43.68% | 56.32% | 16858 | 0 | 100% | 0.00% |
| | bit 30 | 14573 | 3285 | 81.60% | 18.40% | 16138 | 720 | 95.73% | 4.27% | 10576 | 6282 | 62.74% | 37.26% |
| exponent | bit 29 | 3285 | 14573 | 18.40% | 81.60% | 749 | 16109 | 4.44% | 95.56% | 6325 | 10533 | 37.52% | 62.48% |
| | bit 28 | 3285 | 14573 | 18.40% | 81.60% | 763 | 16095 | 4.53% | 95.47% | 6318 | 10540 | 37.48% | 62.52% |
| | bit 27 | 4576 | 13282 | 25.62% | 74.38% | 1816 | 15042 | 10.77% | 89.23% | 6320 | 10538 | 37.49% | 62.51% |
| | bit 26 | 7388 | 10470 | 41.37% | 58.63% | 5149 | 11709 | 30.54% | 69.46% | 6368 | 10490 | 37.77% | 62.23% |
| | bit 25 | 8979 | 8879 | 50.28% | 49.72% | 7718 | 9140 | 45.78% | 54.22% | 9880 | 6978 | 58.61% | 41.39% |
| | bit 24 | 7971 | 9887 | 44.64% | 55.36% | 7072 | 9786 | 41.95% | 58.05% | 8297 | 8561 | 49.22% | 50.78% |
| | bit 23 | 8268 | 9590 | 46.30% | 53.70% | 8347 | 8511 | 49.51% | 50.49% | 8496 | 8362 | 50.40% | 49.60% |
| | bit 22 | 9983 | 7875 | 55.90% | 44.10% | 9858 | 7000 | 58.48% | 41.52% | 9959 | 6899 | 59.08% | 40.92% |
| | bit 21 | 9649 | 8209 | 54.03% | 45.97% | 9194 | 7664 | 54.54% | 45.46% | 9268 | 7590 | 54.98% | 45.02% |
| | bit 20 | 9423 | 8435 | 52.77% | 47.23% | 8798 | 8060 | 52.19% | 47.81% | 8754 | 8104 | 51.93% | 48.07% |
| mantissa | bit 19 | 9173 | 8685 | 51.37% | 48.63% | 8604 | 8254 | 51.04% | 48.96% | 8574 | 8284 | 50.86% | 49.14% |
| | bit 18 | 8938 | 8920 | 50.05% | 49.95% | 8460 | 8398 | 50.18% | 49.82% | 8488 | 8370 | 50.35% | 49.65% |
| | bit 17 | 9013 | 8845 | 50.47% | 49.53% | 8363 | 8495 | 49.61% | 50.39% | 8388 | 8470 | 49.76% | 50.24% |
| | bit 16 | 8930 | 8928 | 50.01% | 49.99% | 8377 | 8481 | 49.69% | 50.31% | 8377 | 8481 | 49.69% | 50.31% |
| | bit 15 | 8980 | 8878 | 50.29% | 49.71% | 8423 | 8435 | 49.96% | 50.04% | 8455 | 8403 | 50.15% | 49.85% |
| | bit 14 | 8912 | 8946 | 49.90% | 50.10% | 8548 | 8310 | 50.71% | 49.29% | 8455 | 8403 | 50.15% | 49.85% |
| | bit 13 | 8857 | 9001 | 49.60% | 50.40% | 8332 | 8526 | 49.42% | 50.58% | 8497 | 8361 | 50.40% | 49.60% |
| | bit 12 | 8857 | 9001 | 49.60% | 50.40% | 8421 | 8437 | 49.95% | 50.05% | 8430 | 8428 | 50.01% | 49.99% |
| | bit 11 | 8929 | 8929 | 50.00% | 50.00% | 8374 | 8484 | 49.67% | 50.33% | 8478 | 8380 | 50.29% | 49.71% |
| | bit 10 | 8991 | 8867 | 50.35% | 49.65% | 8406 | 8452 | 49.86% | 50.14% | 8462 | 8396 | 50.20% | 49.80% |
| bit 9 | 8989 | 8869 | 50.34% | 49.66% | 8492 | 8366 | 50.37% | 49.63% | 8425 | 8433 | 49.98% | 50.02% | |
| bit 8 | 8971 | 8887 | 50.24% | 49.76% | 8447 | 8411 | 50.11% | 49.89% | 8272 | 8586 | 49.07% | 50.93% | |
| bit 7 | 9006 | 8852 | 50.43% | 49.57% | 8495 | 8363 | 50.39% | 49.61% | 8511 | 8347 | 50.49% | 49.51% | |
| bit 6 | 8897 | 8961 | 49.82% | 50.18% | 8406 | 8452 | 49.86% | 50.14% | 8412 | 8446 | 49.90% | 50.10% | |
| bit 5 | 9064 | 8794 | 50.76% | 49.24% | 8404 | 8454 | 49.85% | 50.15% | 8494 | 8364 | 50.39% | 49.61% | |
| bit 4 | 9036 | 8822 | 50.60% | 49.40% | 8468 | 8390 | 50.23% | 49.77% | 8379 | 8479 | 49.70% | 50.30% | |
| bit 3 | 9030 | 8828 | 50.57% | 49.43% | 8349 | 8509 | 49.53% | 50.47% | 8556 | 8302 | 50.75% | 49.25% | |
| bit 2 | 8886 | 8972 | 49.76% | 50.24% | 8393 | 8465 | 49.79% | 50.21% | 8307 | 8551 | 49.28% | 50.72% | |
| bit 1 | 8928 | 8930 | 49.99% | 50.01% | 8414 | 8444 | 49.91% | 50.09% | 8396 | 8462 | 49.80% | 50.20% | |
| bit 0 | 8804 | 9054 | 49.30% | 50.70% | 8377 | 8481 | 49.69% | 50.31% | 8465 | 8393 | 50.21% | 49.79% | |

Key Observations:

1. Sign Bit (Bit 31):

- The sign bit exhibits a relatively balanced distribution across different parameter types in the network, but this balance varies depending on the type of parameter. When considering all parameters together, approximately 47.15% of values are 0 and 52.85% are 1, indicating a substantial number of both positive and negative values.
- However, this distribution is heavily influenced by the weights, which show a similar balance with 47.08% of values being 0 and 52.92% being 1. In contrast, other parameter types like biases and running means display a different distribution: biases have 33.36% of values as 0 and 66.64% as 1, while running means show 43.68% of values as 0 and 56.32% as 1. Notably, the running

variances are entirely positive, with 100% of values being 0. These variations suggest that while the sign bit is not invariant for weights, biases, or running means, it could potentially be fixed for running variances without distorting the parameter values.

2. Exponent Bits (Bits 30 to 23):

- **Bits 30 and 29:** These bits are highly invariant, with bit 30 being 0 in 99.79% of cases and bit 29 being 1 in 99.79% of cases. This suggests that these bits consistently take on the same value across most parameters, indicating a potential opportunity to fix these bits without major impact on the network's performance.
- **Bits 28 to 23:** While there is some variation in these bits, certain bits like 28 (99.67% being 1) and 26 (79.58% being 1) show strong tendencies towards a particular value. However, bits 25 and 24 show a more balanced distribution, making them less likely candidates for fixing.

3. Mantissa Bits (Bits 22 to 0):

- Although the distribution of 0s and 1s in the mantissa bits is approximately even, with most bits hovering around a 50/50 split, not all of these bits significantly impact the network's accuracy. As noted earlier in the "Identification of Significant Bits in the Mantissa" section [3.4], some of these bits can be altered without substantially affecting the precision of the parameter values. Therefore, fixing any of these bits would change the behavior of the network. However, part of these bits is not relevant for that behavior, as stated before.

Conclusion:

The analysis suggests that while the mantissa bits are highly variable and critical for preserving precision, certain exponent bits, particularly bits 30 and 29, are highly invariant and could potentially be fixed without significantly impacting the network's performance.

3.6 Analysis of Parameter Values Across Network Layers

To understand how the bits in float32 parameters contribute to the performance of a CNN, we will perform a detailed analysis of the parameter values across different layers.

The analysis will involve the following steps:

1. **Data Collection:** Extracting the parameter values (weights, biases, running means, and running variances) from each layer of ShuffleNet after training on a benchmark dataset.
2. **Statistical Analysis:** Analyzing the distribution and range of these values to understand their general characteristics. This includes examining the average magnitudes, variances, and any outliers that may indicate special cases.

3. Pattern Identification: Identifying any patterns or correlations in the parameter values that might suggest certain bits in the mantissa or exponent are consistently less significant or vary less than others.

3.6.1 Exponent Distribution Analysis

To facilitate the analysis of parameter values across different layers of the ShuffleNet model, a custom script was developed. This script systematically iterates through all tensors in each layer of the network, extracting relevant parameter values such as weights, biases, running means, and running variances. The script processes these tensors and outputs the results in a structured format, specifically as a CSV file, which allows for easy examination and further analysis, represented in **Table 8**.

This structured data collection ensures that all relevant details about the network's parameters are captured, providing a solid foundation for subsequent analyses, such as identifying insignificant bits in the mantissa and analyzing the impact of exponent bits on the network's performance.

Table 8. Distribution of parameter values in ShuffleNet CNN

| Exponent | Power of '2' | Value Range | Quantity | | | | |
|----------|---------------------------------|--------------|-----------|-----------|--------|--------------|-------------|
| | | | total | weight | bias | running_mean | running_var |
| | | Total: | 7,427,712 | 7,376,138 | 17,858 | 16,858 | 16,858 |
| 00000000 | value less than 2^{-126} or 0 | | 58 | 0 | 0 | 29 | 29 |
| 00000001 | -126 | -126 to -125 | 0 | 0 | 0 | 0 | 0 |
| 00000010 | -125 | -125 to -124 | 0 | 0 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 01000001 | -62 | -62 to -61 | 11 | 11 | 0 | 0 | 0 |
| 01000010 | -61 | -61 to -60 | 24 | 24 | 0 | 0 | 0 |
| 01000011 | -60 | -60 to -59 | 49 | 49 | 0 | 0 | 0 |
| ... | ... | ... | ... | ... | ... | ... | ... |
| 01101110 | -17 | -17 to -16 | 1,841 | 1,514 | 198 | 129 | 0 |
| 01101111 | -16 | -16 to -15 | 3,652 | 2,981 | 388 | 283 | 0 |
| 01110000 | -15 | -15 to -14 | 7,059 | 5,825 | 761 | 473 | 0 |
| 01110001 | -14 | -14 to -13 | 13,442 | 11,230 | 1,284 | 928 | 0 |
| 01110010 | -13 | -13 to -12 | 25,619 | 22,948 | 1,343 | 1,328 | 0 |
| 01110011 | -12 | -12 to -11 | 46,235 | 45,037 | 416 | 782 | 0 |
| 01110100 | -11 | -11 to -10 | 91,006 | 90,683 | 62 | 257 | 4 |
| 01110101 | -10 | -10 to -9 | 179,721 | 179,349 | 39 | 330 | 3 |
| 01110110 | -9 | -9 to -8 | 354,390 | 354,147 | 56 | 181 | 6 |
| 01110111 | -8 | -8 to -7 | 688,424 | 688,191 | 112 | 86 | 35 |
| 01111000 | -7 | -7 to -6 | 1,255,583 | 1,254,929 | 177 | 154 | 323 |
| 01111001 | -6 | -6 to -5 | 1,892,608 | 1,891,034 | 358 | 287 | 929 |
| 01111010 | -5 | -5 to -4 | 1,844,598 | 1,841,987 | 483 | 846 | 1,282 |
| 01111011 | -4 | -4 to -3 | 805,732 | 802,318 | 408 | 1,820 | 1,186 |
| 01111100 | -3 | -3 to -2 | 135,691 | 132,522 | 668 | 1,812 | 689 |
| 01111101 | -2 | -2 to -1 | 23,474 | 19,140 | 1,343 | 1,792 | 1,199 |

| | | | | | | | |
|----------|----|---------|--------|--------|-------|-------|-------|
| 01111110 | -1 | -1 to 0 | 12,627 | 4,593 | 2,707 | 2,239 | 3,088 |
| 01111111 | 0 | 0 to 1 | 18,186 | 11,619 | 3,065 | 1,718 | 1,784 |
| 10000000 | 1 | 1 to 2 | 6,834 | 3,088 | 1,136 | 539 | 2,071 |
| 10000001 | 2 | 2 to 3 | 6,716 | 1,927 | 1,779 | 104 | 2,906 |
| 10000010 | 3 | 3 to 4 | 1,314 | 2 | 369 | 50 | 893 |
| 10000011 | 4 | 4 to 5 | 274 | 0 | 1 | 23 | 250 |
| 10000100 | 5 | 5 to 6 | 84 | 0 | 0 | 3 | 81 |
| 10000101 | 6 | 6 to 7 | 55 | 0 | 0 | 1 | 54 |
| 10000110 | 7 | 7 to 8 | 25 | 0 | 0 | 0 | 25 |
| 10000111 | 8 | 8 to 9 | 2 | 0 | 0 | 0 | 2 |
| 10001000 | 9 | 9 to 10 | 0 | 0 | 0 | 0 | 0 |

As part of the ongoing analysis of parameter values across the layers of the ShuffleNet model, I conducted a detailed examination of the exponent values within the 32-bit floating-point (float32) representation of all network parameters. This analysis aimed to categorize the parameters based on the value of their exponent, which significantly influences the magnitude of the floating-point numbers.

Key Findings:

- **Total Parameters Analyzed:** 7,427,712
- **Exponent Categories:**
 1. **Exponents Less Than 01111000₂:**
 - **Number of parameters:** 1,423,909 (19.17% of total)
 - **Characteristics:** These parameters correspond to very small values, typically close to zero.
 - **Important Consideration:** We cannot simply fix the exponent bits in this group without taking additional steps. If we were to fix the bits directly, say at 01111000₂, and retain the original mantissa and sign bits, it could drastically alter the parameter's magnitude. For instance, fixing these bits at a value like 01001100₂ (which corresponds to 2^{-51}) would instead produce 01111100₂ (which corresponds to 2^{-3}), a significant and unintended increase in magnitude.
 - **Algorithm Adjustment:** To address this, the algorithm assumes that not only are the bits fixed, but all lower bits in the exponent are also zeroed out. Essentially, we take all parameters with exponents below 01111000₂ and round them up to 01111000₂, while simultaneously setting all lower bits of the exponent to zero. This approach ensures that the values are rounded rather than increased, maintaining the intended small magnitude of the parameters.
 2. **Exponents Greater Than or Equal to 01111000₂ and Less Than 10000000₂:**
 - **Number of parameters:** 5,988,499 (80.62% of total).

- **Characteristics:** This group represents the majority of the parameters, which have moderate values that are well-distributed across the network. These parameters are central to the network's operations, as they typically encode the most critical information.
- **Strategy:** For this group, we explore fixing bits 26 to 30 in the exponent to simplify the network's calculations while still preserving the core functionality. The relative stability and concentration of values within this range suggest that such modifications may be feasible without substantial loss of precision or accuracy.

3. Exponents Greater Than 1000000_2 :

- Number of parameters: 15,304 (0.21% of total).
- **Characteristics:** Parameters in this group have large magnitudes, which could correspond to critical operations or features within the network.
- **Approach:** While these parameters are less common, they are potentially more impactful when altered. Careful consideration must be given to any modifications in this range to avoid unintended consequences on the network's performance.

Hypothesis for Fixed Bit Analysis:

Given the distribution of exponent values, I hypothesize that we might be able to fix certain exponent bits—specifically bits 26 to 30—without significantly impacting the network's performance. The rationale is based on the observation that a considerable number of parameters fall within the middle range of exponent values (01111000_2 to 10000000_2). Fixing these bits could simplify the network's computations while maintaining accuracy.

3.6.2 Parameter Categorization and Exponent Range Analysis

To validate previous hypothesis, I developed a script that categorizes these parameters further into weights, biases, running_mean, and running_var, and counts the occurrences of each category within the defined exponent ranges. The goal is to understand how the network's parameters are distributed across these categories, which will guide us in determining the potential impact of fixing specific bits in the exponent. We can see the result of the distribution in **Table 9**.

Table 9. Distribution of parameter values in ShuffleNet CNN

| | Total | % | WEIGHT | BIAS | MEAN | VAR |
|------------------------------------|------------------|--------|------------------|---------------|---------------|---------------|
| ... < 01111000_2 | 1,423,909 | 19.17% | 1,412,979 | 5,364 | 5,470 | 96 |
| 01111000_2 <= ... < 10000000_2 | 5,988,499 | 80.62% | 5,958,142 | 9,209 | 10,668 | 10,480 |
| 10000000_2 <= ... | 15,304 | 0.21% | 5,017 | 3,285 | 720 | 6,282 |
| Total | 7,427,712 | | 7,376,138 | 17,858 | 16,858 | 16,858 |

To further refine the analysis, we can attempt to categorize the network layers based on whether their parameters fall predominantly into the group with exponents less than or equal to

0111111₂, or greater than 0111111₂. This involves evaluating each of the network's parameter sets (of which there might be 282, for instance) to determine whether they comply with the “exponent \leq 0111111₂” rule.

It is likely, based on previous observations from other networks we have studied, that the vast majority of these parameter sets will fall into the group that complies with this rule. Identifying layers where all parameters either do or do not meet this criterion is crucial because it allows us to apply targeted bit-fixing strategies that are appropriate for the specific distribution of values within each layer.

For example, layers where all parameters have exponents less than or equal to 0111111₂ might be candidates for applying bit-fixing strategies that would simplify computations without affecting the overall performance. Conversely, layers with exponents predominantly greater than 0111111₂ might require a different approach to maintain the necessary precision and network functionality.

Table 10. Distribution of parameter values in ShuffleNet CNN

| | Total | | all parameters in layer have an exponent: | | | | mix | |
|--------------|-------------|------------------|---|------------------|------------------------|--------------|-------------|---------------|
| | | | <1000000 ₂ | | >=1000000 ₂ | | | |
| | # of layers | # of param | # of layers | # of param | # of layers | # of param | # of layers | # of param |
| weight | 113 | 7,376,138 | 61 | 7,356,802 | 1 | 2,048 | 51 | 17,288 |
| bias | 57 | 17,858 | 22 | 6,880 | 1 | 2,048 | 34 | 8,930 |
| running_mean | 56 | 16,858 | 33 | 9,296 | 0 | 0 | 23 | 7,562 |
| running_var | 56 | 16,858 | 13 | 4,148 | 10 | 2,928 | 33 | 9,782 |
| Total | 282 | 7,427,712 | 129 | 7,377,126 | 12 | 7,024 | 141 | 43,562 |

Based on the analysis presented in **Table 10**, we observe that in nearly half of all layers (129 + 12 layers), which account for 7,384,150 parameters (or 99.41% of the total), we can potentially fix 5 bits (from 26 to 30) in the exponent. This presents a significant opportunity to repurpose these bits for error-correction codes, thereby enhancing fault tolerance while preserving memory efficiency.

Specifically:

- **Layers where all parameters have exponents less than 1000000₂:** In 129 layers, comprising 7,377,126 parameters, fixing these bits is feasible as the exponents fall within a manageable range. This allows us to maintain the precision required for accurate network performance while simplifying the bit structure.
- **Layers where all parameters have exponents greater than or equal to 1000000₂:** In 12 layers, comprising 7,024 parameters in this particular network, fixing these bits is also possible. Although these layers contain larger values, the controlled fixing of bits ensures that the network's functionality remains intact. However, it is important to note that this result is specific to this network and its training process. In other networks, the ability to fix exponent bits will depend on the maximum exponent values present. If there are exponent values greater than 10000111₂, rounding could significantly reduce the represented value, potentially altering the inference process.

Furthermore, when combined with the Least Significant Bits (LSBs) of the mantissa (in the case of ShuffleNet, we identified 10 such LSBs), supplementing these with the 5 fixed bits from the exponent allows us to create a more robust protection mechanism. This combination can significantly improve the network's resilience against errors while maintaining overall performance and memory efficiency.

As a potential area for further research, this bit-fixing strategy could be explored in the context of quantized networks (using 16-bit or 8-bit precision). In such cases, focusing solely on bit-fixing might yield similar benefits, providing robust fault tolerance in even more memory-constrained environments.

3.7 Rounding Experiments on Exponent Bits

In this section, we outline the experiments designed to evaluate the impact of rounding value of the exponent on the accuracy of the ShuffleNet model. Given our previous analysis, we have determined that fixing bits 26-30 in the exponent should not affect the prediction accuracy for parameters in Group 2 (with exponents between 01111000_2 and 10000000_2) and Group 3 (with exponents greater than 10000000_2). This is because those bits do not alter the actual value of the parameters in these groups (keeping in mind that other networks may need to do additional Group 3 research).

3.7.1 Experimental Goal

The primary goal of our experiments is to iteratively determine how rounding the exponent values in Group 1 impacts the overall accuracy of the network. Instead of a single step, we will conduct a series of experiments starting from “pure” invariants. For instance, if all weights have exponents in the form “ $011xxxx_2$ ”, we begin with 3 invariant bits without any rounding, ensuring no loss in precision. The next step involves forcing all exponents into the form “ $0111xxx_2$ ”, where some rounding begins to occur. If the variation in accuracy remains within acceptable limits, we consider these 4 bits as invariants and proceed to the next step, forcing exponents to “ $01111xxx_2$ ”, and so on. This iterative process continues until we identify the maximum number of bits that can be fixed without significantly compromising the network’s ability to correctly classify images.

3.7.2 Experimental Procedure

1. Baseline Accuracy Measurement:

- Before any modifications, we will measure and record the baseline accuracy of ShuffleNet on a standard test set. This provides a reference point to compare the effects of our rounding technique.

2. Parameter Modification:

- We will iteratively modify the parameters in Group 1 by rounding their exponent values progressively higher values, starting from 01100000_2 and moving towards 01111000_2 . This adjustment will be applied to all relevant parameters across the network's layers during each iteration (see **Code Fragment 8**).

Code Fragment 8

```
exponents_list = [0b01100000, 0b01110000, 0b01111000, 0b01111100]

for new_exponent in exponents_list:
    ...
    test_run(new_exponent)

def test_run(new_exponent):
    ...
    current_model = deepcopy(CNN_MODEL)
    ...
    for key in dict.keys():
        ...
        for i in range(len(flat)):
            # Get the float as an integer
            int_value = flat[i].view(torch.int)
            int_bits = np.frombuffer(int_value.numpy().tobytes(),
dtype=np.uint32)[0]
            # Current exponent and mantissa
            # 10000000000000000000000000000000 = 0x80000000
            sign_bit = (int_bits & 0x80000000) >> 31
            # 01111111100000000000000000000000 = 0x7F800000
            exponent = (int_bits & 0x7F800000) >> 23
            mantissa = int_bits & 0x007FFFFF

            # Check exponent:
            if exponent < new_exponent:
                new_exponent_bits = new_exponent
                mantissa = 0
                tensors_modified_exp += 1
            else:
                new_exponent_bits = exponent

            # Assembling a new value
            new_int_bits = (sign_bit << 31) | (new_exponent_bits << 23) |
mantissa
            new_int_value =
torch.tensor(np.frombuffer(np.uint32(new_int_bits).tobytes(),
dtype=np.int32)[0], dtype=torch.int32)

            # Convert back to float32
            flat[i] = new_int_value.view(torch.float32)

        # Reshape the flattened tensor
        dict[key] = flat.view(shape)
    ...
```

3. Accuracy Evaluation:

- After applying the rounding modifications, we will re-evaluate the network's accuracy using the same test set. The objective is to determine if there is any significant drop in accuracy performance compared to the baseline.

4. Analysis of Results:

- The results will be analyzed to assess the impact of the iterative modifications. Starting with smaller exponent values, we progressively rounded the exponent values in Group 1 parameters to increasingly higher values, as shown in **Table 11**. If the accuracy remains within an acceptable range after each iteration, this would validate our hypothesis that such modifications can be made without adversely affecting the network's accuracy. Conversely, if there is a noticeable drop in accuracy, this would suggest that even small changes in the exponent can have a significant impact on the network's predictions.

Table 11. Comparison of network accuracy with modified parameters and Golden Run

| run | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|-----------------------|-----------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| Golden Run | | | | | | | 38,097 | 11,903 | 76.194% | 23.806% |
| rounding to 2^{-31} | 01100000 ₂ | 282 | 73 | 7,427,712 | 8,917 | 0.12% | 38,097 | 11,903 | 76.194% | 23.806% |
| rounding to 2^{-15} | 01110000 ₂ | | 132 | | 18,013 | 0.24% | 38,100 | 11,900 | 76.200% | 23.800% |
| rounding to 2^{-7} | 01111000 ₂ | | 177 | | 1,423,909 | 19.17% | 38,009 | 11,991 | 76.018% | 23.982% |
| rounding to 2^{-3} | 01111100 ₂ | | 206 | | 7,222,430 | 97.24% | 41 | 49,959 | 0.082% | 99.918% |

Table 11 summarizes the results of our iterative experiments comparing the baseline (“Golden Run”) accuracy of the ShuffleNet model with the performance after modifying the exponent values in Group 1 parameters. Each row in the table represents a different rounding iteration, showing how the network's accuracy changes as the exponent values are rounded to different fixed points, culminating in the final value of 01111000₂.

3.7.3 Key Observations:

• **Golden Run:**

- **Baseline Accuracy:** The network achieved a baseline accuracy of **76.194%** on the test set, with **38,097** correct predictions (TRUE) and **11,903** incorrect predictions (FALSE).

• **Rounding Experiment:**

- Rounding to 01100000₂ (2^{-31})

- **Modified Tensors and Values:** Out of the 282 tensors in the network, 73 were modified, affecting a total of 8,918 values.
 - **Post-Modification Accuracy:** The network's accuracy remained unchanged at 76.194%, with 38,097 correct predictions and 11,903 incorrect predictions.
 - **Accuracy Impact:** The minimal modification of only 8,917 values led to no observable impact on the network's accuracy, indicating that rounding to 0110000_2 has negligible effects.
- Rounding to 01110000_2 (2^{-15})
 - **Modified Tensors and Values:** 132 tensors were modified, affecting a total of 18,013 values.
 - **Post-Modification Accuracy:** The network's accuracy remained virtually unchanged at 76.200%, with 38,100 correct predictions and 11,900 incorrect predictions.
 - **Accuracy Impact:** Despite modifying twice as many values as in the previous experiment, the accuracy remained stable, suggesting that rounding to 01110000_2 does not significantly impact the network's performance.
- Rounding to 01111000_2 (2^{-7})
 - **Modified Tensors and Values:** 177 tensors were modified, affecting a total of 1,423,909 values.
 - **Post-Modification Accuracy:** The network's accuracy showed a slight decrease to 76.018%, with 38,009 correct predictions and 11,991 incorrect predictions.
 - **Accuracy Impact:** Although the number of modified values increased dramatically, the accuracy only decreased slightly by 0.176 pp, indicating that rounding to 01111000_2 introduces minimal impact while affecting a substantial portion of the network's parameters.
- Rounding to 01111100_2 (2^{-3})
 - **Modified Tensors and Values:** Out of the 282 tensors, 206 were modified, affecting a total of 7,222,430 values.
 - **Post-Modification Accuracy:** The network's accuracy dropped drastically to 0.082%, with only 41 correct predictions and 49,959 incorrect predictions.
 - **Accuracy Impact:** The significant drop in accuracy demonstrates that rounding to 01111100_2 severely compromises the network's functionality, rendering it practically useless. This level of rounding

dramatically reduces the representational capacity of the parameters, leading to substantial errors in inference.

The results of these experiments highlight the delicate balance between parameter modification and network accuracy. Rounding the exponent values to 01100000_2 , 01110000_2 , and 01111000_2 results in minimal accuracy loss, even as the number of modified parameters increases significantly. This suggests that moderate rounding strategies can be employed for memory optimization without significantly compromising the network's performance.

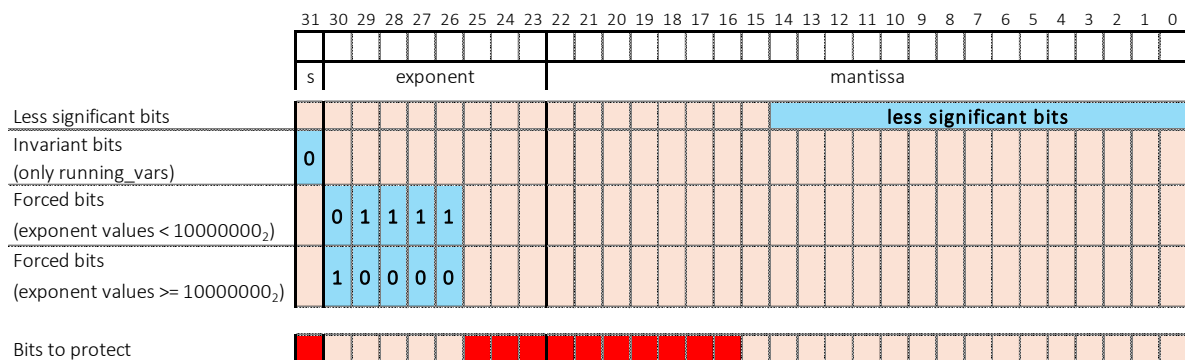
However, the results also show that more aggressive rounding, such as to 01111100_2 , leads to a catastrophic drop in accuracy, indicating that there is a critical threshold beyond which further modifications render the network ineffective. This finding underscores the importance of careful consideration when implementing exponent bit-fixing strategies, as overly aggressive modifications can severely degrade the network's performance.

Moving forward, this approach should be applied judiciously, with further research needed to explore the optimal balance between rounding extent and network robustness across different neural networks and application scenarios.

3.8 Summary of Results and Final Analysis

During the experiments, the exponent and mantissa bits in the ShuffleNet parameters were identified and classified, leading to several categories. **Figure 6** shows the results.

Figure 6. Classification of Exponent and Mantissa Bits



3.8.1 Categories of values of bits.

Less Significant Bits (LSBs):

In the mantissa, bits that have minimal impact on the network's accuracy were identified (highlighted in blue on **Figure 6**, bits 0 through 14). These bits can be used to store parity bits, as their alteration does not significantly affect the overall network performance. Moreover, the prediction accuracy of this network increased by 0.022 pp, from 76.194% to 76.216%.

Invariant Bits:

In the running variances parameters (the blue cell with value “0” on **Figure 6**), an invariant bit was found that remains unchanged. This bit can be used to store any information, as its value is independent of changes in other bits.

Forced Bits (Exponent Values):

- **For exponents less than 1000000_2 :** Bits were identified (highlighted in blue cells) that can be forcibly fixed at values “01111xxx₂” (for exponents less than 1000000_2), minimizing the impact on accuracy (the network's prediction accuracy decreased by only 0.176 pp, from 76.194% to 76.018%). These bits remain invariant after rounding and can be used to store parity bits.
- **For exponents greater than or equal to 1000000_2 :** Bits were identified that can be forcibly fixed at values “10000xxx₂” (for exponents $\geq 1000000_2$). These bits can also be used to store parity bits, as they remain invariant during rounding. However, it is crucial to monitor the maximum exponent value that may occur in the tensors, as exceeding this fixed range could lead to significant inaccuracies or loss of precision in the network's predictions.

3.8.2 Bits to Protect:

The red cells on the diagram indicate bits that must be protected, as they directly affect the accuracy of the network's computations and predictions. These bits include:

- The sign bit (bit 31) in weights, biases and running means.
- 3 bits of the exponent.
- 7 bits of the mantissa.

3.8.3 Final Result and Potential Application of Error-Correcting Codes (ECC):

Next, we applied simultaneous modifications by altering the least significant bits (LSBs) in the mantissa and fixing specific bits in the exponent. This combined approach aimed to explore the cumulative impact of these adjustments on the network's accuracy while potentially freeing up additional bits for fault-tolerant encoding. The results demonstrated that even with both sets of modifications, the accuracy remained within an acceptable range, indicating the feasibility of implementing such changes in tandem. The detailed results of this experiment are presented in **Table 12**.

Table 12. Impact of Combined Exponent Rounding and Mantissa Bit Alteration on Model Accuracy

| run | bit exponent mask | TRUE | FALSE | TRUE | FALSE |
|---|-----------------------|--------|--------|---------|---------|
| Golden Run | | 38,097 | 11,903 | 76.194% | 23.806% |
| Rounding to 2^{-7} | 01111000 ₂ | 38,009 | 11,991 | 76.018% | 23.982% |
| Altering 16 bits of mantiss | - | 37,629 | 12,371 | 75.258% | 24.742% |
| Rounding to 2^{-7} and altering 16 bits of mantissa | 01111000 ₂ | 37,116 | 12,884 | 74.232% | 25.768% |

As we can see from the results table, the accuracy decreased to 75.448%, showing a slight drop of 0.746 pp compared to the baseline. This is a more significant decrease than when the methods are applied separately. However, it still has a moderate impact on the model's accuracy.

3.8.4 Potential Application of Error-Correcting Codes (ECC):

Based on the results obtained, we have **21 bits** available for use as parity bits and **11 bits** that need to be protected. To protect these bits, depending on the requirements, various types of Error-Correcting Codes (ECC) can be employed:

- **Hamming (15,11):** If 11 data bits need to be protected (1 sign bit, 3 exponent bits, and 7 mantissa bits), a Hamming (15,11) code can be used. This code provides protection against single-bit errors by using 4 additional parity bits. Given the available non-significant and invariant bits, this code can be easily implemented.
- **Hamming (21,16):** In layers where invariants are absent and all 8 exponent bits, along with the sign bit and 7 mantissa bits (a total of 16 data bits), need to be protected, a Hamming (21,16) code can be utilized, which requires 5 parity bits. This code can also be effectively implemented using the available bits.
- **SEC-DED (Single Error Correction, Double Error Detection):** This code extends the Hamming code by adding an additional parity bit, allowing for the correction of single-bit errors and the detection of double-bit errors. It provides a balance between complexity and reliability, making it a robust choice for systems where double-error detection is critical.
- **BCH (Bose-Chaudhuri-Hocquenghem) Codes:** BCH codes are more flexible and can be configured to correct multiple errors. For example, a BCH (31,21) code can correct up to two errors in a block of 31 bits. This makes BCH codes suitable for scenarios where higher fault tolerance is required.
- **SEC-SED (Single Error Correction, Single Error Detection):** While less common, SEC-SED codes can be used in specific applications where detection and correction of single errors in different parts of the data are needed.

The implementation of these codes and further experimentation can serve as the foundation for future research. By exploring different ECC strategies, it is possible to find the optimal balance between accuracy, complexity, and fault tolerance, paving the way for more robust neural network models.

3.9 Summary of the Methodology

This section summarizes the systematic methodology developed to analyze and optimize the performance of Convolutional Neural Networks (CNNs) by exploring the potential for fixing specific bits within the floating-point parameters. This approach focuses on understanding the impact of rounding exponent values and modifying bit representations on network accuracy, with the goal of enhancing memory efficiency and incorporating fault tolerance mechanisms. The methodology is structured into the following key steps:

1. Golden Run:

- The process begins with conducting a “Golden Run” for the target neural network. This involves evaluating the network on a standard test set to establish a baseline accuracy. The results from this baseline run provide a critical reference point for assessing the impact of any subsequent modifications to the network's parameters.

2. Identification of Least Significant Bits (LSBs):

- Next, we identify the Least Significant Bits (LSBs) within the mantissa that can be considered less critical for the network's accuracy. This identification is done separately for different types of parameters, including weights, biases, `running_mean`, and `running_var`. The goal is to determine which bits in the mantissa can potentially be fixed or modified without significantly affecting the network's performance.

3. Identification of Invariant Bits:

Following the analysis of least significant bits and parameter magnitudes, we identify the invariant bits across different parameters, such as weights, biases, running means, and running variances. Invariant bits are those that remain constant across various data values and do not affect the accuracy of the network. By identifying these bits, we can determine which bits can be safely fixed or utilized for storing additional information, such as parity bits for error correction. This step is critical for networks where a significant portion of the parameters contains invariant bits, enabling further optimization of memory and fault-tolerance mechanisms.

4. Assessment of Parameter Magnitudes:

- The methodology then involves analyzing the magnitude of the parameter values, specifically focusing on the exponent part of the floating-point representation. This assessment is performed for each category of parameters—weights, biases, `running_mean`, and `running_var`. The analysis helps to categorize the parameters

into groups based on their exponent values, which will inform the bit-fixing strategy.

5. Rounding and Bit-Fixing Analysis for Group 1:

- Finally, assuming that the distribution of parameter magnitudes observed in ShuffleNet is representative of other networks, we conduct an analysis focusing on parameters with exponents in **Group 1** (exponents less than 01111000_2). The analysis involves rounding these small values to a standard exponent (e.g., 01111000_2) and zeroing out the less significant bits in the exponent. This step aims to assess how such rounding affects the network's accuracy and whether it can be applied as a general strategy across different CNN architectures.

6. Evaluation of Available Parity Bits and Data Bits for ECC:

The final step involves checking the number of available parity bits and the number of required data bits. This step is crucial for determining the feasibility of applying suitable Error-Correcting Codes (ECC), such as Hamming codes, within the network. By evaluating the balance between available parity bits and the bits that need protection, we can identify the most appropriate ECC strategy to enhance the network's fault tolerance.

This step-by-step methodology provides a structured approach to optimizing CNNs by selectively modifying their parameter representations. By systematically applying these steps, we can explore the potential for memory savings and improved fault tolerance in a variety of neural network models while maintaining acceptable levels of accuracy. This approach sets the stage for further experiments on different CNN architectures to determine the generalizability and effectiveness of the bit-fixing strategy.

4. Experiments and Results

In this section, we present the results of the experiments conducted to evaluate the proposed methodology for optimizing Convolutional Neural Networks (CNNs) by modifying specific bits within the floating-point parameters. The experiments were designed to explore the impact of rounding exponent values and altering the least significant bits (LSBs) in the mantissa, with the goal of assessing the effects on network accuracy and fault tolerance. By systematically modifying the floating-point representations, we aim to determine whether the proposed bit-fixing strategies can enhance the memory efficiency of CNNs while maintaining their robustness in critical applications.

To ensure comprehensive analysis, the experiments were conducted across multiple well-known CNN architectures, including GoogLeNet, InceptionV3, MobileNet V2, MobileNet V3 (both Small and Large variants), ResNet50 (versions 1 and 2), and ShuffleNet V2 with a width multiplier of 2.0. These networks represent a wide variety of model complexities and applications, making them ideal candidates for evaluating the generality of the proposed methods. For each network, we performed a baseline evaluation (“Golden Run”) to establish reference accuracy levels, followed by incremental tests focusing on rounding and modifying floating-point parameters. The results highlight the balance between precision loss and memory savings, as well as the potential for integrating Error-Correcting Codes (ECC) to improve fault tolerance.

4.1 Golden Run Results

The Golden Run provides the baseline accuracy for each neural network model, serving as the reference point for evaluating subsequent bit-modification experiments. **Table 13** presents the results of the Golden Run, showing the number of correct and incorrect predictions for each CNN model, along with the corresponding accuracy percentages.

Table 13. Golden Run Results for Each CNN

| CNN | TRUE | FALSE | TRUE | FALSE |
|--------------------|--------|--------|---------|---------|
| GoogLeNet | 34,886 | 15,114 | 69.772% | 30.228% |
| InceptionV3 | 34,761 | 15,239 | 69.522% | 30.478% |
| MobileNet V2 | 36,003 | 13,997 | 72.006% | 27.994% |
| MobileNet V3 Small | 33,834 | 16,166 | 67.668% | 32.332% |
| MobileNet V3 Large | 37,657 | 12,343 | 75.314% | 24.686% |
| ResNet50V1 | 38,073 | 11,927 | 76.146% | 23.854% |
| ResNet50V2 | 40,427 | 9,573 | 80.854% | 19.146% |
| ShuffleNetV2_2.0 | 38,097 | 11,903 | 76.194% | 23.806% |

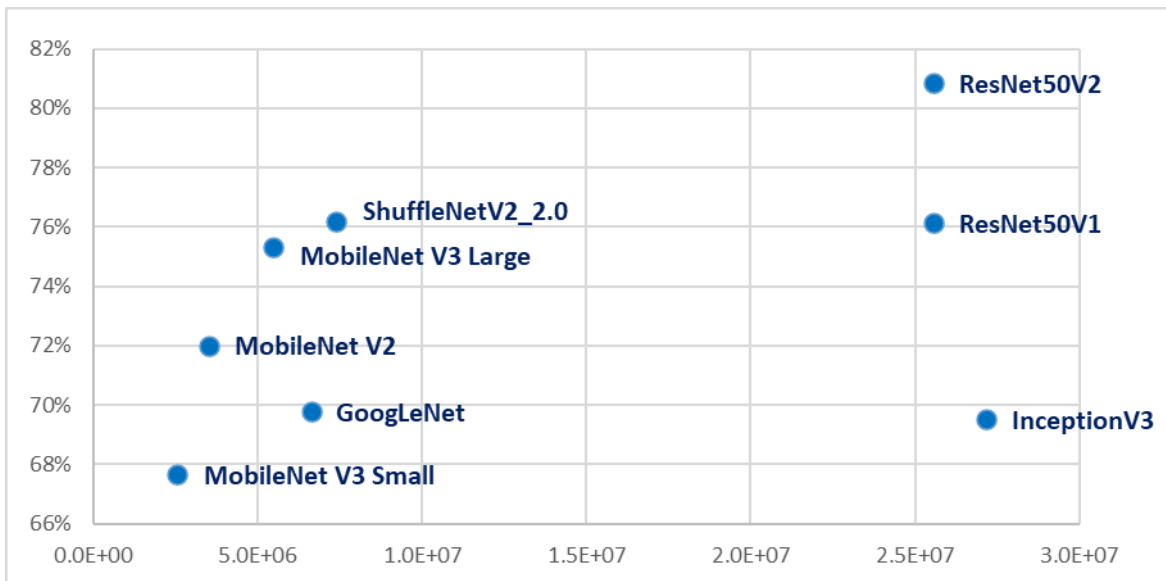
To further analyze the performance of the networks, **Table 14** compares the accuracy of each CNN model with the number of parameters (weights and biases only). This provides insight into the relationship between model complexity (measured by the number of parameters) and prediction accuracy.

Table 14. Comparison of Network Accuracy and Parameter Counts

| CNN | Number of Parameters | Accuracy |
|--------------------|----------------------|----------|
| GoogLeNet | 6,624,904 | 69.772% |
| InceptionV3 | 27,161,264 | 69.522% |
| MobileNet V2 | 3,504,872 | 72.006% |
| MobileNet V3 Small | 2,542,856 | 67.668% |
| MobileNet V3 Large | 5,483,032 | 75.314% |
| ResNet50V1 | 25,557,032 | 76.146% |
| ResNet50V2 | 25,557,032 | 80.854% |
| ShuffleNetV2_2.0 | 7,393,996 | 76.194% |

Figure 7 illustrates the relationship between the accuracy of each CNN and the number of parameters. As shown, ResNet50V2 achieves high accuracy with a relatively large number of parameters, while smaller networks like MobileNet V3 Small have lower accuracy but fewer parameters. Notably, InceptionV3, despite having a significant number of parameters, does not outperform networks with fewer parameters such as ResNet50V2.

Figure 7. Ratio of CNN Accuracy to Number of Parameters



Although larger models like ResNet50V2 tend to have higher accuracy due to a greater number of parameters, this is not always the case. For example, InceptionV3, despite having one of the highest parameter counts, does not achieve better accuracy than models like ShuffleNetV2_2.0, which has fewer parameters. This suggests that increasing the number of parameters alone does not necessarily guarantee improved performance.

However, the primary goal of this study is not to optimize parameter efficiency but to establish the baseline accuracy (Golden Run) for each model. These results serve as a reference point for evaluating the impact of bit modifications and fault tolerance techniques in subsequent experiments.

4.2 Identification of Least Significant Bits (LSBs)

As part of our methodology, we identify the Least Significant Bits (LSBs) within the mantissa that can be considered less critical for the network's accuracy. This identification is carried out separately for different types of parameters, including weights, biases, running means, and running variances. The goal is to determine which bits in the mantissa can potentially be fixed or modified without significantly affecting the network's overall performance.

Table 15 presents the accuracy of each network as we progressively alter bits from 0-12 to 0-22. Networks exhibit different behavior, but one common trend is that at a certain point, the accuracy begins to sharply decline.

Table 15. Network Accuracy When Altering Least Significant Bits (LSBs) in the Mantissa

| Altered bits | GoogLeNet | InceptionV3 | MobileNet V2 | MobileNet V3 Small | MobileNet V3 Large | ResNet50V1 | ResNet50V2 | ShuffleNet V2_2.0 |
|-------------------|----------------|----------------|----------------|--------------------|--------------------|----------------|----------------|-------------------|
| bit_22 | 0.104% | 0.102% | 0.112% | 0.088% | 0.110% | 0.110% | 0.098% | 0.100% |
| bit_21 | 0.110% | 0.086% | 0.098% | 0.092% | 0.136% | 0.160% | 0.100% | 0.100% |
| bit_20 | 0.166% | 0.130% | 0.124% | 0.108% | 0.126% | 23.164% | 3.146% | 0.266% |
| bit_19 | 15.620% | 4.436% | 0.836% | 0.218% | 0.624% | 70.436% | 58.824% | 23.990% |
| bit_18 | 54.468% | 44.430% | 34.014% | 15.032% | 37.038% | 74.788% | 75.760% | 53.016% |
| bit_17 | 67.200% | 64.506% | 64.538% | 56.546% | 71.080% | 75.812% | 80.124% | 71.606% |
| bit_16 | 69.264% | 68.330% | 70.610% | 63.206% | 74.384% | 75.946% | 80.558% | 75.258% |
| bit_15 | 69.686% | 69.066% | 71.844% | 65.498% | 74.920% | 76.112% | 80.804% | 75.818% |
| bit_14 | 69.782% | 69.346% | 71.874% | 67.270% | 75.304% | 76.086% | 80.810% | 76.216% |
| bit_13 | 69.790% | 69.450% | 72.014% | 67.582% | 75.354% | 76.108% | 80.796% | 76.212% |
| bit_12 | 69.782% | 69.468% | 71.982% | 67.598% | 75.312% | 76.150% | 80.830% | 76.194% |
| Golden Run | 69.772% | 69.522% | 72.006% | 67.668% | 75.314% | 76.146% | 80.854% | 76.194% |

Next, **Table 16** shows the deviation in percentage points from the Golden Run. The networks are color-coded to highlight the threshold at which we will focus our further experiments. This threshold is the maximum number of bits that can be altered without a significant loss of accuracy.

Table 16. Deviation in Accuracy from Golden Run (Percentage Points)

| Altered bits | GoogLeNet | InceptionV3 | MobileNet V2 | MobileNet V3 Small | MobileNet V3 Large | ResNet50V1 | ResNet50V2 | ShuffleNet V2_2.0 |
|-------------------|----------------|----------------|----------------|--------------------|--------------------|----------------|----------------|-------------------|
| bit_22 | -69.668 pp | -69.420 pp | -71.894 pp | -67.580 pp | -75.204 pp | -76.036 pp | -80.756 pp | -76.094 pp |
| bit_21 | -69.662 pp | -69.436 pp | -71.908 pp | -67.576 pp | -75.178 pp | -75.986 pp | -80.754 pp | -76.094 pp |
| bit_20 | -69.606 pp | -69.392 pp | -71.882 pp | -67.560 pp | -75.188 pp | -52.982 pp | -77.708 pp | -75.928 pp |
| bit_19 | -54.152 pp | -65.086 pp | -71.170 pp | -67.450 pp | -74.690 pp | -5.710 pp | -22.030 pp | -52.204 pp |
| bit_18 | -15.304 pp | -25.092 pp | -37.992 pp | -52.636 pp | -38.276 pp | -1.358 pp | -5.094 pp | -23.178 pp |
| bit_17 | -2.572 pp | -5.016 pp | -7.468 pp | -11.122 pp | -4.234 pp | -0.334 pp | -0.730 pp | -4.588 pp |
| bit_16 | -0.508 pp | -1.192 pp | -1.396 pp | -4.462 pp | -0.930 pp | -0.200 pp | -0.296 pp | -0.936 pp |
| bit_15 | -0.086 pp | -0.456 pp | -0.162 pp | -2.170 pp | -0.394 pp | -0.034 pp | -0.050 pp | -0.376 pp |
| bit_14 | 0.010 pp | -0.176 pp | -0.132 pp | -0.398 pp | -0.010 pp | -0.060 pp | -0.044 pp | 0.022 pp |
| bit_13 | 0.018 pp | -0.072 pp | 0.008 pp | -0.086 pp | 0.040 pp | -0.038 pp | -0.058 pp | 0.018 pp |
| bit_12 | 0.010 pp | -0.054 pp | -0.024 pp | -0.070 pp | -0.002 pp | 0.004 pp | -0.024 pp | 0.000 pp |
| Golden Run | 69.772% | 69.522% | 72.006% | 67.668% | 75.314% | 76.146% | 80.854% | 76.194% |

Further Research

As seen in the tables, different networks exhibit varying levels of resilience when altering the least significant bits. However, one consistent observation is that beyond a certain point, accuracy sharply decreases. For further experiments, we will focus on the thresholds highlighted in the second table, selecting the maximum number of bits that can be modified while maintaining an acceptable level of accuracy. This step is one part of our broader experiment to optimize network performance by leveraging LSBs and invariant bits.

4.3 Analysis of Invariant Bits

According to our methodology, we conducted an analysis of invariant bits across various CNN architectures. Invariant bits are those that remain unchanged across different parameter values and can potentially be used for error-correcting codes (ECC) or optimization techniques without significantly affecting the network's performance. The results presented in **Table 17** show the number of invariant bits identified in different parameter types (weights, biases, running means, and running variances) across several CNN architectures.

Table 17. Distribution of Invariant Bits Across Various CNN Architectures and Parameter Types

| CNN | Invariant bits (number of bits and their numbers) | | | | Number of parameters with invariant bits | % of total |
|--------------------|---|---------------------------|---------------|------------------|--|------------|
| | weights | biases | running means | running variance | | |
| GoogLeNet | 3 bits: 28, 29, 30 | | | 1 bit: 31 | 6,623,904 | 99.766% |
| InceptionV3 | 2 bits: 29, 30 | | | 1 bit: 31 | 27,159,264 | 99.859% |
| MobileNet V2 | | | | 1 bit: 31 | 17,056 | 0.482% |
| MobileNet V3 Small | | | | 1 bit: 31 | 6,056 | 0.237% |
| MobileNet V3 Large | | | | 1 bit: 31 | 12,200 | 0.222% |
| ResNet50V1 | 2 bits: 29, 30 | 3 bits: 28, 29, 30 | | 1 bit: 31 | 25,583,592 | 99.896% |
| ResNet50V2 | | | | 1 bit: 31 | 26,560 | 0.104% |
| ShuffleNetV2_2.0 | | | | 1 bit: 31 | 16,858 | 0.227% |

- High percentage of invariant bits:** Architectures such as GoogLeNet, InceptionV3, and ResNet50V1 have a significant portion of parameters with invariant bits. For instance, GoogLeNet has 3 invariant bits (bits 28, 29, 30) in its weights and 1 invariant bit (bit 31) in its running variances, covering 99.766% of its parameters. Similarly, InceptionV3 and ResNet50V1 demonstrate a high number of invariant bits across both weights and biases, affecting 99.859% and 99.896% of their parameters, respectively. This makes these networks prime candidates for further exploration of techniques that leverage invariant bits, such as error-correcting codes or bit-fixing strategies, to potentially improve fault tolerance or memory efficiency.
- Low percentage of invariant bits:** On the other hand, networks like MobileNet V2, MobileNet V3 Small, MobileNet V3 Large, and ShuffleNetV2_2.0 show a much lower percentage of parameters with invariant bits, ranging from 0.104% to 0.482%. These networks only exhibit a single invariant bit in their running variances (bit 31), and the overall number of parameters with invariant bits is much smaller compared to the previous group. These architectures may not benefit as much from techniques focused on

invariant bits, as the potential impact on accuracy may be more pronounced with fewer invariant bits available for modification.

Further Research

For networks like **GoogLeNet**, **InceptionV3**, and **ResNet50V1**, where the majority of parameters exhibit invariant bits, additional experiments can be conducted to study how fixing these bits might affect the overall accuracy of the network. By systematically fixing these invariant bits, we can explore the trade-off between fault tolerance, memory optimization, and network accuracy. Such experiments would help determine whether these networks can maintain their high accuracy while benefiting from techniques that exploit their abundant invariant bits.

For networks with fewer invariant bits, such as **MobileNet V2** or **ShuffleNetV2_2.0**, different strategies might be needed, focusing more on selective parameter optimization or exploring the effects of fixing smaller subsets of bits.









4.4 Assessment of Parameter Magnitudes

As part of our methodology, we conducted an analysis of parameter magnitudes, focusing specifically on the exponent part of the floating-point representation. This assessment was carried out across various parameter categories—weights, biases, running means, and running variances. The goal of this step is to classify parameters based on their exponent values, which will inform our bit-fixing strategy.

Detailed results for each network are provided in Section 7: Annexes. Here, we present a summary of the key findings aimed at analyzing the maximum exponent values. The outcome of this analysis is to determine how many bits in the exponent can be safely fixed without significantly impacting the network’s performance.

Table 18 summarizes the results. For each architecture, the maximum exponent value and the number of bits that can be “freely” fixed are shown.

Table 18. Analysis of maximum exponent value

| CNN | Max exponent value | Bits "free" to fix |
|--------------------|-----------------------|--|
| GoogLeNet | 10000011 ₂ |  6 |
| InceptionV3 | 10000011 ₂ |  6 |
| MobileNet V2 | 10000110 ₂ |  5 |
| MobileNet V3 Small | 10000101 ₂ |  5 |
| MobileNet V3 Large | 10001000 ₂ |  4 |
| ResNet50V1 | 10000010 ₂ |  6 |
| ResNet50V2 | 10000111 ₂ |  5 |
| ShuffleNetV2_2.0 | 10000111 ₂ |  5 |

Key Observations:

- **Maximum Number of Bits to Fix:** Networks like **GoogLeNet**, **InceptionV3**, and **ResNet50V1** have the highest potential for bit-fixing, allowing up to 6 bits in the

exponent to be fixed. This provides significant room for optimization and fault-tolerance improvements.

- **Moderate Fixing Potential:** Networks such as **MobileNet V2**, **MobileNet V3 Small**, **ResNet50V2**, and **ShuffleNetV2_2.0** allow for fixing 5 bits. While this is slightly lower than the previous networks, it still offers opportunities for optimization.
- **Lower Fixing Potential:** **MobileNet V3 Large** has a maximum exponent value of 10001000_2 , allowing only 4 bits to be fixed. This network offers less flexibility in terms of bit-fixing, requiring a more cautious approach when modifying exponent bits.

Thus, analyzing the maximum exponent values allows us to identify the number of bits that can be safely fixed in each architecture. This provides opportunities for further application of optimization techniques and error correction based on fixing invariant bits and less significant exponent bits.

4.5 Rounding and Bit-Fixing Analysis

In this stage of our methodology, we focus on parameters with exponents in **Group 1** (exponents less than 01111000_2). The goal is to evaluate how rounding these small exponent values to a standard exponent (such as 01111000_2) and zeroing out the less significant bits in the exponent impacts the overall accuracy of the network. This analysis will help determine if this rounding and bit-fixing approach can be applied as a general strategy across various CNN architectures. Detailed results for each network are provided in Section 7: Annexes.

Table 19 presents the absolute accuracy of the modified networks, where the exponent bits have been adjusted by rounding to different values. For comparison, the accuracy of the original network without any modifications (Golden Run) is also provided. The rounding is performed to exponents 01100000_2 , 01110000_2 , 01111000_2 , and 01111100_2 .

Table 19. Network Accuracy after Rounding Exponents

| CNN | Golden Run | rounding to... | | | |
|--------------------|------------|----------------|--------------|--------------|--------------|
| | | 01100000_2 | 01110000_2 | 01111000_2 | 01111100_2 |
| GoogLeNet | 69.772% | 69.772% | 69.772% | 68.980% | 0.100% |
| InceptionV3 | 69.522% | 69.522% | 69.522% | 65.060% | 0.096% |
| MobileNet V2 | 72.006% | 72.006% | 72.006% | 70.254% | 0.106% |
| MobileNet V3 Small | 67.668% | 67.668% | 67.670% | 60.664% | 0.104% |
| MobileNet V3 Large | 75.314% | 75.314% | 75.314% | 55.304% | 0.108% |
| ResNet50V1 | 76.146% | 76.146% | 76.146% | 0.250% | 0.090% |
| ResNet50V2 | 80.854% | 80.854% | 80.854% | 76.780% | 0.096% |
| ShuffleNetV2_2.0 | 76.194% | 76.194% | 76.200% | 76.018% | 0.082% |

Table 20 highlights the deviation in accuracy (in percentage points) from the Golden Run for each network after applying the rounding of exponents. Green cells indicate the rounding strategy that will be used in further experiments, as it provides the maximum number of bits that can be modified while keeping the accuracy within acceptable limits.

Table 20. Difference in Accuracy (Percentage Points) from Golden Run

| CNN | Golden Run | rounding to... | | | |
|--------------------|------------|----------------------|----------------------|----------------------|----------------------|
| | | 0110000 ₂ | 0111000 ₂ | 0111100 ₂ | 0111110 ₂ |
| GoogLeNet | 69.772% | 0 pp | 0 pp | -0.792 pp | -69.672 pp |
| InceptionV3 | 69.522% | 0 pp | 0 pp | -4.462 pp | -69.426 pp |
| MobileNet V2 | 72.006% | 0 pp | 0 pp | -1.752 pp | -71.900 pp |
| MobileNet V3 Small | 67.668% | 0 pp | 0.002 pp | -7.004 pp | -67.564 pp |
| MobileNet V3 Large | 75.314% | 0 pp | 0 pp | -20.010 pp | -75.206 pp |
| ResNet50V1 | 76.146% | 0 pp | 0 pp | -75.896 pp | -76.056 pp |
| ResNet50V2 | 80.854% | 0 pp | 0 pp | -4.074 pp | -80.758 pp |
| ShuffleNetV2_2.0 | 76.194% | 0 pp | 0.006 pp | -0.176 pp | -76.112 pp |

Key Observations:

- **Rounding to 0110000₂** has no impact on the performance of any of the networks, as their accuracy remains identical to the Golden Run for all architectures. This suggests that this level of rounding is safe to apply without degrading the network's predictive capabilities.
- **Rounding to 0111110₂** is catastrophic for all networks, leading to drastic accuracy reductions across the board. In most cases, the networks become unusable with near-zero accuracy, making this level of rounding impractical for further experiments.

Given these results, we will focus our subsequent experiments on the intermediate rounding values of 0111000₂ and 0111100₂, which offer a balance between modifying the exponent bits and maintaining acceptable levels of accuracy.

Additionally, the number of modified parameters resulting from these rounding strategies is detailed in **Section 7**. This section provides a comprehensive breakdown of how many parameters are affected in each network and helps inform our next steps in the experimental process.

4.6 Evaluation of Available Parity Bits and Data Bits for ECC

In this section, we analyze the results obtained from the previous stages to assess how many parity bits are available for each network, and how many data bits need to be protected. This analysis is crucial for determining the potential effectiveness of Error-Correcting Codes (ECC) within the architecture and identifying the most suitable ECC schemes for each network.

4.6.1 Analysis Summary:

- **4.2 Identification of Least Significant Bits (LSBs):** This step identified the least significant bits in the mantissa of each network, which are less critical for maintaining the accuracy of the network. These bits can be repurposed to store parity information for ECC, as their modification does not significantly impact network accuracy.

- 4.4 Assessment of Parameter Magnitudes:** In this stage, we assessed the constraints imposed by the maximum exponent values across different networks. By analyzing the exponent values, we determined how many bits in the exponent can be safely fixed without affecting the accuracy, such as rounding exponents to 0b1000xxx or 0b100xxxx. These fixed bits can also be utilized for storing parity bits, further enhancing the fault-tolerance of the network.
- 4.5 Rounding and Bit-Fixing Analysis:** This section evaluated the effect of modifying small exponent values, focusing on minimizing the impact of rounding on accuracy. By zeroing out the less significant bits in the exponent, we were able to identify which bits can be fixed across each network without causing performance degradation.

4.6.2 Overview of Results:

Figure 8 below provides a comprehensive overview of the available parity bits and the bits that must be protected for each network.

Figure 8. The overview of the available parity bits and the bits that must be protected

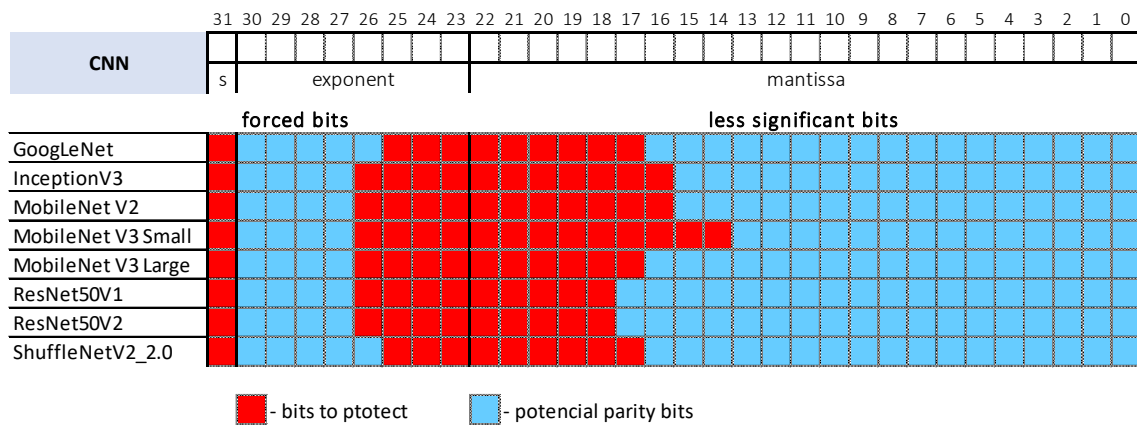


Table 21 summarizes the number of parity bits available and the number of data bits that need to be protected for each network. The parity bits are calculated based on the identification of less significant mantissa bits and fixed exponent bits. The bits to protect include critical bits from the mantissa, exponent, and sign bit that are essential for the network’s functionality.

Table 21. The number of parity bits available and the number of data bits that need to be protected

| CNN | Potencial parity bits | Bits to protect |
|--------------------|-----------------------|-----------------|
| GoogLeNet | 22 | 10 |
| InceptionV3 | 20 | 12 |
| MobileNet V2 | 20 | 12 |
| MobileNet V3 Small | 18 | 14 |
| MobileNet V3 Large | 21 | 11 |
| ResNet50V1 | 22 | 10 |
| ResNet50V2 | 22 | 10 |
| ShuffleNetV2_2.0 | 22 | 10 |

4.6.3 Parity Bits and Protection Requirements:

In this section, we analyze how the accuracy of each network changes when applying the scenarios we developed for identifying parity bits and bits to protect. The focus is on evaluating the impact of rounding exponent bits and altering mantissa bits, as well as the cumulative effect of combining these approaches (the “complex experiment”). This analysis helps us understand the trade-offs involved in optimizing network robustness while minimizing the loss of accuracy.

Table 22 compares the accuracy results for several networks before and after applying different bit modification strategies, including rounding the exponent and altering the mantissa. The **Golden Run** accuracy serves as the baseline for comparison.

Table 22. The accuracy results after applying different bit modification strategies

| CNN | GoogLeNet | Inception V3 | MobileNet V2 | MobileNet V3 Small | MobileNet V3 Large | ResNet50 V1 | ResNet50 V2 | ShuffleNet V2_2.0 |
|------------------------------|------------------|------------------|------------------|--------------------|--------------------|------------------|------------------|-------------------|
| Golden Run | 69.772% | 69.522% | 72.006% | 67.668% | 75.314% | 76.146% | 80.854% | 76.194% |
| Rounding the exponent | | | | | | | | |
| to 2^{-7} | 68.980% | | | | | | | 76.018% |
| to 2^{-15} | | 69.522% | 72.006% | 67.670% | 75.314% | 76.146% | 80.854% | |
| Deviation | -0.792 pp | 0.000 pp | 0.000 pp | 0.002 pp | 0.000 pp | 0.000 pp | 0.000 pp | -0.176 pp |
| Altering the mantissa | | | | | | | | |
| 14 bits | | | | 67.270% | | | | |
| 15 bits | | | 71.874% | | | | | |
| 16 bits | 69.264% | 68.330% | | | 74.384% | | | 75.258% |
| 17 bits | | | | | | 75.812% | 80.124% | |
| Deviation | -0.508 pp | -1.192 pp | -0.132 pp | -0.398 pp | -0.930 pp | -0.334 pp | -0.730 pp | -0.936 pp |
| Complex experiment | 68.548% | 68.328% | 71.840% | 67.270% | 74.386% | 75.814% | 80.124% | 74.232% |
| Deviation | -1.224 pp | -1.194 pp | -0.166 pp | -0.398 pp | -0.928 pp | -0.332 pp | -0.730 pp | -1.962 pp |

Key Observations:

- **GoogLeNet, Inception V3, and ShuffleNet V2_2.0** show the most significant accuracy loss during the **Complex Experiment**, with deviations of **-1.224 pp**, **-1.194 pp**, and **-1.962 pp**, respectively. These networks are more sensitive to the combined modification

of both the exponent and mantissa, suggesting that a more cautious approach may be required when applying both strategies simultaneously.

- **ResNet50 V2** and **MobileNet V3 Large** also experience considerable drops in accuracy during the **Complex Experiment** with deviations of **-0.730 pp** and **-0.928 pp**. This indicates that altering both the exponent and mantissa in these networks can lead to non-negligible degradation in performance, highlighting the need for more selective bit modification.
- **MobileNet V2**, **ResNet50 V1**, and **MobileNet V3 Small** handle the combined approach better, with relatively small deviations in accuracy. **MobileNet V2** sees a deviation of only **-0.166 pp**, and **ResNet50 V1** experiences a drop of **-0.332 pp**, and **MobileNet V3 Small** has **-0.398 pp** deviation, suggesting that these architectures are more resilient to the simultaneous rounding of the exponent and alteration of mantissa bits. This makes them suitable candidates for optimization using the combined strategy.

Conclusion:

The analysis suggests that while the mantissa bits are highly variable and critical for

This analysis demonstrates that there are enough available parity bits in all neural networks to implement simple error-correcting codes (ECC), such as Hamming codes, to protect critical data bits. The number of bits to protect varies between **10** and **14**, depending on the network architecture. Based on these findings, we can propose specific ECC schemes for each network, ensuring optimal protection without unnecessary overhead.

- For networks that need to protect **10 bits** (e.g., **GoogLeNet**, **ResNet50V1**, **ResNet50V2**, and **ShuffleNetV2_2.0**), a **Hamming (14,10)** code can be used, which requires 4 parity bits to protect 10 data bits. Given that these networks have 22 available parity bits, this approach is efficient and ensures robust error correction against single-bit errors.
- For networks that need to protect **11 bits**, such as those with slightly higher protection needs (**MobileNet V3 Large**), the Hamming (15,11) code is a perfect fit, offering protection with 4 parity bits. Again, networks with 22 available parity bits can implement this code without any issues.
- For networks where **12 bits** need protection (e.g., **InceptionV3** and **MobileNet V2**), the Hamming (17,12) code can be used, which requires 5 parity bits. These networks typically have 20 available parity bits, so there is sufficient capacity to apply this code, ensuring efficient error correction without compromising memory usage.
- In cases where **14 bits** need protection, such as **MobileNet V3 Small**, the Hamming (19,14) code, which requires 5 parity bits, can be applied. With 18 available parity bits, this network has just enough capacity to accommodate this code, providing fault tolerance while minimizing performance loss.

Additionally, for enhanced error detection and correction, **SEC-DED** (Single Error Correction, Double Error Detection) codes can be applied. Depending on the number of bits to protect, the following SEC-DED codes may be suitable:

- For **10 bits: SEC-DED (15,10)**.
- For **11 bits: SEC-DED (16,11)**.
- For **12 bits: SEC-DED (18,12)**.
- For **14 bits: SEC-DED (20,14)**.

These codes add an extra parity bit to enhance error detection and correction capabilities, providing protection against single-bit errors and detecting double-bit errors.

In summary, the number of parity bits available in each network allows for the tailored selection of ECC schemes, ensuring efficient and reliable error correction. By applying codes like Hamming or SEC-DED, specifically adjusted for each network's bit protection requirements, we can achieve robust fault tolerance while minimizing any impact on performance and maintaining memory efficiency.

While some networks experienced significant accuracy loss during the **Complex Experiment** – which involved both rounding exponents and modifying mantissa bits – several architectures demonstrated resilience. This suggests that combining these bit-modification strategies is still viable in specific cases. For networks more sensitive to these changes, further refinement of bit-fixing strategies, along with carefully chosen ECC schemes, could reduce accuracy loss, enabling the use of fewer parity bits without sacrificing performance.

Overall, ECC strategies can be applied effectively across all neural networks. Simple error-correcting codes (such as Hamming) can be generally implemented across most architectures, offering a balance between fault tolerance and minimal impact on inference time and energy consumption. However, the real challenge lies in developing and applying more complex ECC schemes to protect CNNs from multiple faults, such as double bit-flips or burst errors affecting 3 or 4 bits.

The choice between simpler or more complex ECC schemes depends largely on the trade-offs required by the specific application. Simpler codes, while providing less protection, execute faster and consume less energy, making them ideal for real-time or energy-efficient applications. On the other hand, as error coverage increases, more advanced ECC schemes (like SEC-DED or burst-error correction codes) introduce higher computational overhead, which increases inference time and energy consumption but offers stronger error protection, making these codes more suitable for critical applications where reliability is paramount.

Ultimately, finding the right balance between error coverage, inference performance, and energy consumption is key to selecting the most appropriate ECC strategy for each neural network architecture.

5. Conclusions and future work

This thesis has explored a methodology for improving the memory efficiency and fault tolerance of Convolutional Neural Networks (CNNs) by modifying specific bits within the floating-point representations of their parameters. The approach focused on analyzing the impact of rounding exponent values and altering least significant bits (LSBs) in the mantissa, with the goal of maintaining network accuracy while freeing up bits for potential error correction codes (ECC).

The Golden Run results established the baseline accuracy of various CNN architectures, showing that networks such as ResNet50V2 achieve higher accuracy due to their increased parameter count, while smaller models like MobileNet V3 Small exhibit lower accuracy but greater efficiency. This analysis confirmed that network complexity, in terms of parameter count, does not always directly translate into better performance, underscoring the importance of architecture design in CNNs.

5.1 Key Contributions

1. **Bit-Level Analysis:** This work introduced a detailed analysis of the significance of individual bits in the mantissa and exponent of floating-point representations across different CNN layers (weights, biases, `running_mean`, `running_var`). The identification of non-critical bits for network accuracy offers the potential to optimize memory usage without compromising performance.

2. **Rounding and Bit-Fixing Techniques:** The study developed a strategy to round smaller exponent values and modify LSBs, which showed minimal impact on network accuracy during the Golden Run. This technique opens up the possibility of using freed-up bits for embedding parity information, paving the way for more memory-efficient CNN deployments.

3. **Foundation for Error-Correcting Codes (ECC):** By evaluating available parity bits and the number of data bits to be protected, this research laid the groundwork for applying suitable ECCs, such as Hamming codes, to enhance the fault tolerance of CNNs. This is particularly relevant for applications in safety-critical environments, where even minor errors can lead to significant consequences.

5.2 Limitations

Despite these advancements, there are several limitations to this research. First, the experiments conducted in this work were limited to specific CNN architectures and focused on a fixed set of modifications. The generality of the proposed techniques across a broader range of models, including more complex architectures or models trained on different datasets, has yet to be fully explored.

For example, transformer-based architectures like Vision Transformers (ViT), which are gaining popularity in computer vision tasks, rely on a fundamentally different structure than CNNs, using attention mechanisms instead of convolutions. The bit-fixing strategies applied to

floating-point parameters in CNNs might not translate directly to transformer models, which process information in a more distributed manner across multiple heads [22].

Similarly, recurrent neural networks (RNNs) and Long Short-Term Memory (LSTM) networks, commonly used in sequence-based tasks like speech recognition or time-series forecasting, maintain state information across time steps. Modifying the floating-point parameters in these architectures could have a different impact, as errors may propagate across time, leading to accumulated inaccuracies [23].

Additionally, the datasets used in this research were focused on ImageNet, which is a large, diverse dataset used for image classification tasks. However, applying these techniques to CNNs trained on more specialized datasets, such as medical imaging data (e.g., chest X-rays or MRI scans) [24] or autonomous driving datasets (e.g., KITTI or Waymo) [25], may yield different results. Networks trained on such datasets often require higher precision due to the critical nature of their applications, making them potentially more sensitive to bit modifications and errors.

Furthermore, the focus was primarily on the mantissa and exponent bits in floating-point numbers. Future studies could investigate the impact of modifying other aspects of floating-point representations or explore additional layers in greater depth, such as recurrent layers or attention mechanisms.

5.3 Future Work

The findings of this research open several promising directions for future investigation:

- 1. Extending the Analysis to More Networks:** Future experiments should evaluate the proposed methodology across a wider variety of CNN architectures and training datasets. By doing so, the robustness and applicability of the bit-fixing strategies can be more comprehensively understood.
- 2. Advanced Error-Correction Techniques:** In addition to Hamming codes, more sophisticated ECC techniques such as **BCH codes** or **Reed-Solomon codes** could be explored to handle multi-bit errors. These techniques may be particularly beneficial for high-reliability applications, where single-bit correction methods might not be sufficient.
- 3. Optimization of Quantization Strategies:** As neural networks increasingly rely on quantization for efficiency, future research could investigate the combination of bit-fixing techniques with quantization-aware training. This approach could ensure that networks remain resilient to quantization errors while still benefiting from the memory and computational savings offered by reduced precision.
- 4. Deployment in Real-World Systems:** Finally, the practical implementation of the proposed bit-modification and ECC strategies in hardware accelerators or embedded systems represents a valuable avenue for future work. By integrating these methods into real-world applications, their impact on both performance and fault tolerance can be thoroughly assessed under realistic deployment conditions.

5.4 Conclusion

In summary, this research has demonstrated that strategic modifications to the bit-level representation of floating-point parameters in CNNs can provide meaningful memory optimizations and enable the use of error-correction mechanisms. These findings have implications for the design of more efficient and robust neural networks, especially in environments where fault tolerance is critical. While the current work provides a strong foundation, further exploration into more advanced ECC methods, broader model applicability, and real-world deployment will be essential for maximizing the potential of this approach.

6. References

1. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25, 1097-1105.
2. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770-778.
3. LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436-444.
4. Sze, V., Chen, Y. H., Yang, T. J., & Emer, J. S. (2017). Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12), 2295-2329.
5. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 18, 6869-6898.
6. Han, S., Mao, H., & Dally, W. J. (2016). Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. *International Conference on Learning Representations (ICLR)*.
7. Syed, R. T., Ulbricht, M., Piotrowski, K., & Krstic, M. (2021, September). Fault resilience analysis of quantized deep neural networks. In *2021 IEEE 32nd International Conference on Microelectronics (MIEL)* (pp. 275-279). IEEE.
8. Ruiz, J. C., de Andrés, D., Saiz-Adalid, L. J., & Gracia-Morán, J. (2024). Zero-space in-weight and in-bias protection for floating-point-based CNNs. 2024 19th European Dependable Computing Conference (EDCC).
9. Gupta, S., Agrawal, A., Gopalakrishnan, K., & Narayanan, P. (2015). Deep learning with limited numerical precision. *International Conference on Machine Learning*, 1737-1746.
10. Zhang, D., Yang, J., Ye, D., & Shi, Y. (2018). Lq-nets: Learned quantization for highly accurate and compact deep neural networks. *European Conference on Computer Vision (ECCV)*, 365-382.
11. Han, S., Pool, J., Tran, J., & Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. *Advances in Neural Information Processing Systems*, 28, 1135-1143.
12. LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
13. Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
14. Courbariaux, M., Bengio, Y., & David, J. P. (2015). BinaryConnect: Training deep neural networks with binary weights during propagations. *Advances in Neural Information Processing Systems*, 28, 3123-3131.
15. Ibrahim, Younis, et al. "Soft errors in DNN accelerators: A comprehensive review." *Microelectronics Reliability* 115 (2020): 113969.
16. Li, Guanpeng, et al. "Understanding error propagation in deep learning neural network (DNN) accelerators and applications." *Proceedings of the International*

Conference for High Performance Computing, Networking, Storage and Analysis.
2017.

17. Mittal, Sparsh. "A survey on modeling and improving reliability of DNN algorithms and accelerators." *Journal of Systems Architecture* 104 (2020): 101689.
18. Shafique, Muhammad, et al. "Robust machine learning systems: Challenges, current trends, perspectives, and the road ahead." *IEEE Design & Test* 37.2 (2020): 30-57.
19. Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2), 147-160.
20. "IEEE Standard for Floating-Point Arithmetic," in IEEE Std 754-2019 (Revision of IEEE 754-2008) , vol., no., pp.1-84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229.
21. "Pytorch". An optimized tensor library for deep learning using GPUs and CPUs. Available: <https://pytorch.org>.
22. Dosovitskiy, Alexey. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv preprint arXiv:2010.11929* (2020).
23. Hochreiter, S. "Long Short-term Memory." Neural Computation MIT-Press (1997).
24. Litjens, Geert, et al. "A survey on deep learning in medical image analysis." *Medical image analysis* 42 (2017): 60-88.
25. Geiger, Andreas, et al. "Vision meets robotics: The kitti dataset." *The International Journal of Robotics Research* 32.11 (2013): 1231-1237.

7. Annexes

7.1 Model Wrapper

All experiments with bit-level modifications conducted in this thesis followed the fault-injection method. This method involves deliberately modifying specific parameters of the neural network—such as weights, biases, and running statistics—before evaluating the impact of these changes on the network’s performance. The purpose of these modifications was to simulate potential errors (e.g., soft errors or bit flips) and study the network’s resilience or vulnerability under such conditions.

While fault-injection serves as a useful tool for testing the robustness of CNNs, it does not reflect how **error-correcting codes (ECC)** would be applied in real-world systems. In practical applications, the goal is to intercept and correct errors **without directly modifying the network’s parameters**. Instead, the system would intervene at the point where the neural network accesses a parameter, dynamically correcting its value based on integrated ECC bits, such as Hamming codes or other error-correcting mechanisms.

7.1.1 Model Wrapper for On-the-Fly Parameter Correction

To simulate this real-world scenario, a custom **model wrapper** was developed. The wrapper intercepts calls to the neural network’s parameters during inference and replaces the accessed parameter values with corrected versions, without modifying the original parameters stored in memory. This allows for dynamic error correction based on predefined correction rules, similar to how ECC would function in hardware.

Code fragment 9 demonstrates the implementation of the model wrapper.

Code Fragment 9

```
# model wrapper
class ModelWrapper(torch.nn.Module):
    def __init__(self, model, parameter_corrections, buffer_corrections):
        super(ModelWrapper, self).__init__()
        self.model = model
        # Dictionary of corrected values for parameters and buffers
        self.parameter_corrections = parameter_corrections
        self.buffer_corrections = buffer_corrections

    def forward(self, x):
        # Apply parameter corrections before forward pass
        for name, param in self.model.named_parameters():
            if name in self.parameter_corrections:
                # Replace parameter value with corrected value
                param.data.copy_(self.parameter_corrections[name])

        # Apply buffer corrections (e.g., running_mean, running_var)
        for name, buffer in self.model.named_buffers():
```

```

        if name in self.buffer_corrections:
            buffer.data.copy_(self.buffer_corrections[name])

    # Forward pass with corrected parameters
    return self.model(x)

```

7.1.2 Example of Parameter and Buffer Correction

To demonstrate the functionality, **Code fragment 9** is an example of how the wrapper can be applied. The **parameter_corrections** and **buffer_corrections** dictionaries contain the corrected values for specific parameters and buffers, based on the identified bit modifications.

```

# Example of creating corrected parameters
parameter_corrections = {
    'stage2.0.branch1.0.weight': torch.tensor([...]),
    'stage2.1.branch2.1.bias': torch.tensor([...]),
    # Add more parameters as necessary
}

buffer_corrections = {
    'conv1.1.running_mean': torch.tensor([...]),
    'conv1.1.running_var': torch.tensor([...]),
    # Add more buffers as necessary
}

# Obtain the base model (e.g., ResNet, MobileNet, etc.)
current_model = CNN_MODEL

# Wrap the model with the correction wrapper
wrapped_model = ModelWrapper(current_model, parameter_corrections,
                              buffer_corrections)

```

7.1.3 Application of the Wrapper

This model wrapper allows us to simulate the application of error-correcting codes without directly modifying the original neural network parameters. During inference, any parameter or buffer accessed by the network will be dynamically replaced by its corrected version. This approach can be extended to handle a wide range of parameter corrections, making it a flexible tool for testing fault tolerance mechanisms in deep learning models.

7.1.4 Future Work on Wrapper Extensions

This model wrapper can be further expanded to incorporate more sophisticated error-correction mechanisms, such as detecting multiple-bit errors or implementing real-time parity checks, as would be required in highly fault-tolerant environments. Additionally, integrating this method with hardware accelerators or embedded systems could provide insights into the real-world applicability of error-correction techniques in neural networks.

7.2 Assessment of Parameter Magnitudes

Tables 23-30 present the detailed analysis referenced in Section 4.4, focusing on the distribution of exponent values across different layers and parameters for each neural network architecture. This includes a breakdown of the maximum and minimum exponent values observed in weights, biases, running means, and running variances.

This analysis is crucial for understanding the magnitude of parameter values and identifying which bits can be safely modified or fixed without significantly impacting the network's performance. The results serve as the foundation for applying bit-fixing strategies that aim to improve fault tolerance and memory optimization.

Table 23. Detailed Distribution of Exponent Values Across Parameters for GoogLeNet

| GoogLeNet | | | | | |
|--------------------------------|------------------|------------------|--------------|--------------|--------------|
| Value | Total number | WEIGHT | BIAS | MEAN | VAR |
| ... < 0b01100000 | 0 | 0 | 0 | 0 | 0 |
| 0b01100000 <= ... < 0b01110000 | 5,506 | 5,505 | 0 | 1 | 0 |
| 0b01110000 <= ... < 0b01111000 | 1,342,459 | 1,342,233 | 154 | 72 | 0 |
| 0b01111000 <= ... < 0b01111100 | 5,169,789 | 5,165,789 | 1,633 | 1,459 | 908 |
| 0b01111100 <= ... < 0b10000000 | 120,971 | 103,097 | 6,382 | 5,419 | 6,073 |
| 0b10000000 <= ... | 739 | 0 | 111 | 329 | 299 |
| Total | 6,639,464 | 6,616,624 | 8,280 | 7,280 | 7,280 |

Table 24. Detailed Distribution of Exponent Values Across Parameters for InceptionV3

| InceptionV3 | | | | | |
|--------------------------------|-------------------|-------------------|---------------|------------------|---------------|
| Value | Total number | WEIGHT | BIAS | MEAN | VAR |
| ... < 0b01100000 | 321 | 319 | 0 | 0 | 2 |
| 0b01100000 <= ... < 0b01110000 | 41,753 | 32,801 | 1 | 8,951 | 0 |
| 0b01110000 <= ... < 0b01111000 | 9,681,678 | 7,574,020 | 90 | 2,107,457 | 111 |
| 0b01111000 <= ... < 0b01111100 | 17,358,242 | 13,475,129 | 1,888 | 3,875,262 | 5,963 |
| 0b01111100 <= ... < 0b10000000 | 114,990 | 59,845 | 16,011 | 26,443 | 12,691 |
| 0b10000000 <= ... | 504 | 6 | 122 | 191 | 185 |
| Total | 27,197,488 | 21,142,120 | 18,112 | 6,018,304 | 18,952 |

Table 25. Detailed Distribution of Exponent Values Across Parameters for MobileNet V2

| MobileNet V2 | | | | | |
|--------------------------------|------------------|------------------|---------------|---------------|---------------|
| Value | Total number | WEIGHT | BIAS | MEAN | VAR |
| ... < 0b01100000 | 1,516 | 1,476 | 0 | 21 | 19 |
| 0b01100000 <= ... < 0b01110000 | 8,576 | 1,965 | 1,052 | 5,556 | 3 |
| 0b01110000 <= ... < 0b01111000 | 433,685 | 429,569 | 633 | 3,060 | 423 |
| 0b01111000 <= ... < 0b01111100 | 2,842,137 | 2,831,250 | 1,957 | 3,677 | 5,253 |
| 0b01111100 <= ... < 0b10000000 | 237,854 | 219,617 | 11,348 | 4,266 | 2,623 |
| 0b10000000 <= ... | 15,216 | 2,939 | 3,066 | 476 | 8,735 |
| Total | 3,538,984 | 3,486,816 | 18,056 | 17,056 | 17,056 |

Table 26. Detailed Distribution of Exponent Values Across Parameters for MobileNet V3 Small

| MobileNet V3 Small | | | | | |
|--------------------------------|------------------|------------------|---------------|--------------|--------------|
| Value | Total number | WEIGHT | BIAS | MEAN | VAR |
| ... < 0b01100000 | 982 | 897 | 7 | 39 | 39 |
| 0b01100000 <= ... < 0b01110000 | 14,887 | 11,233 | 653 | 3,000 | 1 |
| 0b01110000 <= ... < 0b01111000 | 158,025 | 156,923 | 240 | 397 | 465 |
| 0b01111000 <= ... < 0b01111100 | 1,773,013 | 1,766,537 | 3,224 | 1,325 | 1,927 |
| 0b01111100 <= ... < 0b10000000 | 605,644 | 595,589 | 6,722 | 1,212 | 2,121 |
| 0b10000000 <= ... | 2,417 | 709 | 122 | 83 | 1,503 |
| Total | 2,554,968 | 2,531,888 | 10,968 | 6,056 | 6,056 |

Table 27. Detailed Distribution of Exponent Values Across Parameters for MobileNet V3 Large

| MobileNet V3 Large | | | | | |
|--------------------------------|------------------|------------------|---------------|---------------|---------------|
| Value | Total number | WEIGHT | BIAS | MEAN | VAR |
| ... < 0b01100000 | 1,114 | 1,098 | 15 | 1 | 0 |
| 0b01100000 <= ... < 0b01110000 | 13,282 | 6,621 | 1,222 | 5,439 | 0 |
| 0b01110000 <= ... < 0b01111000 | 595,474 | 592,106 | 550 | 915 | 1,903 |
| 0b01111000 <= ... < 0b01111100 | 4,464,249 | 4,452,158 | 6,788 | 2,649 | 2,654 |
| 0b01111100 <= ... < 0b10000000 | 422,070 | 408,780 | 9,167 | 2,824 | 1,299 |
| 0b10000000 <= ... | 11,243 | 2,709 | 1,818 | 372 | 6,344 |
| Total | 5,507,432 | 5,463,472 | 19,560 | 12,200 | 12,200 |

Table 28. Detailed Distribution of Exponent Values Across Parameters for ResNet50V1

| ResNet50V1 | | | | | |
|--------------------------------|-------------------|-------------------|---------------|---------------|---------------|
| Value | Total number | WEIGHT | BIAS | MEAN | VAR |
| ... < 0b01100000 | 826 | 639 | 0 | 1 | 186 |
| 0b01100000 <= ... < 0b01110000 | 122,536 | 122,015 | 191 | 227 | 103 |
| 0b01110000 <= ... < 0b01111000 | 11,114,351 | 11,087,372 | 2,442 | 6,650 | 17,887 |
| 0b01111000 <= ... < 0b01111100 | 14,316,910 | 14,271,026 | 21,167 | 17,344 | 7,373 |
| 0b01111100 <= ... < 0b10000000 | 55,505 | 48,420 | 3,760 | 2,337 | 988 |
| 0b10000000 <= ... | 24 | 0 | 0 | 1 | 23 |
| Total | 25,610,152 | 25,529,472 | 27,560 | 26,560 | 26,560 |

Table 29. Detailed Distribution of Exponent Values Across Parameters for ResNet50V2

| ResNet50V2 | | | | | |
|--------------------------------|-------------------|-------------------|---------------|---------------|---------------|
| Value | Total number | WEIGHT | BIAS | MEAN | VAR |
| ... < 0b01100000 | 1,273 | 1,273 | 0 | 0 | 0 |
| 0b01100000 <= ... < 0b01110000 | 30,992 | 30,959 | 0 | 14 | 19 |
| 0b01110000 <= ... < 0b01111000 | 7,091,300 | 7,090,655 | 387 | 215 | 43 |
| 0b01111000 <= ... < 0b01111100 | 18,244,841 | 18,236,009 | 2,656 | 3,245 | 2,931 |
| 0b01111100 <= ... < 0b10000000 | 212,949 | 160,647 | 14,531 | 21,036 | 16,735 |
| 0b10000000 <= ... | 28,797 | 9,929 | 9,986 | 2,050 | 6,832 |
| Total | 25,610,152 | 25,529,472 | 27,560 | 26,560 | 26,560 |

Table 30. Detailed Distribution of Exponent Values Across Parameters for ShuffleNetV2_2.0

| ShuffleNetV2_2.0 | | | | | |
|--------------------------------|------------------|------------------|---------------|---------------|---------------|
| Value | Total number | WEIGHT | BIAS | MEAN | VAR |
| ... < 0b01100000 | 8,917 | 8,829 | 0 | 43 | 45 |
| 0b01100000 <= ... < 0b01110000 | 9,096 | 6,740 | 1,291 | 1,062 | 3 |
| 0b01110000 <= ... < 0b01111000 | 1,405,896 | 1,397,410 | 4,073 | 4,365 | 48 |
| 0b01111000 <= ... < 0b01111100 | 5,798,521 | 5,790,268 | 1,426 | 3,107 | 3,720 |
| 0b01111100 <= ... < 0b10000000 | 189,978 | 167,874 | 7,783 | 7,561 | 6,760 |
| 0b10000000 <= ... | 15,304 | 5,017 | 3,285 | 720 | 6,282 |
| Total | 7,427,712 | 7,376,138 | 17,858 | 16,858 | 16,858 |

7.3 Rounding and Bit-Fixing Analysis

Tables 31-38 present a detailed analysis of how the modification of exponent bits affects the total number of parameters, accuracy, and the number of true and false predictions for each CNN (see Section 4.5). The analysis is focused on different rounding levels, starting from the least significant exponents (e.g., 2^{-31}) to the most significant (e.g., 2^{-3}). Each table outlines the following:

- **Total tensors:** The total number of tensors in the network.
- **Modified tensors:** The number of tensors that were modified as a result of rounding the exponent values.
- **Total values:** The total number of parameter values in the network.
- **Modified values:** The number of parameter values that were altered due to the rounding.
- **%:** The percentage of modified values relative to the total.
- **Accuracy:** The number of correct (TRUE) and incorrect (FALSE) predictions, along with the corresponding accuracy percentages for each rounding level.

This detailed breakdown provides insight into how different exponent bit modifications affect the network's performance, as well as the extent of modification applied to the parameters. Each network is analyzed separately to observe how the architecture responds to rounding strategies.

Protecting FP-based CNNs Against Faults without Increasing Their Memory Footprint

Table 31. Effect of Exponent Bit Rounding on GoogLeNet Accuracy

| GoogLeNet | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|------------------------------|-------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| GR | | | | | | | 34,886 | 15,114 | 69.772% | 30.228% |
| rounding to 2 ⁻³¹ | 0b01100000 | 287 | 0 | 6,639,464 | 0 | 0.00% | 34,886 | 15,114 | 69.772% | 30.228% |
| rounding to 2 ⁻¹⁵ | 0b01110000 | | 58 | | 5,506 | 0.08% | 34,886 | 15,114 | 69.772% | 30.228% |
| rounding to 2 ⁻⁷ | 0b01111000 | | 107 | | 1,347,965 | 20.30% | 34,490 | 15,510 | 68.980% | 31.020% |
| rounding to 2 ⁻³ | 0b01111100 | | 166 | | 6,517,754 | 98.17% | 50 | 49,950 | 0.100% | 99.900% |

Table 32. Effect of Exponent Bit Rounding on InceptionV3 Accuracy

| InceptionV3 | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|------------------------------|-------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| GR | | | | | | | 34,761 | 15,239 | 69.522% | 30.478% |
| rounding to 2 ⁻³¹ | 0b01100000 | 484 | 4 | 27,197,488 | 321 | 0.00% | 34,761 | 15,239 | 69.522% | 30.478% |
| rounding to 2 ⁻¹⁵ | 0b01110000 | | 103 | | 42,074 | 0.15% | 34,761 | 15,239 | 69.522% | 30.478% |
| rounding to 2 ⁻⁷ | 0b01111000 | | 152 | | 443,777 | 1.63% | 32,530 | 17,470 | 65.060% | 34.940% |
| rounding to 2 ⁻³ | 0b01111100 | | 178 | | 3,285,914 | 12.08% | 48 | 49,952 | 0.096% | 99.904% |

Table 33. Effect of Exponent Bit Rounding on MobileNet V2 Accuracy

| MobileNet V2 | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|------------------------------|-------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| GR | | | | | | | 36,003 | 13,997 | 72.006% | 27.994% |
| rounding to 2 ⁻³¹ | 0b01100000 | 262 | 43 | 3,538,984 | 1,516 | 0.04% | 36,003 | 13,997 | 72.006% | 27.994% |
| rounding to 2 ⁻¹⁵ | 0b01110000 | | 102 | | 10,092 | 0.29% | 36,003 | 13,997 | 72.006% | 27.994% |
| rounding to 2 ⁻⁷ | 0b01111000 | | 152 | | 443,777 | 12.54% | 35,127 | 14,873 | 70.254% | 29.746% |
| rounding to 2 ⁻³ | 0b01111100 | | 178 | | 3,285,914 | 92.85% | 53 | 49,947 | 0.106% | 99.894% |

Table 34. Effect of Exponent Bit Rounding on MobileNet V3 Small Accuracy

| MobileNet V3 Small | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|------------------------------|-------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| GR | | | | | | | 33,834 | 16,166 | 67.668% | 32.332% |
| rounding to 2 ⁻³¹ | 0b01100000 | 210 | 25 | 2,554,968 | 982 | 0.04% | 33,834 | 16,166 | 67.668% | 32.332% |
| rounding to 2 ⁻¹⁵ | 0b01110000 | | 84 | | 15,869 | 0.62% | 33,835 | 16,165 | 67.670% | 32.330% |
| rounding to 2 ⁻⁷ | 0b01111000 | | 146 | | 173,894 | 6.81% | 30,332 | 19,668 | 60.664% | 39.336% |
| rounding to 2 ⁻³ | 0b01111100 | | 173 | | 1,946,907 | 76.20% | 52 | 49,948 | 0.104% | 99.896% |

Table 35. Effect of Exponent Bit Rounding on MobileNet V3 Large Accuracy

| MobileNet V3 Large | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|------------------------------|-------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| GR | | | | | | | 37,657 | 12,343 | 75.314% | 24.686% |
| rounding to 2 ⁻³¹ | 0b01100000 | 266 | 10 | 5,507,432 | 1,114 | 0.02% | 37,657 | 12,343 | 75.314% | 24.686% |
| rounding to 2 ⁻¹⁵ | 0b01110000 | | 85 | | 14,396 | 0.26% | 37,657 | 12,343 | 75.314% | 24.686% |
| rounding to 2 ⁻⁷ | 0b01111000 | | 161 | | 609,870 | 11.07% | 27,652 | 22,348 | 55.304% | 44.696% |
| rounding to 2 ⁻³ | 0b01111100 | | 202 | | 5,074,119 | 92.13% | 54 | 49,946 | 0.108% | 99.892% |

Table 36. Effect of Exponent Bit Rounding on ResNet50V1 Accuracy

| ResNet50V1 | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|------------------------------|-------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| GR | | | | | | | 38,073 | 11,927 | 76.146% | 23.854% |
| rounding to 2 ⁻³¹ | 0b01100000 | 267 | 33 | 25,610,152 | 826 | 0.00% | 38,073 | 11,927 | 76.146% | 23.854% |
| rounding to 2 ⁻¹⁵ | 0b01110000 | | 123 | | 123,362 | 0.48% | 38,073 | 11,927 | 76.146% | 23.854% |
| rounding to 2 ⁻⁷ | 0b01111000 | | 214 | | 11,237,713 | 43.88% | 125 | 49,875 | 0.250% | 99.750% |
| rounding to 2 ⁻³ | 0b01111100 | | 259 | | 25,554,623 | 99.78% | 45 | 49,955 | 0.090% | 99.910% |

Table 37. Effect of Exponent Bit Rounding on ResNet50V2 Accuracy

| ResNet50V2 | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|------------------------------|-------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| GR | | | | | | | 40,427 | 9,573 | 80.854% | 19.146% |
| rounding to 2 ⁻³¹ | 0b01100000 | 267 | 8 | 25,610,152 | 1,273 | 0.00% | 40,427 | 9,573 | 80.854% | 19.146% |
| rounding to 2 ⁻¹⁵ | 0b01110000 | | 66 | | 32,265 | 0.13% | 40,427 | 9,573 | 80.854% | 19.146% |
| rounding to 2 ⁻⁷ | 0b01111000 | | 144 | | 7,123,565 | 27.82% | 38,390 | 11,610 | 76.780% | 23.220% |
| rounding to 2 ⁻³ | 0b01111100 | | 192 | | 25,368,406 | 99.06% | 48 | 49,952 | 0.096% | 99.904% |

Table 38. Effect of Exponent Bit Rounding on ShuffleNetV2_2.0 Accuracy

| ShuffleNetV2_2.0 | bit exponent mask | total tensors | modified tensors | total values | modified values | % | TRUE | FALSE | TRUE | FALSE |
|------------------------------|-------------------|---------------|------------------|--------------|-----------------|--------|--------|--------|---------|---------|
| GR | | | | | | | 38,097 | 11,903 | 76.194% | 23.806% |
| rounding to 2 ⁻³¹ | 0b01100000 | 282 | 73 | 7,427,712 | 8,917 | 0.12% | 38,097 | 11,903 | 76.194% | 23.806% |
| rounding to 2 ⁻¹⁵ | 0b01110000 | | 132 | | 18,013 | 0.24% | 38,100 | 11,900 | 76.200% | 23.800% |
| rounding to 2 ⁻⁷ | 0b01111000 | | 177 | | 1,423,909 | 19.17% | 38,009 | 11,991 | 76.018% | 23.982% |
| rounding to 2 ⁻³ | 0b01111100 | | 206 | | 7,222,430 | 97.24% | 41 | 49,959 | 0.082% | 99.918% |