UNIVERSITY OF CALIFORNIA SAN DIEGO

Algorithm-Centric Design of Reliable and Efficient Deep Learning Processing Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Elbruz Ozen

Committee in charge:

Professor Alex Orailoglu, Chair
Professor Chung-Kuan Cheng
Professor Sicun Gao
Professor Farinaz Koushanfar
Professor Tajana Rosing

2023

The dissertation of Elbruz Ozen is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

Dedicated to my parents, whose unwavering support and encouragement made this achievement possible, and to everyone who has guided and inspired me along the way.

EPIGRAPH

"The one who knows all the answers has not been asked all the questions."

– *Confucius*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Alex Orailoglu for his tremendous support as the chair of my committee. His guidance and efforts have been truly invaluable in the preparation of this dissertation and all associated publications.

The following chapters are re-organized re-prints of previous publications:

Chapter 6 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Low-Cost Error Detection in Deep Neural Network Accelerators with Linear Algorithmic Checksums," *Journal of Electronic Testing (JETTA)*, vol. 36, no. 6, pp. 703–718, 2020 ([1]). The initial conference version of this manuscript is published in Elbruz Ozen and Alex Orailoglu, "Sanity-Check: Boosting the Reliability of Safety-Critical Deep Neural Network Applications," in *Proceedings of the 28th Asian Test Symposium (ATS)*, pp. 7–12, IEEE, 2019 ([2]). The dissertation author was the primary investigator and author of both papers.

Chapter 7 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 169–174, IEEE, 2020 ([3]). The dissertation author was the primary investigator and author of this paper.

Chapter 8 contains re-organized reprints of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Shaping Resilient AI Hardware Through DNN Computational Feature Exploitation," *IEEE Design & Test (D&T)*, vol. 40, no. 2, pp. 59–66, 2023 ([4]), Elbruz Ozen and Alex Orailoglu, "Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3250–3262, 2020 ([5]), and Elbruz Ozen and Alex Orailoglu, "Just Say Zero: Containing Critical Bit-Error Propagation in Deep Neural Networks with Anomalous Feature Suppression," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, IEEE/ACM, 2020 ([6]). The dissertation author was the primary investigator and author of all three papers.

Chapter 9 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021 ([7]). The dissertation author was the primary investigator and author of this paper.

Chapter 10 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022 ([8]). The dissertation author was the primary investigator and author of this paper.

Chapter 11 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Squeezing Correlated Neurons for Resource-Efficient Deep Neural Networks," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD) - Part II*, pp. 52–68, Springer, 2021 ([9]). The dissertation author was the primary investigator and author of this paper.

Chapter 12 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Unleashing the Potential of Sparse DNNs Through Synergistic Hardware-Sparsity Co-Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 4, pp. 1147–1160, 2023 ([10]). The initial conference version of this manuscript is published in Elbruz Ozen and Alex Orailoglu, "Evolving Complementary Sparsity Patterns for Hardware-Friendly Inference of Sparse DNNs," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE/ACM, 2021 ([11]). The dissertation author was the primary investigator and author of both papers.

The following chapters partially contain material from previous publications:

Chapter 1 partially contains material from Elbruz Ozen and Alex Orailoglu, "Low-Cost Error Detection in Deep Neural Network Accelerators with Linear Algorithmic Checksums," *Journal of Electronic Testing (JETTA)*, vol. 36, no. 6, pp. 703–718, 2020 ([1]), Elbruz

Ozen and Alex Orailoglu, "Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 169–174, IEEE, 2020 ([3]), Elbruz Ozen and Alex Orailoglu, "Shaping Resilient AI Hardware Through DNN Computational Feature Exploitation," *IEEE Design & Test (D&T)*, vol. 40, no. 2, pp. 59–66, 2023 ([4]), Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021 ([7]), and Elbruz Ozen and Alex Orailoglu, "Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022 ([8]).

Chapter 2 partially contains material from Elbruz Ozen and Alex Orailoglu, "Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 169–174, IEEE, 2020 ([3]), Elbruz Ozen and Alex Orailoglu, "Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3250–3262, 2020 ([5]), Elbruz Ozen and Alex Orailoglu, "Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022 ([8]), and Elbruz Ozen and Alex Orailoglu, "Unleashing the Potential of Sparse DNNs Through Synergistic Hardware-Sparsity Co-Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 4, pp. 1147–1160, 2023 ([10]).

Chapter 3 partially contains material from Elbruz Ozen and Alex Orailoglu, "Low-Cost Error Detection in Deep Neural Network Accelerators with Linear Algorithmic Checksums," *Journal of Electronic Testing (JETTA)*, vol. 36, no. 6, pp. 703–718, 2020 ([1]), Elbruz Ozen and Alex Orailoglu, "Shaping Resilient AI Hardware Through DNN Computational Feature

Exploitation," *IEEE Design & Test (D&T)*, vol. 40, no. 2, pp. 59–66, 2023 ([4]), Elbruz Ozen and Alex Orailoglu, "Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3250–3262, 2020 ([5]), Elbruz Ozen and Alex Orailoglu, "Just Say Zero: Containing Critical Bit-Error Propagation in Deep Neural Networks with Anomalous Feature Suppression," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, IEEE/ACM, 2020 ([6]), Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021 ([7]), Elbruz Ozen and Alex Orailoglu, "Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022 ([8]), Elbruz Ozen and Alex Orailoglu, "Squeezing Correlated Neurons for Resource-Efficient Deep Neural Networks," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD) - Part II*, pp. 52–68, Springer, 2021 ([9]), and Elbruz Ozen and Alex Orailoglu, "Unleashing the Potential of Sparse DNNs Through Synergistic Hardware-Sparsity Co-Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 4, pp. 1147–1160, 2023 ([10]).

Chapter 13 partially contains material from Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021 ([7]).

VITA

2017	Bachelor of Science in Electrical and Electronics Engineering, İhsan Doğramacı Bilkent University
2020	Master of Science in Computer Science (Computer Engineering), University of California San Diego
2023	Doctor of Philosophy in Computer Science (Computer Engineering), University of California San Diego

PUBLICATIONS

Elbruz Ozen and Alex Orailoglu, "Unleashing the Potential of Sparse DNNs Through Synergistic Hardware-Sparsity Co-Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 4, pp. 1147–1160, 2023.

Elbruz Ozen and Alex Orailoglu, "Shaping Resilient AI Hardware Through DNN Computational Feature Exploitation," *IEEE Design & Test (D&T)*, vol. 40, no. 2, pp. 59–66, 2023.

Elbruz Ozen and Alex Orailoglu, "Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022.

Elbruz Ozen and Alex Orailoglu, "Evolving Complementary Sparsity Patterns for Hardware-Friendly Inference of Sparse DNNs," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE/ACM, 2021.

Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021.

Elbruz Ozen and Alex Orailoglu, "Squeezing Correlated Neurons for Resource-Efficient Deep Neural Networks," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD) - Part II*, pp. 52–68, Springer, 2021.

Elbruz Ozen and Alex Orailoglu, "Low-Cost Error Detection in Deep Neural Network Accelerators with Linear Algorithmic Checksums," *Journal of Electronic Testing (JETTA)*, vol. 36, no. 6, pp. 703–718, 2020.

Elbruz Ozen and Alex Orailoglu, "Just Say Zero: Containing Critical Bit-Error Propagation in Deep Neural Networks with Anomalous Feature Suppression," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, IEEE/ACM, 2020.

Elbruz Ozen and Alex Orailoglu, "Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3250–3262, 2020.

Elbruz Ozen and Alex Orailoglu, "Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 169–174, IEEE, 2020.

Elbruz Ozen and Alex Orailoglu, "Sanity-Check: Boosting the Reliability of Safety-Critical Deep Neural Network Applications," in *Proceedings of the 28th Asian Test Symposium (ATS)*, pp. 7–12, IEEE, 2019.

Elbruz Ozen and Alex Orailoglu, "The Return of Power Gating: Smart Leakage Energy Reductions in Modern Out-of-Order Processor Architectures", in *Proceedings of the International Conference on Architecture of Computing Systems (ARCS)*, pp. 253–266, Springer, 2019.

ABSTRACT OF THE DISSERTATION

Algorithm-Centric Design of Reliable and Efficient Deep Learning Processing Systems

by

Elbruz Ozen

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2023

Professor Alex Orailoglu, Chair

Artificial intelligence techniques driven by deep learning have experienced significant advancements in the past decade. The usage of deep learning methods has increased dramatically in practical application domains such as autonomous driving, healthcare, and robotics, where the utmost hardware resource efficiency, as well as strict hardware safety and reliability requirements, are often imposed. The increasing computational cost of deep learning models has been traditionally tackled through model compression and domain-specific accelerator design. As the cost of conventional fault tolerance methods is often prohibitive in consumer electronics, the question of functional safety and reliability for deep learning hardware is still in its infancy. This dissertation outlines a novel approach to deliver dramatic boosts in hardware safety, reliability, and resource

efficiency through a synergistic co-design paradigm. We first observe and make use of the unique algorithmic characteristics of deep neural networks, including plasticity in the design process, resiliency to small numerical perturbations, and their inherent redundancy, as well as the unique micro-architectural properties of deep learning accelerators such as regularity. The advocated approach is accomplished by reshaping deep neural networks, enhancing deep neural network accelerators strategically, prioritizing the overall functional correctness, and minimizing the associated costs through the statistical nature of deep neural networks. To illustrate, our analysis demonstrates that deep neural networks equipped with the proposed techniques can maintain accuracy gracefully, even at extreme rates of hardware errors. As a result, the described methodology can embed strong safety and reliability characteristics in mission-critical deep learning applications at a negligible cost. The proposed approach further offers a promising avenue for handling the micro-architectural challenges of deep neural network accelerators and boosting resource efficiency through the synergistic co-design of deep neural networks and hardware micro-architectures.

Chapter 1

Introduction

1.1 Background

Deep learning techniques have revolutionized application design by providing an alternative paradigm that eliminates the need for manual software development steps [12] and offering capabilities for electronic systems that were almost unimaginable in the past decade.

The recent past has witnessed remarkable advancements in various deep learning tasks such as computer vision [13, 14, 15], speech recognition [16, 17], and natural language processing [18, 19]. Models such as ChatGPT [20] have demonstrated significant progress in text generation tasks and engaging in high-quality dialogue with humans. DALL·E 2 [21] can generate realistic images and art for the given text descriptions. Many driving assistance and FSD (full self-driving) systems heavily rely on deep learning methods [22, 23]. Deep learning has numerous practical application areas in healthcare, ranging from medical imaging to robotic-assisted surgery [24].

The outstanding success of deep learning techniques has resulted in a wide range of applications in practical domains, including but not limited to autonomous driving, healthcare, robotics, defense, and industrial automation, where intelligent systems have become an integral part of our infrastructure. Meanwhile, the practical requirements for such systems are not limited to mere algorithmic accuracy but also involve the satisfaction of strict hardware constraints such as safety, reliability, and resource efficiency.

1.2 Problem Definition

It is widely recognized that DNNs (deep neural networks) incur high computational costs. Meanwhile, the continuing trend of increasing model sizes [25] is assumed to be a significant contributor behind the success of modern deep learning architectures. Numerous techniques, ranging from algorithmic model optimizations to hardware accelerator design, have been investigated in an effort to increase the resource efficiency of deep learning methods [26].

On the algorithmic side, the increasing computational cost of deep neural networks has been tackled mainly through the design of more efficient deep learning models [15, 27] and model compression methods such as pruning [28, 29, 30] and quantization [31, 32, 33, 34].

The increasing computational cost of DNNs has ignited further efforts to design hardware accelerators [26] that can deliver efficiency in the required operations. Spatial architectures such as systolic arrays are commonly used in recent DNN accelerators in industry [35] and academia [36, 37]. These architectures consist of regular tiles of PEs (processing elements) and perform tensor operations in a distributed manner by transferring variables locally among the neighboring units. The share of deep learning hardware accelerators is anticipated to grow noticeably in the semiconductor domain [38], and these architectures are expected to power billions of embedded artificial intelligence devices in the next decade.

Despite the significant efforts of the past decade, the computational cost of deep neural networks remains an important design consideration. *While isolated algorithmic and hardware optimization is capable of delivering noticeable gains in terms of efficiency, more algorithm-centric and synergistic design paradigms could offer tremendous potential for boosting resource efficiency in deep learning hardware systems.*

Both performance and energy efficiency constitute fundamental design considerations for embedded deep learning applications; strict functional safety constraints are frequently imposed furthermore in various practical application domains such as autonomous driving, healthcare devices, robotics, and industrial control systems. While operating in the field, exposure to harsh

environmental conditions (e.g., high-energy particles, high temperature) over long durations increases the likelihood of hardware errors in these systems considerably. A hardware-caused error can impact functionality with potentially catastrophic consequences; thus, additional electronics safety mechanisms are an absolute requirement in these application domains. Moreover, the rise of machine intelligence systems has coincided with the recent seismic shifts in semiconductor manufacturing technology. The benefits delivered by Moore's Law [39] are expected to come to an end soon, and we have started to witness more pronounced variability between devices, heightened reliability issues such as aging effects, and increased yield loss with every new generation of semiconductor technology nodes [40, 41, 42, 43, 44].

In the context of exploring the safety of DNNs against naturally occurring [45] or adversarial [46, 47] input perturbations, the design of a robust DNN algorithm constitutes but a single aspect of a complex equation; the safety of the underlying hardware is of matching importance in ensuring an entirely trustable system. The tolerance of DNNs to noise and small numerical inaccuracies has been argued, but we are still quite a ways from figuring out the precise extent of tolerance due to their non-linear nature [48]. Even under the presumption of robustness to small numerical inaccuracies, hardware-level faults could pose a significant problem as they might result in relatively large numerical deviations up to the order of a few magnitudes larger than the original data if the significant bit positions are affected. Hardware-level faults can thus diminish accuracy to unacceptable levels, and even single-bit error incidents might compromise system safety by leading to unexpected network decisions.

We perform an experimental analysis on AlexNet [49] (trained on the GTSRB dataset [50]) to demonstrate the issue by injecting bit errors into the activation values and the filter/weight coefficients, then measuring the expected accuracy of the network as a function of the error rate to determine the point where the DNN model experiences a noticeable accuracy drop. Figure 1.1 shows that the accuracy of the network exhibits a relatively sharp degradation at a specific error rate, and error rates surpassing this threshold noticeably distort the network decisions.

Second, we conduct another series of experiments, inspired by [47], in which we inject a

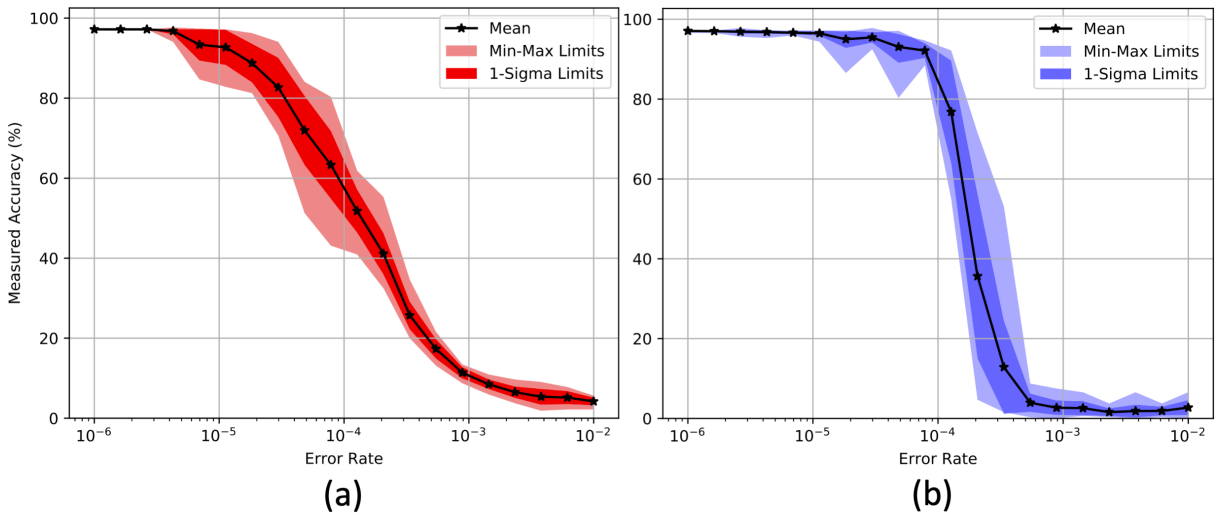

Figure 1.1. DNN accuracy drop under (a) activation (b) weight bit-errors.

single bit-error into the activation values and observe the cases where the neural network changes its decision. The signs pictured in Figure 1.2 have been misinterpreted to the errant commands noted on the caption above the traffic signs, clearly illustrating the havoc even single-bit errors can wreak in deep neural network outputs. In other words, DNNs demonstrate only a limited error-tolerant behavior as even a single-error occurrence may lead to unexpected decisions in DNN applications. An error detection method is consequently needed to preclude unsafe system decisions caused by possible misinterpretations stemming from such fault manifestations.

The safety-critical domains, such as the automotive industry, have been a challenging market for electronics and software designers because of extreme safety requirements [51]. As an unforeseen development can threaten human lives, automotive electronics are enhanced with strict safety features to withstand such eventualities. The state-of-the-art safety features in automotive electronics are comprised of the widely used ECC (error correction codes) to prevent SDCs (silent data corruptions) [52], and full redundancy (e.g., dual-core lockstep, or TMR (triple modular redundancy)[53]) to safeguard the execution path. Circuit-level hardening methods [54] do exist to detect and mitigate the effects of transient SEUs (single event upsets) or timing errors, but their notable area, delay, and power overheads make them a less appealing solution for consumer products.

Figure 1.2. Misprediction examples caused by single-bit errors.

Hardware safety and reliability have been studied widely since the dawn of electronic systems, and assurance of reliable operation has been an essential design consideration in numerous mission-critical application domains [55]. Meanwhile, conventional fault tolerance techniques in mission-critical electronics often interpret the correctness requirements in a rigid and structural fashion that is necessary in the context of general-purpose computing, yet the subsequent excessive costs and overheads restrict the wide-range applicability of these methods in resource-intensive and cost-sensitive commercial deep learning hardware systems. The wide-range adaptation of machine intelligence methods in consumer application domains with strict cost constraints, such as autonomous driving, has thus forced the investigation of techniques that can provide operational assurance in machine intelligence hardware at a negligible cost.

Moreover, significant efficiency and performance gains can be attained in artificial intelligence hardware if the enhanced resilience characteristics of deep neural networks are combined with aggressive hardware optimizations in CMOS (complementary metal-oxide-semiconductor) hardware devices. The outlined approach could further facilitate the adaptation of artificial intelligence devices constructed through emerging device technologies with the potential to deliver efficiency levels up to a few magnitudes higher than conventional digital computing, yet limited by the inherent imprecision and poor manufacturability problems. The outlined hardware efficiency pursuit requires innovative avenues for handling unprecedented error rates so that deep neural networks can operate accurately even under chaotic hardware conditions.

Fault tolerance is a challenging problem in the context of general-purpose computing,

as imposed safety mechanisms often remain agnostic to the possible resilience characteristics of the application. As a result, a worst-case approach ascribes even a minute change in the program variables to potential data corruption, necessitating the taking of further actions, such as correction or program re-execution, to assure the correctness of system behavior.

What makes this investigation for novel fault tolerance promising is the inherent resilience of neural networks to minor perturbations together with the learning flexibility of deep models even when constricted by imposed constraints. This flexibility of neural networks affords the construction of novel error identification mechanisms by shaping the inherent redundancy of deep learning algorithms. Moreover, the resilience of neural networks to minor perturbations opens up opportunities for approximate error mitigation without having to pay for perfect value restoration.

Algorithm-centric and synergistic design paradigms in this dissertation can offer avenues for interpreting the problem of hardware safety and reliability in a more functional manner, incorporating unique algorithmic characteristics of deep neural networks into the picture, thus enabling us to embed strong safety, reliability, and further efficiency characteristics into deep learning hardware systems often at negligible costs and overheads.

1.3 Dissertation Contribution

An effective approach to the outlined hardware challenges necessitates a holistic consideration of the hardware fabrics as well as the computational characteristics of deep learning algorithms to glean insights that can be harnessed for innovative solutions. The dissertation explores novel *algorithm-centric* and *synergistic* co-design techniques for converting unique algorithmic characteristics of deep neural networks, including *plasticity*, *resiliency*, and *redundancy*, as well as the hardware micro-architectural properties such as *regularity*, into significant boosts in safety, reliability, and resource efficiency to address challenging problems of hardware platforms used in artificial intelligence.

1.4 Dissertation Organization

Chapter 2 presents a brief overview of related deep neural network concepts.

Chapter 3 provides a comprehensive review of relevant studies in the prior literature. We aim to describe the current research progress in the corresponding domains and the outlined studies contribute to the shared understanding in the literature.

Chapter 4 presents a high-level overview and motivates the unique perspectives that are explored in this dissertation.

Chapter 5 outlines the explored research questions, introduces the unique characteristics of deep neural networks and the proposed research approach, and finally summarizes our progress within the scope of this dissertation.

The detailed technical discussion in this dissertation consists of **Chapters 6-12**:

Chapter 6 presents an algorithmic method for the detection of hardware datapath errors in deep neural network accelerators through the use of the innate mathematical properties of deep neural network layers such as linearity.

Chapter 7 demonstrates how the learning process can be harnessed to embed computational invariants into deep neural networks in training. Such invariants are utilized for detecting hardware datapath errors in deep neural network accelerators, even across the non-linear stages of the computations.

Chapter 8 presents an alternative methodology to integrate fine-grained computational invariants into deep neural networks for highly precise hardware datapath error localization. The proposed error detection approach is coupled with novel and cost-effective error rectification techniques to allow algorithmic self-checking and correction of hardware errors in deep neural network accelerators.

Chapter 9 observes the inherent resiliency of deep neural networks to small numerical perturbations and utilizes such resiliency characteristics for boosting the reliability of deep learning accelerators by restricting the numerical range in hardware and reshaping the numerical

distribution of deep neural network layers during the training process.

Chapter 10 outlines a co-design methodology for enabling customized and cost-effective adaptation against permanent hardware defects in deep neural network accelerators through the synergistic design of the hardware platforms and decentralized deep neural network algorithms.

Chapter 11 examines the significant output correlations among computational units within a deep neural network layer for quantitative redundancy characterization. We propose a novel deep neural network layer reduction and reconstruction process to obtain more compact and resource-efficient deep neural network architectures.

Chapter 12 focuses on a unique hardware/software co-design methodology for boosting the synergy between sparsity patterns and hardware platforms. We demonstrate unique opportunities for performance and resource efficiency improvements in sparse deep neural network inference without suffering the micro-architectural problems posed by the irregular nature of unstructured sparsity.

Chapter 13 incorporates a concise summary of the technical chapters and a detailed discussion about the significance of the technical results. Furthermore, we present our viewpoints regarding the open questions and potential directions in this research domain.

Chapter 14 presents the conclusion statements of the dissertation.

1.5 Acknowledgements

Chapter 1 partially contains material from Elbruz Ozen and Alex Orailoglu, "Low-Cost Error Detection in Deep Neural Network Accelerators with Linear Algorithmic Checksums," *Journal of Electronic Testing (JETTA)*, vol. 36, no. 6, pp. 703–718, 2020 ([1]), Elbruz Ozen and Alex Orailoglu, "Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 169–174, IEEE, 2020 ([3]), Elbruz Ozen and Alex Orailoglu, "Shaping Resilient AI Hardware Through DNN Computational Feature Exploitation,"

IEEE Design & Test (D&T), vol. 40, no. 2, pp. 59–66, 2023 ([4]), Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021 ([7]), and Elbruz Ozen and Alex Orailoglu, "Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022 ([8]).

Chapter 2

An Overview of Relevant Deep Neural Network Concepts

The central computation unit in a DNN (deep neural network) is called a *neuron*, whose responsibility is taking a weighted sum of its inputs and processing the sum with a non-linear activation function. The neurons are organized as a sequence of layers, with the network getting deeper as the number of layers increases. Modern CNNs (convolutional neural networks) used in computer vision tasks [13, 49, 56, 57] heavily rely on two layer types, namely, *convolutional* and *fully-connected*, where the input is initially processed by a series of convolution layers to extract the useful features, with the fully-connected layers subsequently performing the final classification task. Moreover, deep neural networks frequently employ non-linear activation functions and various other layer types, including pooling, batch normalization, Dropout, and Dropconnect layers [26].

2.1 Common Layer Types in Deep Neural Networks

2.1.1 Fully Connected Layer

Fully connected layers carry out a vector-matrix multiplication operation. Layer inputs could be considered as individual vectors. Fully connected layer weights are represented in the form of a matrix where each matrix column is associated with an individual neuron. A single element in the input vector is called an *input feature*. Fully connected layers multiply input

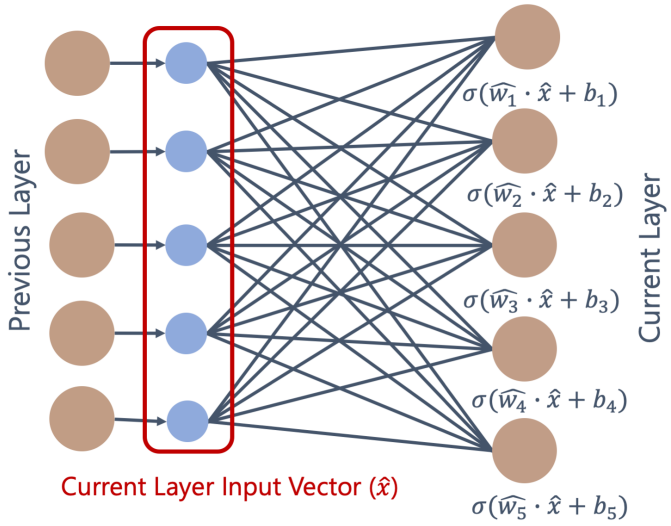

Figure 2.1. Fully connected layer operation.

vectors with the weight matrix, include the bias vector, then process the result with the non-linear activation function to generate the layer outputs, as demonstrated in Figure 2.1.

2.1.2 Convolutional Layer

Unlike the fully-connected layers, convolutional layer neurons are locally connected to the previous layer, and the weight values are shared among different neurons. As a result, the layer behavior could be visualized more naturally with a convolution operation. The input feature map is convolved with several filters where each of them is responsible for producing a channel of the output feature map, as in Figure 2.2. The convolutional layer operation further involves the inclusion of the bias, and the final processing of the result by a non-linear activation function.

2.1.3 Non-linear Activation Functions

Non-linear activation functions are frequently used after fully connected and convolution layers, and they are essential for the complex non-linear behavior of deep neural networks. Typical choices for the non-linear activation functions include widely-used *ReLU (rectified linear unit)*, *leaky ReLU*, *sigmoid*, and *tanh (hyperbolic tangent)*. Further information on non-linear activation functions can be found in [26].

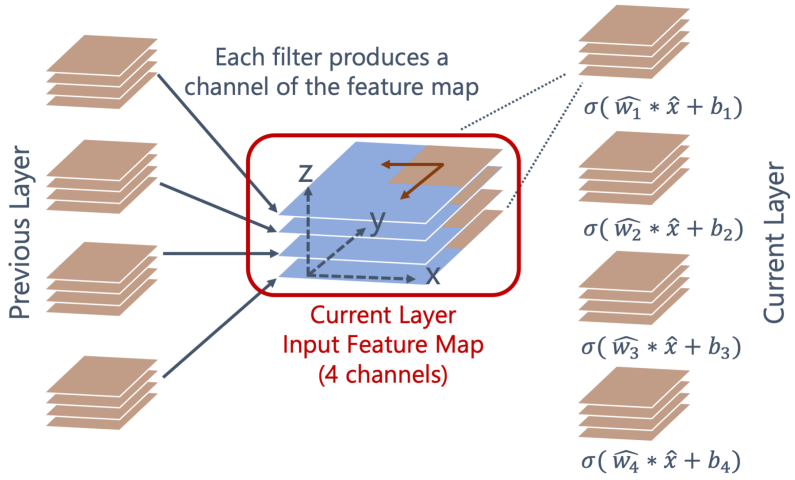

Figure 2.2. Convolutional layer operation.

2.1.4 Pooling

Pooling is used in CNNs to reduce the dimensions of the feature maps. Pooling layers perform a down-sampling operation by dividing the feature maps into small windows (e.g., 2×2 window) and reducing the window into a single entry. The typically used reduction operators include selecting the maximum entry (i.e., max pooling) or computing the mean (i.e., mean pooling) at each window.

2.1.5 Batch Normalization

Batch normalization [58] is commonly used in deep neural networks to reduce internal covariate shift and accelerate model convergence in the training process. Batch normalization adjusts the outputs of each neuron or convolution filter in training by normalizing the outputs with their mini-batch statistics (μ_B, σ_B^2), then scaling and shifting the normalized results with two learned coefficients σ_L and μ_L, as in Equation (2.1). A small coefficient (ε) is used to assure numerical stability.

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \times \sigma_L + \mu_L \tag{2.1}$$

A similar normalization step is performed at inference time; however, mini-batch statistics are replaced by the static values of global mean and variance that are profiled in training.

2.1.6 Dropout and Dropconnect

Deep neural networks could overfit into training data and exhibit poor generalization to test examples upon deployment. Dropout [59] and Dropconnect [60] have been previously investigated, and their effectiveness has been established for mitigating the overfitting problem. The presence of a neuron under *Dropout* is subject to a predetermined probability, with the outputs of absent neurons being set to zero in a training iteration. All neurons are considered to be present at test time, yet their output contribution is re-adjusted to match the training distribution by scaling either the current layer's outputs or the next layer's weights. *Dropconnect* drops individual weights instead of neuron outputs by subjecting the presence of each weight to the given probability in the applied layer. The training distribution can be maintained at test time through a sampling process that is similar to the training phase or utilizing a scaled version of the weights with no dropping.

2.2 Loss Function and DNN Training Process

The training procedure involves a *loss function*, which measures the distance between the expected and produced DNN outputs. To illustrate, a commonly used loss function for single-label classification problems, categorical cross-entropy, can be expressed as follows for a batch of training examples:

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\, log(p_{i,c}) \tag{2.2}$$

In Equation (2.2), N and C denote the training batch size and the total number of classes. The true binary label is represented as $y_{i,c}$ for the example i and class c. For single-label classification, $y_{i,c} = 1$ holds only for a single c for a particular example i. $p_{i,c}$ denotes the

predicted output probability produced by the output layer (with *Softmax* activation) of the network. Training is carried out by calculating the gradient of deep neural network weights through the backpropagation algorithm and updating the weights of each layer to reduce the loss at each step until a minimal point is found.

2.3 DNN Inference with Spatial Hardware Accelerators

The analysis in [26, 61] classifies deep learning accelerators into two primary categories. The first group, *temporal architectures*, includes designs with vector-type instructions such as CPUs (central processing units) and GPUs (graphics processing units). The second group, *spatial architectures*, relies on distributed dataflow processing through a large number of processing elements.

This section will focus on a typical example of spatial deep neural network accelerators, such as systolic arrays, as they will be used for analysis in the later technical chapters. Systolic arrays are widely adapted in practice to improve the inference performance and efficiency of deep neural networks [35, 36]. A systolic array consists of a 2-dimensional grid of MAC (multiply-accumulate) units as in Figure 2.3 where the weights of a single neuron or filter are mapped into a single column. When an input activation vector is provided to a column, each MAC unit acquires the accumulated sum from the neighboring MAC unit, multiplies the provided input activation with the stored weight, updates the accumulated sum, and forwards the updated sum to the next neighboring unit. The transfer of the sums allows a dot product to be computed at each column, which translates into a vector-matrix multiplication in the entire grid.

The outlined design can be classified as a *weight-stationary* architecture that maximizes weight reuse by pinning weights into processing elements during computation. Inputs and outputs are transferred through broadcasting or local communication across units. Google's TPU [35] is a well-known example of a weight-stationary architecture.

Alternative DNN accelerator dataflows discussed in [26, 62, 63] differ from weight-

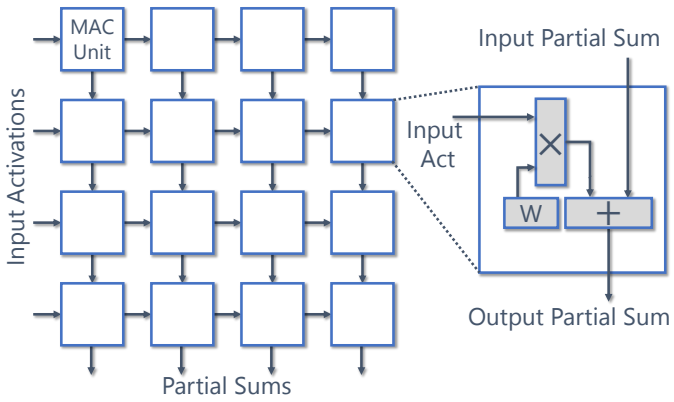

Figure 2.3. Systolic array deep learning accelerator.

stationary designs based on the particularities of the data reuse type. To illustrate, *input-stationary* dataflow maximizes input reuse by tiling and pinning layer inputs into the processing elements. Layer weights and partial sums are transferred across processing elements, and the processing element design is often identical to the one used for the weight-stationary dataflow. While layer inputs and weights are transferred across processing elements in the *output-stationary* dataflow, the partial sum belonging to an output location is pinned to and updated within the same processing element.

2.4 Acknowledgements

Chapter 2 partially contains material from Elbruz Ozen and Alex Orailoglu, “Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions,” in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 169–174, IEEE, 2020 ([3]), Elbruz Ozen and Alex Orailoglu, “Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3250–3262, 2020 ([5]), Elbruz Ozen and Alex Orailoglu, “Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022 ([8]), and

Elbruz Ozen and Alex Orailoglu, “Unleashing the Potential of Sparse DNNs Through Synergistic Hardware-Sparsity Co-Design,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 4, pp. 1147–1160, 2023 ([10]).

Chapter 3

Literature Review

This chapter provides a comprehensive overview of the relevant research literature. We start with the summary of related studies in the domain of safety, reliability, and testing of deep learning hardware. The second part of this chapter will discuss various techniques to improve the hardware performance and efficiency of DNNs (deep neural networks). Finally, we overview a list of studies that focus on deep learning inference in computational mediums with high fault rates.

3.1 Safety, Reliability, and Testing of Deep Learning Hardware

A significant amount of work has been presented in the recent literature regarding the safety, reliability, and testing of deep learning hardware. We start by investigating the techniques to evaluate the reliability of deep neural networks, and then examine methods to detect and correct hardware errors in deep learning hardware. Furthermore, we look into various algorithmic measures for boosting the error resilience of deep neural networks. Finally, we survey a list of studies that consider hardware reliability during the resource scheduling process, and then we explore innovative approaches for testing and yield improvement in deep learning accelerators.

The reader could refer to [64] for a detailed survey of the challenges and the current trends in robust machine learning systems.

3.1.1 Evaluating Reliability

Evaluating reliability is a crucial task for identifying hardware vulnerabilities and guiding effective reliability improvement techniques for deep learning hardware. The previous studies have explored several different avenues for this objective:

1. Graph-level and hardware-level simulation techniques have been proposed to measure the impact of hardware errors on deep neural network accuracy.

2. High-energy beams are harnessed for injecting physical errors into semiconductor chips running deep learning algorithms.

3. Various numerical analysis methods are utilized for estimating the vulnerable portions of deep neural network computations.

Graph-level Fault Simulation

Graph-level error simulation techniques have become a commonly used approach for deep neural network reliability analysis. Graph-level simulation frameworks avoid the high runtime costs of hardware-level simulations by modeling hardware error effects on the computational graph of neural networks during the inference process. On the other hand, graph-level error simulation methods require accurate modeling of the hardware error effects, which involves identifying the impacted neural network variables and the numerical perturbation values associated with the hardware errors. As a result, this simulation technique could suffer from inaccurate conclusions regarding reliability if the hardware error effects are not accurately reflected on the computational graph of deep neural networks.

As a result of the outlined advantages, previous literature has witnessed a flurry of works exploring the error resilience limits of deep neural networks through graph-level error simulation [65, 66, 67, 68, 69, 70, 71, 72, 73, 74]. Various software frameworks have also been made publicly available [75, 76, 77] for graph-level fault simulation in deep neural networks.

Hardware-level Fault Simulation

Hardware-level fault simulation is considered an alternative approach to graph-level fault simulation in various studies [78, 79, 80, 81]. Hardware-level fault simulation (e.g., register-transfer-level or gate-level) offers higher precision in modeling the hardware architecture and the fault effects when compared to graph-level fault simulation. On the other hand, hardware-level fault simulation could incur excessive simulation runtime, as reported in [78]. Furthermore, hardware-level simulation frameworks necessitate setting up a hardware model in addition to the executed deep learning algorithm. The number of studies that utilize hardware-level simulation is thus limited compared to those that employ graph-level simulation given the excessive runtime and the complexity of setting up such frameworks. A limited number of studies, such as [82], complement hardware fault simulation with machine learning-assisted techniques for quick evaluation of structural fault criticality in deep neural network accelerators.

Physical Fault Injection

Physical fault injection experiments are rarely used to evaluate the reliability of deep neural network hardware when compared to fault simulation techniques, primarily due to the necessity of special equipment and the difficulty of setting up such experiments. A limited number of studies in the literature, such as [83] and [84], carry out neutron beam experiments to assess the reliability of deep learning algorithms on the GPU (graphics processing unit) devices.

Numerical Evaluation

The relative vulnerability of deep neural network variables can be estimated through mathematical saliency models, including gradient-based sensitivity analysis [85], the LRP (layer-wise relevance propagation) metric [86, 87], or other vulnerability analysis techniques [88] discussed in the literature.

Although such measures enable the researchers to quickly identify relatively more vulnerable portions of deep neural networks without necessitating an empirical evaluation of

the faults, these methods could often prove insufficient for the task of accurately estimating the overall resilience (e.g., classification accuracy) of the target deep neural network models.

3.1.2 Hardware Error Detection and Rectification

Hardware error detection and correction are widely-studied problems in fault-tolerant systems. While the capabilities for error detection and correction are usually introduced through a considerable amount of redundancy in conventional methods [55], the computational characteristics of deep learning hardware enable researchers to identify novel and cost-effective solutions.

Conventional Error Detection and Rectification Techniques

Error detection and correction is an extensively studied problem in general-purpose computing. Parity and ECC (error correction codes) [52, 89] are frequently utilized in safety-critical designs to detect and correct errors in memory units. Error correction codes are theoretically optimal in terms of information redundancy, yet they can not be readily applied to the execution path since the code invariants are not preserved through the arithmetic operations. Full redundancy such as DMR (dual modular redundancy) or TMR (triple modular redundancy) [53, 90] is employed in automotive designs to protect the execution units at the cost of a significant area and power consumption. Arithmetic codes [55] offer a cheaper alternative than full redundancy for error detection in the arithmetic units. The Razor flip-flop [91] is an efficient technique for timing error detection, but with limited applicability to other hardware error types.

Symptom-Based Error Detection

Li et al. [66] and Schorn et al. [92] propose symptom-based error detectors for DNN computations. Symptom-based detectors often rely on the expected distribution of neural network variables to identify anomalies. Li et al. [66] utilize the expected range of activation magnitudes to identify critical hardware errors with a large magnitude. Schorn et al. [92] make use of a

smaller secondary neural network model to distinguish between critical and non-critical bit-errors in the primary deep neural network.

Algorithm-Based Fault Tolerance

ABFT (Algorithm-Based Fault Tolerance) has been initially introduced for matrix multiplications in multi-processor systems [93, 94]. Algorithm-based fault tolerance techniques [95, 96] utilize the linearity property of multiply-accumulate operations to construct error detection checksums in the fully connected and convolutional layer computations.

Error Detection with Learned Algorithmic Invariants

Novel recent studies [97, 98] demonstrate that algorithmic consistency checks can be encoded in neural networks through the training process, and error detection can be achieved through additional checker neurons.

Approximate Error Rectification

Various novel studies have been proposed to rectify error effects in deep neural network computations. Promising hardware error rectification techniques include activation range restriction [99, 100, 101, 102], and dropping the contribution of erroneous variables [103, 104].

3.1.3 Boosting Hardware Error Resilience of Deep Neural Networks

Resilient Neural Architecture Design

Prior experimental work [71, 105] has demonstrated that compressed models, particularly binary neural networks, exhibit high error resilience characteristics. Fault-tolerant neural architecture search [106, 107] can be utilized to construct deep neural networks that are inherently resilient to hardware errors.

Fault-Aware Training

Fault-tolerant model training [108, 109] can boost the hardware error resilience of deep neural networks. Dropout and Dropconnect are utilized for improving the inference reliability and efficiency of SNNs (spiking neural networks) in [110, 111]. Similarly, a standard version of the Dropout technique is employed in [112] for improving the resilience of an output-stationary deep neural network accelerator.

3.1.4 Reliability-Aware Scheduling

Several studies [73, 85, 87] in the literature observe the asymmetric distribution of vulnerability in deep neural networks (i.e., certain neurons/filters predominating in decision criticality) and schedule the critical portions of deep neural network computations into more reliable processing elements.

The asymmetric reliability difference could stem from various reasons on hardware. Particular processing units are observed to operate faster than others due to process variations in [85], thus being less likely to experience timing errors. Similarly, a designer could prefer to harden a subset of processing units against single-event upsets (i.e., caused by high-energy particles) to reduce the design costs [73, 87]. These techniques often use numerical estimation measures such as in Section 3.1.1 to identify vulnerable deep neural network variables and computations.

Overall, reliability-aware scheduling minimizes the computational effect of hardware errors on deep neural network decisions by scheduling vulnerable portions of the computations into more reliable processing units.

3.1.5 Hardware Testing and Yield Improvement

Hardware Testing

The primary investigation efforts in hardware testing have focused on the applicability of structural and functional test techniques for deep learning hardware, and the use of deep

neural network hardware characteristics for test cost reduction. The comparison of structural vs. functional testing approaches for neuromorphic hardware is discussed in [113]. Various functional testing techniques for deep learning accelerators are presented in [114, 115]. Computational fabric particularities such as regularity are utilized to innovate the permanent defect testing techniques for deep learning accelerators [116].

Hardware Yield Improvement

The yield of deep neural network accelerators can be boosted by making use of redundant backup hardware units to replace defective ones, an approach similar to the conventional work on fault-tolerant systolic arrays [94]. To illustrate, hardware redundancy and backup computational units are utilized in [117] to carry out computations that map into the defective processing elements in a spatial deep learning accelerator.

Alternatively, the computational plasticity of deep neural networks enables the researchers to compensate for the impact of hardware loss through additional algorithmic measures. As a result, hardware architectures can be reused even in a degraded form without requiring additional hardware redundancy. The impact of permanent defects is tolerated in a weight-stationary systolic array deep learning accelerator through bypassing the faulty processing elements and compensating for the accuracy drop through device-specific model training (fine-tuning) in [79, 80]. A similar bypass mechanism is complemented with saliency-driven weight remapping and retraining in [118] to minimize accuracy loss due to permanent hardware defects. A detailed fault impact analysis in deep neural network accelerators and a coarser-grained software-level fault bypassing technique are presented in [119]. The proposed yield improvement techniques often necessitate device-specific model training, which limits the applicability of these methods in practice due to training costs that scale with the number of faulty devices.

3.2 Improving Performance and Efficiency of Deep Learning Inference

3.2.1 Model Compression

The over-parameterized and redundant nature of deep neural networks has led to various model compression techniques in the recent research literature. This section will focus on three active research domains: reducing the precision of deep neural network variables, sparsity introduction through model pruning, and tensor decomposition to re-construct deep neural network layers.

Reducing Precision

Deep neural network inference often does not require high precision for the variables to exhibit competitive accuracy. The cost of data movement and multiply-accumulate operations can thus be reduced drastically by reducing the precision of deep neural network variables [120].

Uniform quantization schemes [34] that allocate a uniform set of distances between the quantization points are relatively easier to implement on hardware, but they can suffer from significant quantization errors at small bit-widths. Learned quantization [31] can identify the optimal non-uniform allocation of quantization points that minimize the quantization error, yet the irregularity of quantization point distances necessitates look-up tables for restoring data from the quantized representation. An alternative log quantization scheme [121] can be achieved by allocating quantization point distances at a logarithmic scale. Log quantization does not require look-up tables and further eliminates the need for multiplication operations in deep neural network inference.

The previous studies demonstrate that deep neural networks can have varying degrees of toleration against aggressive levels of bit-width reduction [32]. Moreover, layers within a neural network can be quantized into different bit-widths to construct mixed-precision deep neural networks [122]. Precision-scalable multiply-accumulate unit architectures [123] are often

necessary to obtain significant power and performance benefits from mixed-precision quantized deep neural networks.

Model Pruning and Sparsity

Obtaining fast and efficient neural networks through model sparsity has been a decades-old interest [124]. While the fundamental approach to neural network parameter pruning has hardly changed over the recent decades, various weight pruning methods [28, 29, 125, 126, 127, 128, 129] with more accurate redundancy identification mechanisms have extended the compression limits while minimizing the consequent accuracy loss. Unstructured parameter pruning often delivers significant compression rates, yet it further necessitates sparse compressed representations and sparse matrix algorithms to extract the potential benefits of unstructured sparsity.

Structured pruning algorithms [29, 30, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149] aim to reduce the number of parameters and operations without introducing additional sparsity in deep neural network models and pose a valuable alternative to deliver practical speedups on commodity hardware.

Structured pruning techniques utilize a variety of significance measures to rank neurons and filters, such as the properties of weight sets [30, 135, 145], or of layer outputs [131, 138]. Some of them utilize Taylor approximation [134, 143, 147] or other back-propagated metrics [144] to estimate the neuron filter significance. He et al. [139] utilize LASSO regression to sparsify feature map channels and perform pruning on the sparsified feature map channels to minimize the accuracy impact. Zhuang et al. [141] consider construction error as well as discriminative power in channel pruning. Xiao et al. [29] perform soft-pruning during the training process through the introduced auxiliary gate parameters in deep neural networks. A limited subset of the extant literature utilizes pairwise weight [146] and activation [132] correlations to carry out computation unit elimination. Mariet et al. [133] pick a subset of diverse neurons through Determinantal Point Processes [150], and perform weight adjustments in the next layer

to retain the magnitude contribution of the eliminated units. However, the analysis in [133] is limited to small models with fully connected layers only.

Liu et al. [151] demonstrate that training the reduced architectures from scratch can outperform the pruned and fine-tuned models if a large number of training iterations can be afforded.

Block-wise [152] and vector-wise [153] sparsity patterns are constructed at a finer granularity than neuron/filter-level sparsity and exhibit a structure more hardware friendly than unstructured sparsity, yet they often fail to deliver compression rates that match those of unstructured sparsity without leading to significant accuracy loss, as discussed in [154]. Balanced sparsity [154] and density bound block sparsity [155, 156, 157] can attain competitive compression rates through pruning, and facilitate efficient hardware utilization due to pre-determined group-wise density. Similar sparsity patterns, including group and exclusive sparsity, can be introduced through weight regularization [158].

Packing algorithms have been proposed in [159, 160] to map sparse neural network layers into dense representations. Packing algorithms first perform pruning on parameter tensors and then combine non-overlapping sparse tensor columns into a single dense representation to facilitate efficient hardware processing on minimally enhanced systolic array architectures.

Tensor Decomposition

An alternative model compression technique, tensor decomposition [161, 162, 163], allows low-rank approximation of a fully connected or convolutional layer weight tensor in the form of multiple yet smaller weight tensors. As a result, a fully connected or convolutional layer is effectively represented as a sequence of multiple smaller layers of the same type. Tensor decomposition provides significant compression in the layer size by reducing the number of parameters and multiply-accumulate operations through a static investigation of the weight set properties; however, the expansion of the layer into a sequence of multiple layers might be problematic in terms of inference latency in the commodity hardware platforms.

3.2.2 Hardware Accelerator Design

The computational demands of deep neural networks have resulted in the development of hardware accelerators. Hardware accelerators provide higher performance and resource efficiency when compared to general-purpose architectures such as CPUs (central processing units) since they can support a large amount of parallelism through dedicated computational resources. Furthermore, these architectures often capitalize on various data-reuse opportunities in deep neural network computations to minimize the amount of costly data transfers between the computational fabric and memory.

The taxonomy in [26] classifies deep neural network accelerators into four categories based on their dataflow patterns during the execution. Weight-stationary [35], input-stationary [164], and output-stationary [165] architectures maximize reuse by pinning the corresponding data type into processing elements during execution. An alternative row-stationary dataflow [166, 167] aims to maximize reuse of all data types. The reuse patterns of the listed dataflows offer unique advantages [26, 62], and certain deep neural network accelerator designs support even multiple dataflow configurations simultaneously to maximize the data reuse benefits [36].

Finally, deep learning accelerators are often designed to accommodate compressed neural networks obtained through the techniques outlined in Section 3.2.1. As deep neural network inference in the reduced precision fixed-point data format has become the de-facto standard across deep neural network inference accelerators in academia and industry [26], various accelerator designs provide support for the optimized storage and accelerated processing of sparse deep neural networks [164, 167, 168, 169].

3.3 Deep Learning Inference in Computational Mediums with High Fault Rates

3.3.1 Aggressive Hardware Optimizations in Digital CMOS Hardware

Aggressive hardware optimizations can significantly reduce the energy consumption of CMOS (complementary metal-oxide-semiconductor) deep learning accelerators if the consequent hardware errors (e.g., timing errors) can be gracefully tolerated by deep learning algorithms or handled through active error rectification methods.

Voltage under-scaling reduces the operating voltage to cut down energy consumption and thus improve the efficiency of deep learning accelerators. Zhang et al. [78] utilize Razor flip-flops [91] to detect timing errors, coupled with an architectural bypassing to block the propagation of bit-errors in voltage under-scaled deep neural network accelerators. Reagen et al. [103] employ Razor register file design [170] and error masking strategies to handle SRAM (static random-access memory) read errors in voltage-scaled deep neural network accelerator buffers. A similar SRAM supply voltage scaling is coupled with selective voltage boosting in [171] to maintain an acceptable level of accuracy in deep neural networks. The improved resilience characteristics of deep neural networks can be coupled with hardware optimizations such as reliability-aware adaptive voltage swing scaling in the chip communication circuitry [172].

An alternative approach, over-clocking, boosts the performance of the accelerator at a given operating voltage at the cost of additional timing errors. Pandey et al. [173] boost the accelerator performance by utilizing an adaptive voltage boosting scheme that makes use of Razor flip-flops [91] and an input sequence memorization technique to proactively prevent timing errors in an over-clocked systolic array deep neural network accelerator.

3.3.2 Alternative Computing Technologies

The homogeneity of deep neural network computations has forced researchers to reconsider the fundamental operation, multiply-accumulate, to obtain significant computational benefits in the inference process. Alternative computing technologies offer a promising avenue for the efficient processing of deep neural networks. Unlike conventional CMOS accelerators that transfer parameters between computational units and memory and utilize digital arithmetic circuits for computation, these devices perform multiply-accumulate operations through unique physical phenomena such as fundamental laws of electronics or photonics. Furthermore, these devices eliminate the need for costly transfer of parameters from the memory to the computational units since the weights are often programmed into the computational fabric. This section will look into two promising technologies for the efficient processing of deep neural networks.

ReRAM-based Computing

ReRAM (resistive random-access memory)-based deep learning accelerators perform deep neural network computations in the analog domain with great efficiency [174, 175, 176]. Deep neural network weights are programmed as resistance values within a crossbar of connected resistors, and multiply-accumulate operations are conducted through Kirchhoff's Current Law in these architectures.

Despite their immense benefits in theory, analog domain noise, device variations, and manufacturing problems often preclude the applicability of ReRAM-based accelerators in practical systems. As a result, the reliability of ReRAM-based deep learning accelerators has been widely investigated to improve the accuracy of deep neural networks under device manufacturing defects, programming variations, and analog domain noise.

Arithmetic codes [177] have been proposed in the literature to enable online error detection and correction, and enhance the resilience characteristics of the neural networks against ReRAM inaccuracies. Liu et al. [178] propose an alternative technique to boost the inherent self-correcting capability of DNN models through Error-Correcting Output Codes [179].

A detailed summary of the ReRAM fault models, the testing approaches, and fault-tolerant ReRAM hardware architectures are presented in [180]. Liu et al. [181] propose an extended ABFT (X-ABFT) method for online testing the faults in ReRAM crossbars. The impact of permanent manufacturing defects and resistance variations in ReRAM devices can be tackled through adaptive training methods [182, 183]. When paired with differential ReRAM crossbar mapping, neural network sparsity can be employed for muting the numerical impact of stuck-on and stuck-off faults in ReRAM devices [184].

The resilience of deep learning algorithms against ReRAM noise and variations can be enhanced by injecting noise perturbations in training and allowing deep neural networks to adapt to the noise effects during the training process [185, 186, 187, 188, 189].

Furthermore, ReRAM variation effects could cause layer output statistics to deviate from the batch normalization statistics profiled in training, and result in consequent accuracy loss due to inaccurate normalization steps performed at inference time. The re-calibration of batch normalization statistics on the ReRAM device could be particularly effective for improving the inference accuracy of deep neural networks under ReRAM variations [190].

Photonics-based Computing

The recent studies in photonics-based deep neural network accelerators [191, 192, 193, 194] prove that photonics-based devices can perform deep neural network computations with extreme efficiency. On the other hand, the existing hardware prototypes are often demonstrated for small-scale deep neural networks only, and the non-idealities in photonics-based devices, such as cross-talk issues and noise, need to be carefully considered to be able to build practical photonics-based devices [195].

3.4 Acknowledgements

Chapter 3 partially contains material from Elbruz Ozen and Alex Orailoglu, “Low-Cost Error Detection in Deep Neural Network Accelerators with Linear Algorithmic Checksums,”

Journal of Electronic Testing (JETTA), vol. 36, no. 6, pp. 703–718, 2020 ([1]), Elbruz Ozen and Alex Orailoglu, "Shaping Resilient AI Hardware Through DNN Computational Feature Exploitation," *IEEE Design & Test (D&T)*, vol. 40, no. 2, pp. 59–66, 2023 ([4]), Elbruz Ozen and Alex Orailoglu, "Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3250–3262, 2020 ([5]), Elbruz Ozen and Alex Orailoglu, "Just Say Zero: Containing Critical Bit-Error Propagation in Deep Neural Networks with Anomalous Feature Suppression," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, IEEE/ACM, 2020 ([6]), Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021 ([7]), Elbruz Ozen and Alex Orailoglu, "Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022 ([8]), Elbruz Ozen and Alex Orailoglu, "Squeezing Correlated Neurons for Resource-Efficient Deep Neural Networks," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD) - Part II*, pp. 52–68, Springer, 2021 ([9]), and Elbruz Ozen and Alex Orailoglu, "Unleashing the Potential of Sparse DNNs Through Synergistic Hardware-Sparsity Co-Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 4, pp. 1147–1160, 2023 ([10]).

Chapter 4

Dissertation Overview

The traditional approach to hardware safety and reliability has conventionally taken a precise direction that prioritizes hardware correctness on the structural level. The conventional tests and functional safety measures employed in semiconductor chips have often ensured the structural correctness of the logic gates and interconnects. On the other hand, the unique landscape of deep neural network hardware brings unique opportunities for approaching the old problems of safety, reliability, and efficiency. This section aims to demonstrate an alternative paradigm to reconsider these problems in the context of machine intelligence hardware and then provide a roadmap for leveraging these observations as a starting point for this dissertation.

The inherent resiliency characteristics and the statistical nature of deep neural networks could enable one to forgo strict correctness requirements at the hardware level and instead prioritize design techniques that can cultivate proactive hardware error toleration characteristics in deep neural network computations. Similar principles further spur non-perfect restoration techniques, which aim to contain the numerical impact of hardware errors without attempting the perfect restoration of the original value and deliver a more graceful accuracy degradation curve in the face of hardware errors. Furthermore, the cultivated relationships across deep neural network variables enable the possibility of embedding computational invariants that can be used for the effective detection and localization of errors into deep neural network hardware. The unique computational characteristics of deep neural networks, including plasticity and

redundancy, fuel the integration of internal invariants throughout the use of the training process. Overall, the novel perspective of looking into deep neural networks through relationships across variables is a powerful strategy in various domains. Some examples in this dissertation include the construction of novel sparsity patterns by harnessing the relationships across the neighboring variables to attain high expressivity in sparse deep neural networks and regularity in sparse hardware micro-architectures.

4.1 Reconsidering DNN Resilience Characteristics

Deep neural networks exhibit a resilient nature to minute perturbations in their variables. A typical compression technique, model quantization, could be considered as a widespread error introduction in the quantized deep neural network. On the other hand, the impact of a proper quantization process on accuracy is often observed to be minute despite its widespread application throughout the model. It is perhaps a non-surprising phenomenon when the inherent resilience characteristics of deep neural networks are considered. The numeric contribution of minor error effects can often be tolerated if they do not result in a significant numeric deviation in the functional outputs of a deep neural network.

One could subsequently question the necessity of worrying about hardware errors in deep neural network hardware. If deep neural networks are fully capable of tolerating hardware errors, does it mean that one could forgo any additional measures in the applications that necessitate strict hardware safety and reliability? Unfortunately, the answer to this question requires reconsideration of the nature of hardware errors as well. Unlike the well-bounded influence of quantization, the impact of hardware errors in modern computing systems is not well-constrained. The binary number representation in modern computers leads to widely diverse significance differences among the bit positions in the representation, in which the most significant bit positions involve magnitudes of higher numeric importance when compared to their least significant counterparts. The impact of high-magnitude errors in such hardware

systems is bound to be significant; a few such errors are well capable of leading to drastic changes in the functional neural network outputs.

It is further known that machine learning techniques, such as deep neural networks, exhibit a statistical nature, and thus the answers obtained from these algorithms are not always correct. As a result, a certain level of imprecision is accepted in machine intelligence applications as long as the functional accuracy remains within the desired bounds. This outlined attribute is helpful to our pursuit because it carves a path for a statistical approach to the problem of safety and reliability, where the desired levels of functional accuracy can be maintained at a drastically reduced cost if the statistical nature of such applications is taken into consideration.

In summary, the resilience of deep neural networks and their statistical nature are unlikely to provide assurance about safety and reliability, yet they promote an effective way of thinking and approaching these problems in the context of deep learning hardware. The outlined unique nature of deep neural networks motivates a fundamental question about addressing the safety and reliability problems even before the occurrence of hardware errors in the first place. Could we design deep learning systems in a way that the occurrence of hardware errors of a critical nature is less likely and the majority of hardware errors can be gracefully tolerated? The outlined proactive strategy constitutes a practical first line of defense and enables us to attain the desired safety and reliability objectives at a reasonable price tag.

4.2 Proactive Toleration of Hardware Errors

The unique nature of deep neural networks opens up possibilities for attenuating the impact of hardware errors in a proactive manner through novel design methodologies that form deep neural networks with boosted resilience characteristics. In this dissertation, we will explore two particularly fruitful directions for boosting the proactive error toleration characteristics of deep neural networks. First, we notice the close relationship between the numerical range and error vulnerability in deep neural networks in our technical analysis. Second, incorporating

proactive training strategies with the proper objectives could boost the level of decentralization in deep neural networks and furnish resilience against the loss of hardware components. *The proactive shaping strategies aim to minimize the impact of hardware errors by ensuring that no individual variable is excessively wrong or excessively important, and thus, an enhanced level of immunity against hardware errors can be enjoyed by deep neural networks.*

4.2.1 Vulnerability Reduction Through Range Manipulation

When the deep neural network variables under-utilize the numerical range offered by the hardware number representation, high-magnitude hardware errors could significantly dwarf the values represented in the distribution of deep neural network variables. As a result, an under-utilized numerical range could become a source of vulnerability against hardware errors.

An effective solution to this problem involves utilizing a hardware number representation that closely matches the distribution of variables and manipulating variable distributions further to make high-magnitude error manifestations even less likely. One could entertain unique visions, such as the proper constraint of specific design parameters to downsize the magnitude of these problems and contribute to the overall goal of proactive error resilience in deep neural networks.

4.2.2 Boosting Algorithmic Decentralization in Training

Training deep neural networks with noise has been a widely-adapted practice for various objectives, such as improving generalization or boosting the algorithmic resilience against noise at the inputs. The intelligent use of such training methodologies is a valuable asset for promoting proactive resilience in deep neural networks. Proper deep neural network training strategies could minimize the saliency of individual variables and cultivate a highly decentralized computational structure in deep neural networks when they are instructed through proper training objectives.

A particularly effective strategy in the technical part of this dissertation makes use of methods such as Dropout [59] and Dropconnect [60] to boost decentralization in the training process, reduce the saliency of individual variables, and facilitate graceful toleration for the

numerical loss of variables. The resilience benefits of decentralization could be promoted even further when the micro-architectural hardware parameters are considered in the training process. As a result of a training strategy that mimics the impact of hardware effects more precisely, the information degradation due to the loss of hardware components can be minimized, and deep neural networks can be empowered to operate accurately even on the computational hardware fabrics with downsized resources.

4.3 Non-Perfect Restoration of Hardware Errors

Proactive toleration techniques such as range manipulation and decentralization shrink the differences between neurons or convolution filters to form a decentralized computational structure. On the other hand, a certain amount of saliency difference among the neurons or convolution filters is needed to ensure overall expressivity. As a result, the impact of hardware errors in critical locations should be identified and addressed explicitly to boost the hardware error resilience of deep neural networks.

The conventional approach to safety and reliability often emphasizes the precise correction of hardware errors through various hardware or software measures, a pursuit that often necessitates excessive information redundancy and extensive computational resources. *Instead of precisely correcting the impact of hardware errors, an alternative perspective we introduce in this dissertation restores the erroneous variables to reasonable levels in a cost-effective manner and consequently minimizes the numerical effect of hardware errors on the functional outputs.*

The non-perfect restoration of hardware errors could be enacted by dropping erroneous variables, clipping their magnitude within a reasonable numerical range, or passing the variables through median filtering stages. Disproportionate hardware error effects can thus be tempered before they propagate and influence the outputs of deep neural networks. Such novel and non-perfect restoration techniques make use of the inherent resilience of deep neural networks into small numerical perturbations and harness their graceful toleration of computational sparsity.

In contrast to the sizeable numerical impact of critical hardware errors, the overall effect of non-perfect restoration techniques is highly muted and oftentimes preferable in maintaining deep neural network accuracy. The outlined strategy engenders a variety of powerful yet cost-effective non-perfect restoration techniques and revolutionizes our perspective on safety and reliability in the context of deep learning hardware.

4.4 Hardware Error Detection Through Invariants

Identifying hardware errors in fault-tolerant systems often necessitates considerable additional information redundancy embedded into the design. Furthermore, the conventional error detection methods are frequently constructed at the structural level, and they ensure the correctness of individual bits and variables, regardless of the impact of the error on the functional neural network outputs. *An alternative perspective brought to the table by this dissertation relies on the construction of relationships across neural network variables and it makes use of such relationships for the detection and localization of hardware errors of critical nature.*

The relationships across variables can be constructed at various granularities. The relationships of coarser nature, such as group-wise relationships across variables, can be built at a lower cost to identify the presence of hardware errors. On the other hand, finer-grained associations, such as neighbor-wise relationships, could facilitate the precise localization of hardware errors at a small additional cost.

Meanwhile, the construction of relationships can be carried out through various measures. This dissertation will focus on two particular strategies that utilize the inherent mathematical characteristics, such as linearity, to embed relationships externally, or harness the plasticity of deep neural networks during the training process to embed computational relationships internally.

4.4.1 Hardware Error Detection Through External Invariants

A significant portion of deep neural networks consists of highly homogeneous multiply-and-accumulate operations of a linear nature. The mathematical characterization of linear

systems is rather straightforward as they retain the linear relationships of their inputs at the system outputs. As a result, the maintained linear relationships could help identify abnormal system behavior and construct hardware error detection measures. External invariants can often be introduced at the cost of a small amount of input and layer extensions; the outputs can be validated in real-time to ascertain whether the external invariants are preserved.

The effectiveness of external invariants has been previously demonstrated for matrix operations [93]. The technical investigation in this dissertation explores the applicability of such external invariants to deep neural network computations to provide rigorous error detection at the cost of small overheads.

4.4.2 Hardware Error Detection Through Internal Invariants

The limited applicability of external invariants in the non-linear computational stages encourages us to make use of the plasticity of deep neural networks in the learning process to embed internal relationships across variables. The internal relationships can be formed at various granularities for coarse-grained detection and fine-grained localization of hardware errors.

Coarse-grained invariants are embedded by partitioning the outputs of the deep neural network layers into groups and encouraging the formation of relationships across groups in the training process with additional penalty terms. The introduced penalty terms embed the desired invariants into deep neural networks and allow the detection of hardware errors when a numerical error distorts the established relationships within a layer.

Finer-grained relationships enable more precise localization of hardware errors by imposing local relationships among variables in the computational graphs. An example invariant of this nature could be a numerical order relationship across the neighboring variables. Such fine-grained invariants can be embedded into deep neural network layers by imposing graph constraints in the target deep neural network and ensuring that it attains a competitive accuracy within the confines of these rules. The violation of a fine-grained invariant due to a hardware error would localize the erroneous variables with high precision in deep neural network computations.

4.5 Usage of DNN Plasticity and Redundancy for Relationship Construction

The embedding of internal relationships across variables necessitates re-shaping deep neural networks for objectives that are not necessarily associated with the primary functional goal. *Implementing such supplementary objectives and constructing relationships across variables naturally comes at an additional cost, yet such expenditures can be minimized through the effective use of the computational characteristics of deep neural networks, including their redundant and over-parametrized nature and their plasticity during the training process.*

One can shape the neural networks and build relationships across variables through the proper use of their plasticity in the architectural design and the training process. Proper penalty terms in the loss function can encourage numerical relationships in training, and custom graph constraints can guide deep neural networks to grow and form associations across variables. The software development process requires no additional manual effort after the definition of such relationships, where the optimization algorithm comes up with a parameter set that can operate within the formed relationships and constraints.

The concept of redundancy in deep neural networks is an involved discussion and necessitates further exploration to ensure the continued viability of redundancy utilization for the outlined goals of this dissertation. Deep neural networks are observed to embed redundancy in various computational dimensions, with certain redundancy types proving less amenable to compression. Under-utilized data width, insignificant activations/parameters, and insignificant neurons/filters are just a few well-known examples of such redundancy phenomena.

We observe a novel type of redundancy that stems from the correlations across the outputs in a deep neural network layer. The outputs produced by the neural network layers exhibit dimensionality significantly lower than the layer size; thus, the information content of the layer outputs could be represented with fewer units than the actual layer dimensions. Unlike the conventional approach that evaluates the numeric contribution to rank saliency and redundancy,

the outlined perspective further considers functional correlations for the precise identification of redundancy in deep neural networks.

In summary, the redundancy types that cannot be effectively extracted for computational benefits can be used instead for constructing relationships in deep neural networks. Fully functional deep neural networks can be formed within such constraints without incurring a noticeable loss of expressivity or operational accuracy.

4.6 Harnessing Relationships for Hardware-Friendly Sparsity Embedding

We have discussed the motivation underlying the construction of relationships for the goals of safety and reliability. *The idea of embedding relationships across variables is a powerful strategy and has applications in various other domains, including hardware resource efficiency improvement by encouraging strategic regularity in sparse deep neural network architectures.*

A particular example in this section will concentrate on novel sparsity paradigms that harness the relationships among a group of variables for cultivating structural regularity within the sparsity patterns. The structural regularity within the sparsity patterns yields significant hardware advantages, such as seamless compression of layers at theoretically optimal overheads and precise predictability of the required hardware resources. Meanwhile, the amount of imposed regularity often needs to be kept at reasonable levels to maintain the overall expressivity of the sparsity patterns.

The novel approach in this dissertation relies on the construction of groups among variables in which only a limited number of variables are allowed to be non-zero entries within each group. As a result, the outlined technique reduces the number of parameters by enforcing a regular sparsity structure that provides both flexibility and expressiveness in the selection of non-zero parameter locations.

On the other hand, identifying the optimal configuration for the non-zero locations within

a group is a challenging design space exploration problem. The described challenge can be resolved in an innovative manner by constructing the rules for interaction and encouraging competition among variables within the same group in training. As a result, the training process could identify the most expressive non-zero entry locations without requiring additional steps.

Overall, embedding relationships across deep neural network variables is a powerful practice in designing efficient deep neural network architectures. When properly defined and evolved in the training process, such relationships could furnish hardware-friendly sparse deep neural networks that exhibit high expressiveness and improved hardware regularity.

Chapter 5

Research Vision

5.1 Explored Research Questions

This dissertation explores algorithm-centric and synergistic solutions for resolving the challenges of emerging artificial intelligence hardware platforms. We first identify the unique algorithmic characteristics of deep neural networks that allow this pursuit and describe how these characteristics can be utilized to design reliable and highly efficient hardware processing systems. We list several central questions that repeatedly appear throughout the technical chapters of the dissertation:

- What are the unique characteristics of artificial intelligence algorithms and hardware platforms when compared to general-purpose computing systems, and how could such characteristics be utilized to create effective solutions to hardware challenges?
- How can we effectively reshape and train deep neural networks for additional hardware-focused goals such as safety, reliability, and resource efficiency without impacting their primary functionality?
- How can we enhance the micro-architecture of deep neural network accelerators strategically to contribute to the overall objective of improving safety, reliability, and resource efficiency?

- How does the identification and prioritization of proper objectives, such as the overall functional correctness, influence our approach to safe, reliable, and efficient artificial intelligence hardware system design?
- How can we harness the statistical nature of deep learning algorithms to reduce the associated costs and overheads of the proposed techniques?

5.2 What is Unique for Deep Learning Processing Systems?

5.2.1 Unique Characteristics of Deep Neural Network Algorithms

Inherent Redundancy

Deep neural networks are known to embed redundancy in various dimensions of the computation. Redundancy is observed to be helpful in obtaining more accurate deep neural network architectures in the training process [196]. Numerous model compression techniques in the literature, including quantization [34], parameter pruning [28], and tensor decomposition [162], aim to construct more compact deep neural networks by minimizing model redundancy.

The model compression techniques enable us to improve the resource efficiency of deep neural networks by reducing their inherent redundancy. On the other hand, the complete elimination of deep neural network redundancy is often impractical and may not lead to appreciable computational efficiency benefits. As a result, the remaining amount of redundancy after model compression, even in the case of highly compressed deep neural network architectures, could suffice for reshaping software with custom-tailored goals and embedding useful properties within deep neural networks without necessitating additional overheads or compromising their primary functionality.

Plasticity in the Design Process

The conventional design flow for deep neural networks starts by identifying the deep neural network architecture. The training process, through backpropagation and gradient descent, then determines a set of parameters that work with the selected neural network architecture.

An almost infinite number of choices exist in the architectural definition process (e.g., number of layers, number of units at each layer) that can deliver comparable accuracy in the desired machine learning task. Similarly, the training process could result in a wildly divergent set of parameters, even for the same neural network architecture and the training algorithm, as a result of a slightly randomized computational step within the training process. Such model plasticity allows the reshaping of deep neural networks through various innovative techniques and enables the introduction of valuable properties into deep neural networks without compromising their primary functionality.

Resiliency to Small Numerical Perturbations

Deep neural networks are known to exhibit robustness against small numerical perturbations in the model parameters and activations. To illustrate, deep neural networks can gracefully tolerate the minor errors introduced by the post-training quantization techniques [34], even though error introduction during the quantization process is widespread throughout the model parameters and activations. The resiliency of deep neural networks is a particularly valuable property in the domain of hardware safety and reliability, as it provides deep neural networks the ability to tolerate hardware errors and maintain the same functional accuracy as long as the numerical impact of hardware errors is adequately contained within deep neural network computations.

5.2.2 Unique Characteristics of Deep Neural Network Hardware

Micro-architectural Homogeneity and Regularity

Numerous hardware designs are proposed in the recent literature to improve the performance and efficiency of deep neural network inference [26]. The micro-architecture of deep neural network accelerators is often highly homogeneous and regular due to the parallelism requirements of the underlying deep neural network computations.

The micro-architecture of deep neural network accelerators provides unique opportunities

for innovation while optimizing such designs for resource efficiency and hardware reliability. To illustrate, the regular and fine-grained micro-architecture, when enhanced with strategic hardware modifications, could facilitate the bypass of hardware defects at the cost of a minimal loss in hardware capabilities. Similarly, a great level of dataflow flexibility can be attained at a low cost through minor strategic enhancements made at the level of individual processing elements within the hardware micro-architecture.

Reduced Criticality of Structural Correctness

Structural hardware correctness is a strict requirement for each manufactured gate or memory cell in the conventional VLSI (Very Large-Scale Integration) flow since it is challenging to anticipate the functional outcome of a hardware defect in general-purpose computing platforms. As a result, the structural correctness of each manufactured device is ensured through various VLSI test and fault tolerance methods developed over the past decades.

The inherent resilience characteristics of deep neural networks allow us to relax the structural correctness requirements at the hardware level as long as the overall functional behavior of the algorithm is not impacted. For example, a hardware defect that introduces a minor numerical perturbation within a variable could be well tolerated if it does not significantly affect the classification accuracy of a deep neural network.

The outlined phenomenon opens up avenues substantially different from the conservative perspective followed in the design and manufacturing of general-purpose computing systems by reducing the absolute necessity of structural correctness for each individual variable and arithmetic operation in deep neural network hardware.

5.3 Proposed Approach

This dissertation aims to solve practical problems of deep learning hardware by making use of unique algorithmic and hardware characteristics of deep neural networks. We identify four fundamental avenues to achieve this goal:

1. We reshape algorithms during the architectural definition and the training process to embed useful algorithmic properties into deep neural networks.

2. We demonstrate the feasibility of obtaining significant computational benefits through strategic enhancements in deep neural network accelerators when they are synergistically combined with embedded algorithmic properties of deep neural networks.

3. The inherent algorithmic resilience of deep neural networks to small numerical perturbations encourages us to prioritize functional correctness and align our solutions with the overall functional objectives more closely.

4. The statistical nature of deep learning algorithms relieves us from the obligation of meeting the strict correctness requirements for every produced output as long as the overall accuracy objectives are satisfied, enabling us to innovate techniques that can attain the desired characteristics under this softer constraint at negligible costs when compared to conventional approaches.

5.3.1 Reshaping Deep Neural Networks

The typical deep learning software development flow starts with the definition of a deep neural network architecture. The training process then identifies a set of parameters for this architecture by minimizing a loss function. The former step in this process offers a certain degree of freedom in the architectural definition process, where the algorithm designer can come up with reasonably divergent architectures yet with comparable final accuracy.

The plasticity and redundancy of deep neural network algorithms empower us to reshape software for additional objectives and utilize the training process to obtain architectures that satisfy the custom-tailored hardware safety, hardware reliability, and resource efficiency goals. We make use of a variety of distinct techniques to reshape software throughout the technical chapters of this dissertation:

- We introduce additional objectives in the training process by making use of custom penalty terms in the loss function. We then let the training process shape deep neural networks in the desired direction as it optimizes the model parameters for the additional objectives in addition to the original training goals.

- We train deep neural networks by modeling hardware error effects (e.g., noise, loss of hardware components) accurately in the training process to obtain a set of parameters tailored uniquely to adapt to the target hardware conditions.

- We constrain the forward propagation of deep neural network variables by introducing custom propagation rules in deep neural network graphs. For instance, such rules are observed to be particularly effective for the detection and rectification of the numerical impact of hardware errors in deep neural network computations.

- We shape sparsity patterns with regularity constraints to form an effective contract between hardware and software and obtain highly compact and accurate sparse deep neural network architectures that can be accommodated efficiently on hardware.

These examples underline the opportunities to reshape deep neural networks and contribute to hardware reliability and efficiency objectives through algorithm-centric design techniques.

5.3.2 Strategic Hardware Enhancements

Hardware accelerator and micro-architecture design for deep neural networks have been an active area of research in the past decade [26]. The scope of this dissertation is focused on strategic enhancements in the existing hardware micro-architectures that can deliver extensive computational benefits while keeping the design and verification costs at minimal levels. We thus enhance deep neural network accelerators strategically throughout the technical chapters of this dissertation for various objectives:

- We co-design and embed dedicated hardware modules into deep neural network accelerators to carry out custom-tailored hardware reliability tasks (e.g., filtering neural network variables) in an efficient manner and without leading to any performance bottlenecks.
- We introduce extensive micro-architectural flexibility in the dataflow of deep neural network accelerators by allowing a small degree of additional strategic reconfigurability at the level of individual computational units. Such reconfigurability is coupled with reshaped regularity within the neural network sparsity patterns to boost the efficiency of sparse deep neural network inference.

Since the effectiveness of such hardware enhancements often relies on the degree of co-operation between software and hardware, enhancing the synergy between deep neural networks and deep learning hardware is a primary objective throughout this dissertation.

5.3.3 Functional Correctness Prioritization

The resilience characteristics of deep neural networks engender the possibility of relaxing the rigid structural correctness requirements for the individual variables and computations on the hardware as long as the final functional outputs of deep neural networks are not impacted. Functional correctness prioritization provides us with novel opportunities to achieve hardware safety and reliability goals while reducing the costs needed in conventional designs to ensure structural hardware correctness. Moreover, our technical investigation reveals innovative avenues for promoting the functional correctness of deep neural networks even further to ensure correct algorithmic operation in the presence of structural hardware problems.

Approaching the problem of deep neural network correctness from the functional perspective brings various innovative avenues and opportunities to the table, yet it further creates practical implementation and verification challenges. The verification of the structural hardware correctness is a relatively straightforward task through conventional VLSI test and fault tolerance methods. On the other hand, the assessment and certification of functional correctness is a more

involved pursuit that involves comprehensive knowledge of the executed deep learning algorithm as well as the underlying hardware architecture.

The reader will further note the overlap of functional correctness prioritization with the concept of *approximate computing* investigated in various application domains [197] since both techniques focus on the final functional outcome as the inaccuracies in the intermediate computational stages are tolerated by the intrinsic resilient nature of the executed algorithms.

5.3.4 Harnessing the Statistical Nature of Deep Neural Networks

The statistical nature of deep neural networks could provide an additional degree of freedom when combined with the outlined strategy of functional correctness prioritization. A certain level of inaccuracy is expected during the regular functional operation of deep neural networks. Such statistical characteristics allow us to forgo the correctness requirements on an exceptional basis or transform the computations in a way that is not entirely functionally equivalent to the baseline models yet with comparable functional accuracy.

The utilization of such statistical characteristics establishes a rich design space for exploration. A minor loss in functional accuracy can be converted into significant gains in terms of hardware performance or efficiency. Moreover, innovative hardware error resilience techniques can maintain a deep neural network accuracy almost as good as the baseline, yet with negligible costs and overheads compared to conventional hardware fault tolerance methods.

5.4 Research Challenges

5.4.1 How to Explore the Design Space?

The development flow of Software 2.0 [12] driven by deep neural networks is remarkably different from the conventional software design. The exploration of functional deep neural networks is often an intractable problem due to the extent of the design exploration space. The Software 2.0 development flow tackles this challenge by defining reasonable deep neural archi-

tectures and then evolving the trainable parameters of the architecture through backpropagation and gradient descent to obtain accurate deep neural networks.

The task of constructing trainable deep neural network architectures could be considered as the problem of constraining the architectures through the architectural hyperparameters so as to deliver expressivity while keeping the computational complexity at reasonable levels. The novel perspectives brought to the table by the dissertation could be considered a process of constraining architectures further for the diverse goals of safety, reliability, and efficiency.

Obtaining fully functional deep neural networks with additional constraints requires significant modifications in the design and training process. While the external enforcement of such constraints is often impractical, imposing the desired objectives during the training process is a feasible avenue to approach the outlined design space exploration problem.

We define the desired constraints and relationships in the architectural definition phase and ensure that the imposed constraints and relationships can operate with the standard training methodologies. As a result, we can use the conventional training algorithms, namely backpropagation and gradient descent, to perform design space exploration and come up with functional deep neural network models that meet the desired design objectives.

The outlined methodology is relatively straightforward, but the imposed additional constraints could lead to several training challenges. Several examples of such challenges and the corresponding solutions are discussed in the next section.

5.4.2 How to Resolve Training Challenges?

An issue that requires attention is that training deep neural networks with constraints can give rise to challenges such as discontinuities and imperfectly aligned objectives, thus impacting the efficacy of the standard training algorithms.

Constructing Balanced Training Objectives

Deep neural networks can be shaped for the auxiliary objectives through modifications in the loss function. To illustrate, the distance between two variables can be minimized by including their absolute difference in the loss function as an additional penalty term and running the neural network optimizer to minimize both the standard loss and the additional penalty terms. The reader will note that the usage of such penalty terms is analogous to the common practice of regularization in the machine learning domain.

On the other hand, the existence of multiple training objectives could lead to training challenges if the significance of such goals is not judiciously balanced. For instance, the penalty terms with disproportionate significance can hinder the original training objectives.

A conventional yet effective solution to this problem from the common practice of regularization involves the usage of scaling coefficients for the additional penalty terms, where such coefficients can be tuned in the hyper-parameter definition process to assign proper weights to the additional training objectives.

Resolving Relationship Conflicts

This dissertation advocates for a novel perspective of constructing relationships across deep neural network variables for the desired safety, reliability, and efficiency objectives. Such relationships could then be utilized for identifying erroneous variables or embedding regularity within the sparsity patterns for hardware efficiency.

Despite the potential benefits, the relationships across deep neural network variables come with additional challenges. The relationships could impose conflicting constraints on the subjected variables, necessitate competition for the same limited resources, and as a consequence, lead to instability in the training process. For example, in a scenario where a variable controls the propagation of its neighbor, e.g., in the form of a median filter, the variable is subject to constraints regarding its own numeric contribution in the neural network graph as well as its neighbor's. If the necessary actions, e.g., gradient updates, for both objectives are not aligned,

the training process may not be able to converge into a solution that satisfies both objectives.

It is feasible to minimize the number of such conflicts if the rules of the relationships, including the nature of the interactions and the number of interacting variables, are appropriately identified in the architectural definition process. The algorithmic redundancy and plasticity of deep neural networks could then be harnessed in the training process to identify a set of parameters that can operate functionally within the scope of such relationships and constraints.

Ensuring Efficient Gradient Propagation

The optimization process requires end-to-end differentiability in the deep neural network graphs for effective gradient backpropagation. The requirement for full differentiability is not always feasible, such as in the case of quantization-aware training, where simple straight-through estimators such as identity functions are used to remedy differentiability problems [34].

On the other hand, the introduced constraints in this dissertation lead to a greater degree of discontinuities that involve relationships across multiple variables. As a result, the usage of simple straight-through estimators such as identity function is not expected to suffice to address such relationships. The relationships across variables could be modeled through discrete mathematical functions such as the Heaviside step function [198] that does not exhibit smooth gradient propagation characteristics. The novel training approach to these problems involves approximating such discrete functions with functionally similar, yet smoother and well-differentiable counterparts such as Sigmoid in the backward training pass to facilitate effective gradient communication.

Furthermore, embedding relationships across variables during training may result in fierce competition across variables. A certain amount of competition is desirable and should be encouraged to avoid locally optimal solutions, yet excessive competition often leads to instability in training and precludes model convergence. The aforementioned challenge can be controlled through the intelligent scaling and distribution of the gradients in the backward training phase to ignite an appropriate level of competition while ensuring long-term training stability.

5.5 Technical Progress

The technical discussion of the dissertation considers the outlined characteristics of deep neural network algorithms and hardware architectures, and it adheres to the proposed approach to tackle the challenging problems of artificial intelligence hardware. We believe that these principles have vast application potential in various hardware-related domains, including but not limited to functional safety, hardware reliability, hardware testing, performance/efficiency, and hardware security.

The technical discussion of this dissertation will primarily focus on the problem of functional safety and hardware reliability, as well as on improving deep neural network inference performance and efficiency. We present the following studies as a part of our technical discussion:

- We present various algorithm-centric methods for embedding invariants into deep neural network computations and demonstrate how these invariants can be utilized for detecting hardware errors in deep neural network accelerators at a low computational cost and without necessitating explicit information redundancy.

- We furthermore build upon this approach to enable fine-grained localization of hardware errors and complement it with innovative error rectification methodologies to maintain deep neural network accuracy. By snapping outsized error effects back to within the realm of minor numerical inaccuracies, the proposed techniques block the propagation of numerically significant error effects and boost the hardware error resilience of deep neural network accelerators.

- The learned lessons regarding deep neural network characteristics open up possibilities for novel algorithmic methods that can enhance the hardware error resilience of deep neural networks by design. Through proper numeric range construction strategies, we enable deep neural networks to operate accurately even under extreme hardware error rates and at the cost of no additional overheads.

- We boost the manufacturing yields and extend the operational lifetime of deep neural network accelerators noticeably through customized and cost-effective adaptivity against permanent hardware defects, which is attained through the synergistic design of adaptable hardware platforms and decentralized deep neural network algorithms.

- We explore diverse types of redundancy in deep neural networks, such as the functional correlation relationships between neuron and convolution filter outputs. We propose novel layer reduction and reconstruction methods to eliminate such redundancy occurrences and improve the inference performance and efficiency of deep neural networks as a result.

- We enhance the synergy between sparse deep neural networks and hardware platforms through the imposition of pre-defined regularity constraints in sparsity patterns, resolve critical micro-architectural challenges posed by unstructured sparsity, and obtain practical performance and efficiency improvements in the inference of sparse deep neural networks through minor enhancements in the hardware micro-architectures that are conventionally tailored for dense arithmetic operations.

We demonstrate significant technical progress in the outlined areas in the scope of this dissertation. Many other hardware domains, such as hardware testing and security, remain promising candidates for applying the advocated principles of this dissertation in future research.

Chapter 6

Hardware Error Detection in DNN Accelerators via External Invariants

The widespread adoption of DNNs (deep neural networks) in safety-critical systems necessitates the examination of the safety issues raised by hardware errors. The consequent interest in fault tolerance methods that are comprehensive yet low-cost to match the margin requirements of consumer deep learning applications can be met through a rigorous exploration of the mathematical properties of deep neural network computations. Our novel technique, *Sanity-Check*, allows error detection in fully-connected and convolutional layers through the use of external algorithmic invariants. The purely software-based implementation of *Sanity-Check* facilitates the widespread adoption of our technique on a variety of off-the-shelf execution platforms while requiring no hardware modification. We further propose a dedicated hardware unit that seamlessly integrates with modern deep learning accelerators and eliminates the performance overhead of the software-based implementation at the cost of a negligible area and power budget in a deep neural network accelerator. The external invariants of *Sanity-Check* deliver perfect critical error coverage in our error injection experiments and offer a promising alternative for low-cost error detection in safety-critical deep neural network applications.

6.1 Introduction

Multiply-and-accumulate operations constitute the backbone of modern deep neural network computations. Multiply-accumulate operations exhibit a linear nature as they maintain the linear relationships of their inputs at the system outputs. This chapter utilizes such linear relationships in the form of external invariants to detect anomalous behavior and hardware error effects. We demonstrate that such external invariants can be embedded into deep neural network layers through a small amount of extensions in the input and weight tensors; meanwhile, the outputs can be checked to determine if the external invariants are maintained at the outputs. The deviation from an expected external invariant is then utilized to identify critical hardware errors. Similar invariants have been previously proposed for various matrix operations [93]. This chapter focuses on the usage of such external invariants in the domain of deep neural networks and deep learning accelerators.

The overall research contributions of this chapter can be summed up as follows:

- We mathematically demonstrate how the linearity property of fully connected and convolutional layers could be utilized to embed external invariants in deep neural network accelerators and deliver proven error detection properties at the cost of a single neuron/filter, and a few additional multiply and accumulate operations at each layer.

- We demonstrably show that our detection rates match those of state-of-the-art error detection methods, yet incur a fraction of their overhead since the cost of additional computations is amortized across the entire layer.

- We set clear guidelines for the purely-software *Sanity-Check* implementation and demonstrate its applicability on off-the-shelf hardware platforms.

- Finally, we characterize a dedicated hardware design that can perform the required operations efficiently in a systolic array DNN accelerator.

We mathematically define the proposed external invariants in Section 6.2 for both fully-connected and convolutional layers. Section 6.2 discusses further the error detection properties of the external invariants and the impact of numerical inaccuracies on the checksum computations. Section 6.3 covers both the software implementation and the suggested hardware architecture. In Section 6.4, we experimentally demonstrate that our technique attains a perfect error-caused misprediction (i.e., critical error) coverage and analyze the area and power overheads of the hardware implementation.

6.2 DNN Hardware Error Detection via Linear Checksums

While error detection through duplication is prohibitively expensive for DNN processing, an alternative technique constructed through the fundamental invariants of the underlying computations can deliver even stronger error detection rates while expending much smaller overheads. The majority of DNN computations are linear, and linearity is a useful property to employ in an error detection scheme. *Sanity-Check*, a checksum-based error detection method for deep neural networks, makes use of the linearity property of convolutional and fully-connected layers and introduces low-overhead checksums in both the spatial and the temporal computation dimensions to check the consistency of the performed operations. The spatial checksums introduced through an extra neuron in the fully-connected and an extra filter in the convolutional layers guarantee that the layer outputs produced for a single prediction always hold a linear invariant. The temporal checksums necessitate an extra input processing step after a certain number of inputs have been processed, guaranteeing the consecutive outputs of the same neuron or filter are always linearly related. We introduce the spatial and temporal checksums in this section for both fully-connected and convolutional layers, and proceed to explain how these checksums engender superior error detection in DNN computations. The preponderance of the computational time in CNNs (convolutional neural networks) devoted to multiply-accumulate operations in the fully connected and convolutional layers, frequently surpassing even 90% [199], puts them center

stage in the quest to attain significant error coverage in safety-critical DNN applications.

6.2.1 Checksums in Fully-Connected Layers

The fully-connected layer operation demonstrated in Figure 2.1 can be mathematically formulated through Equation (6.1):

$$O[z][u] = B[u] + \sum_{k=0}^{C-1} (I[z][k] * W[k][u]), \qquad 0 \leq z \leq N-1, \;\; 0 \leq u \leq M-1 \tag{6.1}$$

In Equation (6.1), we use matrices I, O, W, and vector B to represent input activations (a batch), layer output pre-activations (a batch), layer weights and layer bias values, respectively. N denotes the number of inputs in the batch, and M the number of neurons. Finally, C corresponds to the number of input connections for each neuron.

Our goal is to introduce unvarying properties, *invariants*, at the fully connected layer outputs so that they could be utilized for error checking. As a first step, let us try to embed the following error-checking invariant to the output matrix so that the summation of the output over the spatial axis (u') always yields zero:

$$\sum_{u'=0}^{M-1} O[z][u'] = 0, \qquad \forall z \tag{6.2}$$

The described invariant indicates that each row will accumulate to zero in the output matrix O. As each output column is generated by a single neuron, it could be alternatively visualized as neuron outputs in Figure 6.1 summing to zero for a single prediction. If the described invariant can be embedded into DNN layers to hold regardless of the provided input activations, it could be utilized for error detection since an erroneous neuron output will result in the violation of this invariant. To investigate whether the introduction of such a property is feasible, let us sum the right-hand side of Equation (6.1) in a similar manner. By distributing the accumulation operation over the addition, we can obtain the following intermediate expression:

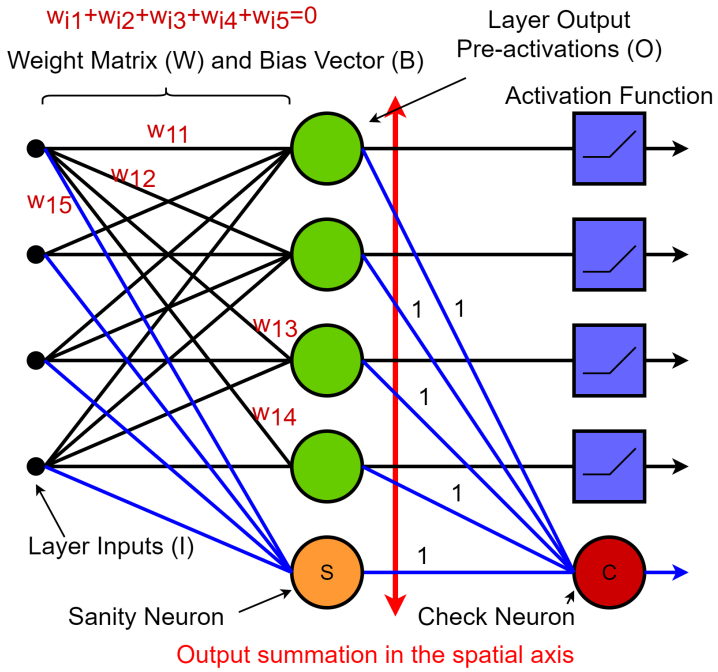

Figure 6.1. Spatial checksum in the fully-connected layers.

$$\sum_{u'=0}^{M-1} O[z][u'] = \sum_{u'=0}^{M-1} B[u'] + \sum_{u'=0}^{M-1} \sum_{k=0}^{C-1} (I[z][k] * W[k][u']) \tag{6.3}$$

Let us further change the order of the two summation symbols in the second term of the right-hand side in Equation (6.3) and extract the $I[z][k]$ term outside of the inner summation since it has no term with u', and thus can be treated as constant in the inner summation:

$$\sum_{u'=0}^{M-1} O[z][u'] = \sum_{u'=0}^{M-1} \boldsymbol{B[u']} + \sum_{k=0}^{C-1} (I[z][k] \sum_{u'=0}^{M-1} \boldsymbol{W[k][u']}) \tag{6.4}$$

It could be clearly seen that if the highlighted weight and bias summations hold the value of zero in this formulation, the right-hand side reduces to zero and causes the invariant introduced in Equation (6.2) to be always satisfied. While the mathematical validity of this argument can thus be ascertained, questions may linger as to why the described solution is particularly useful for embedding an error-checking property. The reader will note that its usefulness stems from its reliance solely on the assumptions on the weight and bias values, thus ensuring that the error-

checking invariant holds at the output regardless of the particular input combination introduced into the layer. Each row in the output matrix is guaranteed to accumulate to zero as indicated in Equation (6.2) as long as the weight and bias values add up to zero when accumulated over u':

$$\sum_{u'=0}^{M-1} B[u'] = 0 \quad \wedge \quad \sum_{u'=0}^{M-1} W[k][u'] = 0, \quad \forall k \tag{6.5}$$

Although the conditions in Equation (6.5) will not hold in a trained DNN layer directly, they can be easily introduced by adding one more column to the weight matrix and setting the additional column values to the additive inverse of the summation of all the values in the same row. Similarly, one additional bias value is required to be set equal to the additive inverse of the sum of all other bias values. Interestingly, these modifications are equivalent to adding one more neuron in the fully-connected layer, which we will refer to as the *sanity neuron*. The *sanity neuron* in Figure 6.1 guarantees that the layer outputs always sum to zero regardless of the layer input activation values. An additional neuron in the next layer, which is termed as *check neuron*, can perform the described check operation by summing the neuron outputs at each prediction. If the invariant is embedded through the outlined modifications, the *check neuron* will always accumulate to zero in the absence of errors. We refer to this checksum as the *weight checksum* as it is introduced through the weight and bias modifications, and categorize it as a *spatial checksum* since the layer outputs are checked through accumulations in the spatial output dimension as indicated by the red axis line in Figure 6.1.

We can perform the analogous steps in the temporal axis (z'), which is equivalent to accumulating each column individually in the output matrix:

$$\sum_{z'=0}^{N-1} O[z'][u] \tag{6.6}$$

We sum the right side of Equation (6.1) analogously, and after similar steps, we obtain:

$$\sum_{z'=0}^{N-1} O[z'][u] = \sum_{z'=0}^{N-1} B[u] + \sum_{k=0}^{C-1} (W[k][u] \sum_{z'=0}^{N-1} \boldsymbol{I[z'][k]}) \tag{6.7}$$

In Equation (6.7), if the highlighted input summation is zero over the temporal dimension ($\sum_{z'=0}^{N-1} I[z'][k] = 0, \quad \forall k$), the left-hand side reduces to:

$$\sum_{z'=0}^{N-1} O[z'][u] = \sum_{z'=0}^{N-1} B[u] = N \cdot B[u], \qquad \forall u \tag{6.8}$$

The output matrix summation over the temporal axis while not necessarily zero is nonetheless constant and equals to the product of the bias with the temporal batch size N. As a result, it can be used as an invariant for error detection. The described solution is unique, similar to the introduced property in the weight and bias values, because it only relies on the assumptions on inputs and holds regardless of the weight and bias values. Although input vectors are not expected to yield zero when point-wise accumulated over time for a set of predictions, the required condition can be implemented by keeping the accumulation of the input vectors in a *temporal batch* on the fly and providing the additive inverse of the accumulation as an additional input vector at the end of the batch. Similarly, the check operation can be actualized by accumulating the layer outputs over time. We denote this checksum as an *input activation checksum* and categorize it as a *temporal checksum* since the summation is over the temporal dimension of the input and output matrices as indicated by red axis lines in Figure 6.2.

6.2.2 Checksums in Convolutional Layers

The linear checksums can be constructed in the convolutional layers as well. As the convolutional layers are traversed in four dimensions instead of the two of fully connected, one can entertain the possibility of introducing four analogous checksums. Let us formulate the convolution layer operation demonstrated in Figure 2.2 with the notation introduced in [61]:

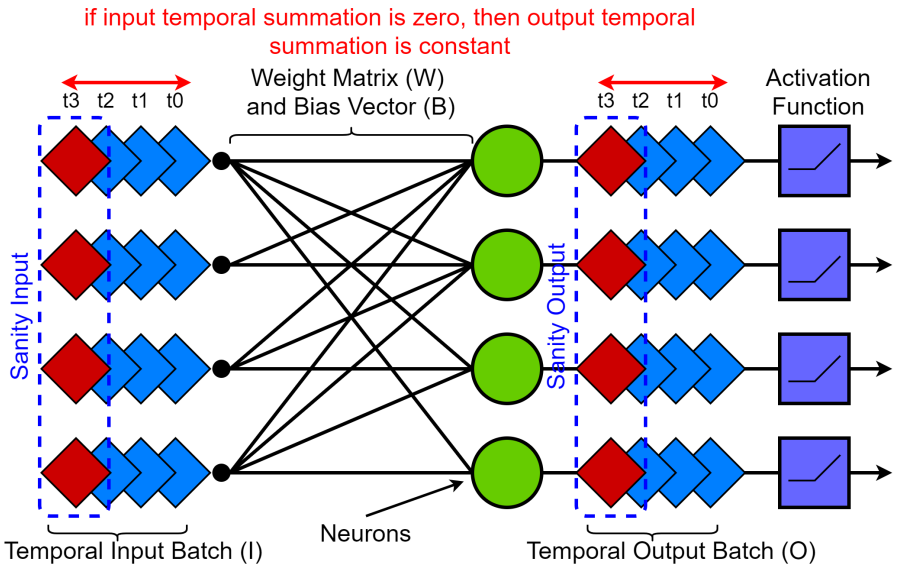

Figure 6.2. Temporal checksum in the fully-connected layers.

$$O[z][u][x][y] = B[u] + \sum_{k=0}^{C-1} \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} (I[z][k][Ux+i][Uy+j] * W[u][k][i][j])$$

$$0 \leq z \leq N-1, \;\; 0 \leq u \leq M-1 \quad (6.9)$$

In Equation (6.9), I, W, and O refer to the 4-dimensional input, filter weight, and output tensors respectively, and B indicates the bias vector. N denotes the batch size, M the number of filters, and C the number of filter channels, each of dimensions S and R. Unlike the fully connected layers, each element-wise multiplication is replaced by a 2-D convolution over the x and y dimensions with U denoting the stride size.

Let us start by investigating the channel (u) and temporal dimension (z) of the output tensor, analogous to the case for the fully connected layers. We want to introduce an invariant in the output tensor O, where the summation over the channel dimension (u) always yields zero so that it could be utilized for error checking. We can start by accumulating both sides of Equation (6.9) over the spatial axis (u) to investigate the conditions which yield such an error-checking property at the output. After algebraic simplifications, we obtain:

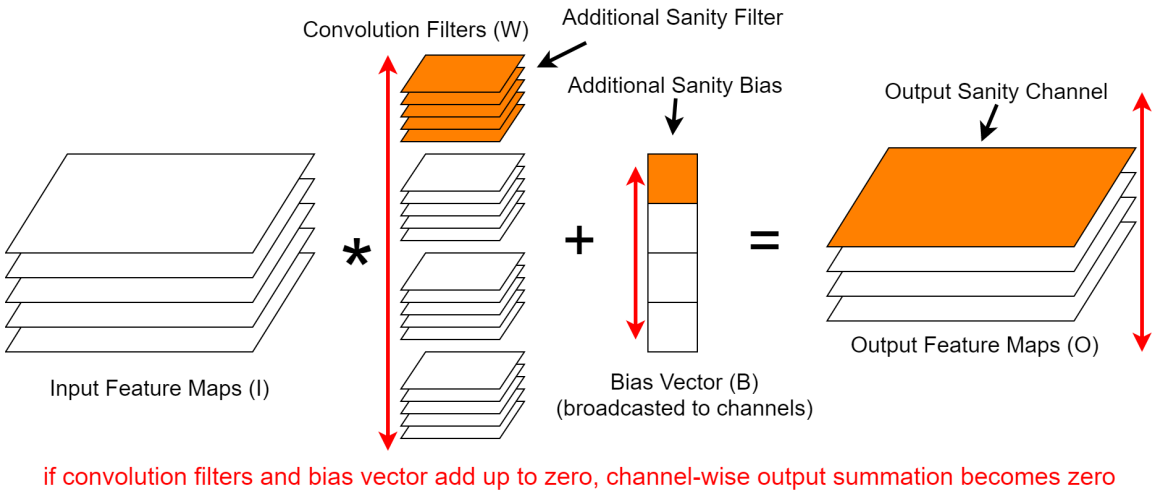

Figure 6.3. Spatial checksum in the convolutional layers.

$$\sum_{u'=0}^{M-1} O[z][u'][x][y] = \sum_{u'=0}^{M-1} \boldsymbol{B[u']} + \sum_{k=0}^{C-1}\sum_{i=0}^{S-1}\sum_{j=0}^{R-1}(I[z][k][Ux+i][Uy+j]\sum_{u'=0}^{M-1}\boldsymbol{W[u'][k][i][j]}) \quad (6.10)$$

The left-hand side will be zero in Equation (6.10) ($\sum_{u'=0}^{M-1} O[z][u'][x][y] = 0$) for all x, y, z if the highlighted weight and bias summations are zero:

$$\sum_{u'=0}^{M-1} B[u'] = 0 \quad \wedge \quad \sum_{u'=0}^{M-1} W[u'][k][i][j] = 0 \quad \forall\, i,j,k \quad (6.11)$$

The solution in Equation (6.11) indicates that if the filters and biases accumulate to zero over u, then the summation of the output tensor over u will yield zero regardless of the inputs, enabling its use as an error-checking invariant. We need to introduce an extra filter and bias value, as shown in Figure 6.3, and the additional filter/bias values should be set accordingly to satisfy Equation (6.11). We refer to this checksum as a *filter checksum* and categorize it as a *spatial checksum* since the output summation is performed in the spatial dimension as demonstrated in Figure 6.3.

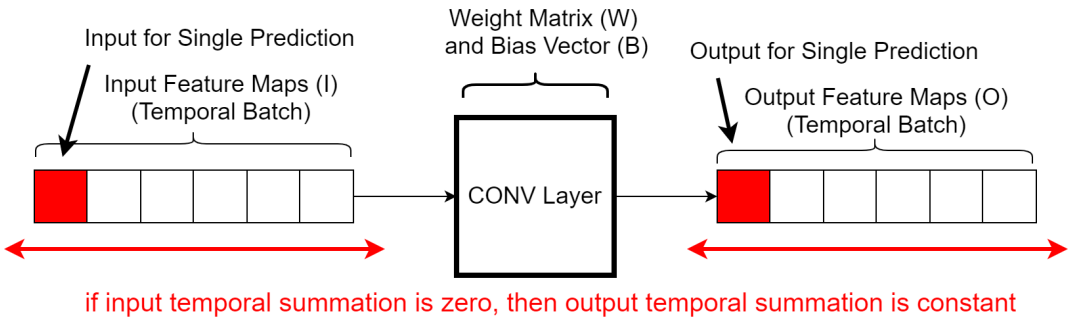

Figure 6.4. Temporal checksum in the convolutional layers.

The derivation of the temporal checksum over the dimension of z requires similar steps. If the input tensor yields zero when accumulated over z as in Equation (6.12), we eventually obtain a constant summation at the output when accumulated over the same dimension, as shown in Equation (6.13). As in the case of the fully connected layers, the checksum can be implemented through the point-wise accumulation of the inputs over the temporal batch and feeding the additive inverse of the accumulated sum as an additional input. The outputs are accumulated similarly and compared with the expected constants to enable error detection as in Figure 6.4. We refer to this invariant as *input feature map checksum* and categorize it as a *temporal checksum*.

$$\sum_{z'=0}^{N-1} I[z'][k][Ux+i][Uy+j] = 0 \qquad \forall\, k,x,y,i,j \tag{6.12}$$

$$\sum_{z'=0}^{N-1} O[z'][u][x][y] = \sum_{z'=0}^{N-1} B[u] = N \cdot B[u] \quad \forall\, u,x,y \tag{6.13}$$

The checksums in the remaining dimensions involve certain technical complications. A constant checksum in the $x(y)$ dimension of the output may require input feature maps to satisfy up to $S(R)$ different zero checksums. In particular, the input feature map columns(rows) that are multiplied with the same filter channel column(row) should sum to the zero vector. Implementing the last two checksums might require up to $S(R)$ more padding columns(rows) in each feature map and tracking $S(R)$ different summations. We limit our study to the presented

Table 6.1. Error detection guarantees of the Sanity-Check checksums.

Error/Checksum	Spatial	Temporal	Spatial+Temporal
Input	No Detection (*L.1*)	1 Error (*L.2*)	1 Error
Weight	1 Error (*L.2*)	No Detection (*L.1*)	1 Error
Bias	1 Error (*L.3*)	All Errors (*L.4*)	All Errors
Output/Comp.	1 Error (*L.5*)	1 Error (*L.5*)	3 Errors (*L.6*)

first two checksums to avoid such overheads. The experimental results confirm that the first two checksums suffice in attaining high error coverage.

6.2.3 Error Detection Guarantees of the Checksums

This section focuses on the delineation of the error detection properties of the presented checksums. Table 6.1 maps these properties by the dimensions of the delivering checksums and by error type; six lemmas are used to establish these error detection properties. The proofs of the lemmas for fully-connected layers carry *mutatis mutandis* for the convolutional layers.

Lemma 1: Spatial checksums can not detect input errors. Similarly, temporal checksums fail to detect weight errors. *Proof:* The spatial checksum invariant is satisfied as independent of the layer inputs. Similarly, the temporal checksum is fulfilled regardless of the weight values.

Lemma 2: A single input error is guaranteed to be detected by the temporal checksums. Similarly, a single error in the weight/filter coefficients is guaranteed to be detected by the spatial checksums. *Proof:* A single error in the input value $I[z = z'][k]$ can only impact the output entries $O[z = z'][u]$ due to the formulation of a fully-connected layer. As only a single error manifests at each output column, the column-wise temporal checksum guarantees the detection of the error deviation. The proof for the spatial checksum can be constructed similarly.

Lemma 3: A single bias error is guaranteed to be detected by the spatial checksums. *Proof:* A single bias error $B[u = u']$ manifests itself in the output entries $O[z][u = u']$. As only a single error manifests at each output row, the deviation can always be detected by a row summation.

Lemma 4: Temporal checksums detect all possible bias errors. *Proof:* A single error in the bias value $B[u = u']$ impacts the entire output column $O[z][u = u']$ by adding the same error amount to each element in the corresponding output column. The temporal checksum multiplies the error value by N as it accumulates the output tensor along the columns. Thus, any possible bias error is guaranteed to be detected.

Lemma 5: Spatial checksums are guaranteed to detect a single output value error or a single computation error. Temporal checksums deliver the same error detection guarantees. *Proof:* An error with location $O[z = z'][u = u']$ deviates the checksum results in one row-wise and one column-wise summation, thus guaranteeing detection in either dimension.

Lemma 6: Three or fewer output/computation errors are guaranteed to be detected by the combination of the spatial and temporal checksums. *Proof:* An error at $O[z = z'_1][u = u'_1]$ can be masked by two errors $O[z = z'_1][u = u'_2]$ and $O[z = z'_2][u = u'_1]$ such that $z_1 \neq z_2 \quad \wedge \quad u_1 \neq u_2$. A fourth error $O[z = z'_2][u = u'_2]$ is required to mask the remaining dimensions of the second and third error. It should be noted that the error magnitudes should align accordingly for this scenario to happen. The combination of spatial and temporal checksums can also correct a single output error as it deviates in only a single row and column summation. However, certain three error checksum signatures may match those of single error cases; consequently, three error cases may end up being miscorrected. We prioritize triple error detection over single error correction, resulting in *Sanity-Check* forgoing error correction.

6.2.4 Impact of Numerical Inaccuracies on Checksum Calculations

Our previous discussion has delivered theoretical proofs of the outlined error detection properties. Nevertheless, real-world considerations necessitate an examination of the impact of finite-length computation on this certainty. Floating-point or scaled fixed-point operations may need to fit their results in fixed-length representations, necessitating the invocation of techniques such as truncation and rounding. The consequent minute error accumulation deviates the sums from the expected values to be checked against. This issue has been a concern for ABFT

(Algorithm-Based Fault Tolerance) [93] and the concurrent test methods for digital linear systems [200]. Different number coding techniques [201] have been proposed in the prior literature to tackle the inaccuracy problem, but the integration of such schemes into existing systems involves significant challenges. This section investigates the numerical inaccuracy problem for mainstream DNN hardware implementations with floating-point (i.e., GPU (graphics processing unit)) and fixed-point (i.e., embedded DNN accelerator) data types and sets guidelines to deploy *Sanity-Check* in these systems without experiencing significant numerical issue problems.

We identify two fundamental numerical inaccuracy sources that can lead to checksum deviation even in the absence of hardware errors. First, layer outputs are generated through a series of multiply-accumulate operations, and the checksum is derived through the summation of these output values. If each multiplication and addition operation introduces round-off errors, the final checksum value will be impacted by these deviations. For instance, assuming that the round-off error introduced at an operation is bounded by ε_r, the maximum accumulated error is proportional to $CM|\varepsilon_r|$ for the spatial checksum calculation in the fully-connected layers, where M and C denote the number of neurons and their input connections, respectively. While the described bound is overly pessimistic, the expected checksum deviation range can be approximated for a rounding scheme through a Gaussian distribution [200] centered at zero ($\mu = 0$), and standard deviation proportional to the round-off error of each operation, and the square root of the operations performed ($\sigma \propto \sqrt{CM}|\varepsilon_r|$). Second, a more covert inaccuracy source may stem from the invariant embedding process. For instance, each additional sanity neuron weight in the fully connected layers might be off up to $M|\varepsilon_r|$ due to round-off errors since they are generated through the accumulation of M other weight values. This inaccuracy will be amplified while generating the sanity neuron output, and it may consequently lead to an additional deviation up to $M|\varepsilon_r| \times L_1(\vec{I})$ in the generated checksum value where the last multiplicative term is the L1-norm of the input activation vector.

In floating-point data representations (i.e., IEEE 754 single-precision), we have observed the impact of numerical inaccuracies, yet the final deviation on the checksum result is negligible

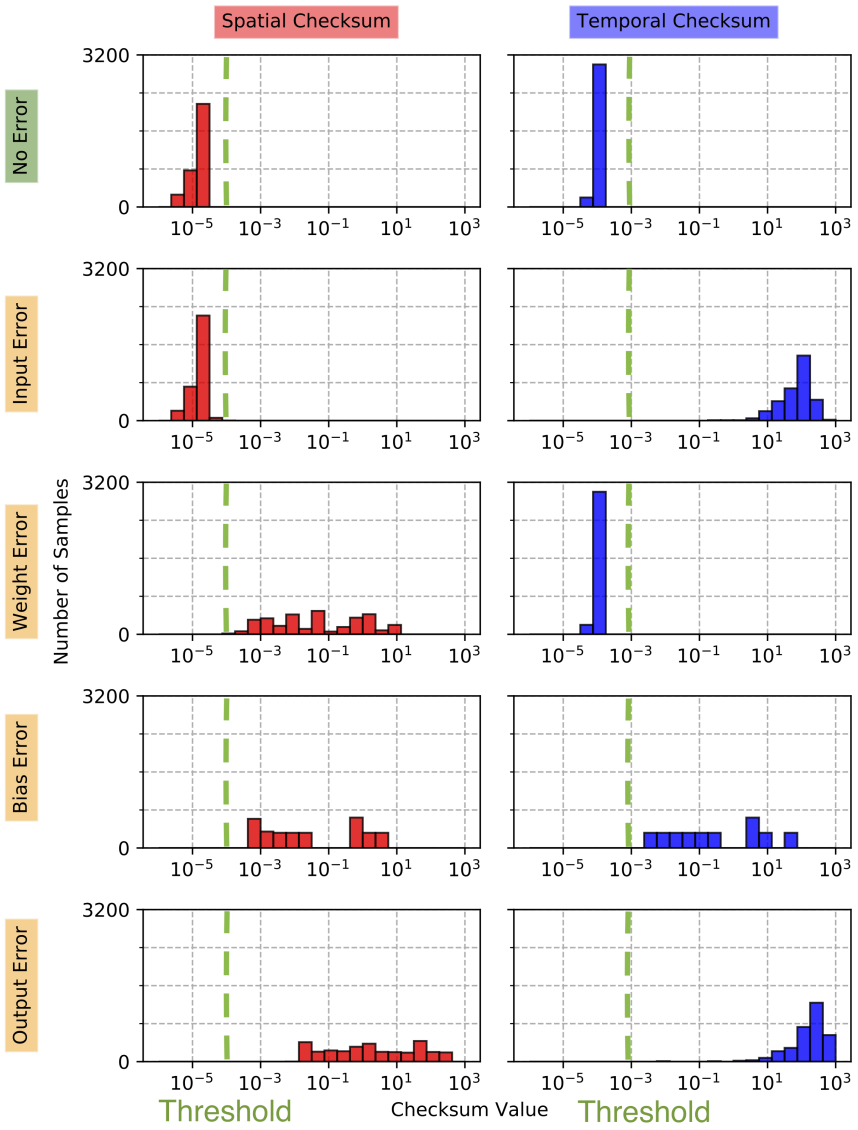

Figure 6.5. The distribution of the measured checksum values.

for both spatial and temporal checksums in both large fully-connected and convolutional layers. For instance, Figure 6.5 illustrates the difference in the distribution of the temporal and spatial checksum values for the no error case (only numerical inaccuracy effects) and in the presence of single bit-errors. The most important observation from Figure 6.5 is that there is at least one checksum for each error type, which can clearly distinguish error-caused deviations from numerical inaccuracies with a pre-defined threshold. As the numerical imprecision effect is

sharply distinguishable from the error cases with high accuracy, we can achieve an outstanding error coverage without leading to false alarms. We can check whether the generated checksum exceeds a threshold rather than aiming to ascertain a strict match to a zero value, thus preventing false alarms in our execution environment with the floating-point backend. To achieve this, the target DNN model needs to be profiled after the checksum integration to measure the largest deviation that can occur due to numerical inaccuracies, with the spatial and temporal checksum thresholds subsequently being set for each layer accordingly.

The impact of round-off errors is less of a problem for fixed-point implementations with full bit-width multiplication and accumulation. For instance, if the multiplication operation in a DNN accelerator doubles the bit-width of its operands in the result, and accumulations are performed with sufficient bit-width and no truncation, then the partial sums can be accumulated without any round-off errors. Modern DNN accelerators perform MAC (multiply-accumulate) operations in the described manner, and accumulated sums are truncated/scaled at the end of the layer; thus, round-off errors can be avoided by utilizing full resolution accumulation results in *Sanity-Check* calculations. The inaccuracy stemming from the sanity weights and inputs can be avoided by deriving these quantities through the accumulation of the quantized versions of the weights and activations. If the data representation range used for weights or activations is not sufficient to represent the additional variables, *Sanity-Check* operations can be performed in residue arithmetic [93] as discussed in Section 6.3.3, with such design decisions ensuring no introduction of any round-off errors.

Even when the numerical inaccuracy effects exist in the system and they suppress the impact of small bit error deviations, it rarely becomes a practical issue for *Sanity-Check* deployment. It is observed that such small bit error effects are almost always benign for deep neural networks; thus, the checksum thresholds can be set to relatively large values to avoid false alarms while also tolerating the benign bit errors which do not lead to DNN misprediction. The experimental validation of this claim is provided in Section 6.4.

6.3 Experimental Method

We implemented *Sanity-Check* on AlexNet [49] to measure the coverage rates and the performance impact on various hardware platforms. AlexNet has a reasonable size for practical applications, and it allows us to demonstrate the *Sanity-Check* concept easily as it consists of plain convolutional and fully-connected layers without residual connections. We used Keras [202] with the Tensorflow [203] backend for the development, with all experiments conducted on an NVIDIA GTX1060 GPU. Additional performance measurements are taken on an Intel i5-8600K CPU (central processing unit). We trained the model on the German Traffic Sign Recognition Benchmark (GTSRB) [50] and achieved 96.09% top-1 test set accuracy. Finally, we have designed the *Sanity-Check* hardware and integrated it into an open-source DNN accelerator [37]. We have characterized both area and power overheads through synthesis experiments.

6.3.1 Error Injection Method

We consider bit errors to model the impact of transient hardware-level errors in our experiments by following [66, 68, 69]. We inject activation errors into layer inputs after checksum encoding and into layer outputs before the outputs have been checked. In addition, we inject errors into the weight and bias parameters of each layer. Any transient error during the computation (during multiply-accumulate) only impacts one output entry, enabling us to model a single computation error as a single output error. This property can be easily seen by picking a weight and activation pair for multiplication in either the fully-connected or convolutional layers and observing that the multiplication result can only source a single output.

ABFT evaluations are commonly conducted through fault models either at the module level [93] or at the arithmetic operation level. However, the random bit-flip fault model is more commonly employed in functional safety research because of its tight correlation with the physical SEU (single event upset) effects, thus urging us to use it to model hardware faults in mainstream architectures. We assume 16-bit fixed-point data types for both weights and

Algorithm 6.1: Details of the bit error injection procedure

Input : Input Tensor, Error Rate
Output : Output Tensor

```
Calculate injected error count using error rate and input tensor size
Randomly determine fault locations
for each error injected data location do
    Get the numerical value of the data (n)
    Determine the random bit location (i)
    if i is the sign bit position then
        Invert the numerical sign of the data
    else
        Perform a pseudo-quantization on n to find the bit value at i^th position
        if i^th bit value is zero then
            Add (quantization scale factor) × 2^i to the data value
        else
            Subtract (quantization scale factor) × 2^i from the data value
        end
    end
end
```

activations while modeling the bit flips on the fixed-point format. After profiling the weight and activation ranges on the AlexNet model, we allocate the integer and fraction bits to accommodate the possible data range. Algorithm 6.1 outlines the error injection procedure.

Weight and Bias Errors

Weight and bias errors are injected through direct modifications on these values. We first read the weights/biases of the target model, then apply the procedure outlined in Algorithm 6.1. Finally, we load the modified weights/biases into the model to perform predictions. The faults on the weights/biases have a long-lasting impact as they influence all predictions until the weights/biases are refreshed by the correct ones. This is a typical scenario for a DNN accelerator if the weight/bias values are stored for reuse in an SRAM (static random-access memory) buffer with no special safety features. The absence of such features may result in the persistence of the weight errors throughout the entire device operation.

Input and Output Errors

Layer input and output errors are injected dynamically. We create dedicated DNN layers that receive the error count and input tensor, produce an error injected tensor, and forward it to the next layer. We integrate these layers before and after the target fully-connected and convolutional layers to simulate the input and output errors. Dedicated non-trainable parameters in these layers control the number of injected errors, and error patterns are generated dynamically for each prediction.

6.3.2 Sanity-Check Implementation on Software

The software implementation of the spatial checksums necessitates one more neuron/filter in the fully-connected/convolutional layers. We need to calculate the required weight and bias values of the additional neurons/filters statically as a one-time post-processing step after training because modern DNN processing systems do not typically perform online learning after deployment; thus, the weights remain constant during the inference. While the fully-connected or convolutional layer outputs are being processed by the activation functions, we sum-reduce the outputs concurrently, and the generated check values are forwarded to the DNN output together with the predictions. If any of the check values significantly deviates from zero, we flag an error signal.

Temporal checksums do not require extra parameters, but we retain the accumulation of each input fiber (e.g., single-pixel location across different inputs) over the temporal batch which the checksum is integrated into. After each input fiber is accumulated over the temporal batch, we invert the sign of the accumulation and process it as an additional input. We accumulate the output in a similar manner and forward the results to the DNN output. The results are compared with the pre-calculated constants to detect errors.

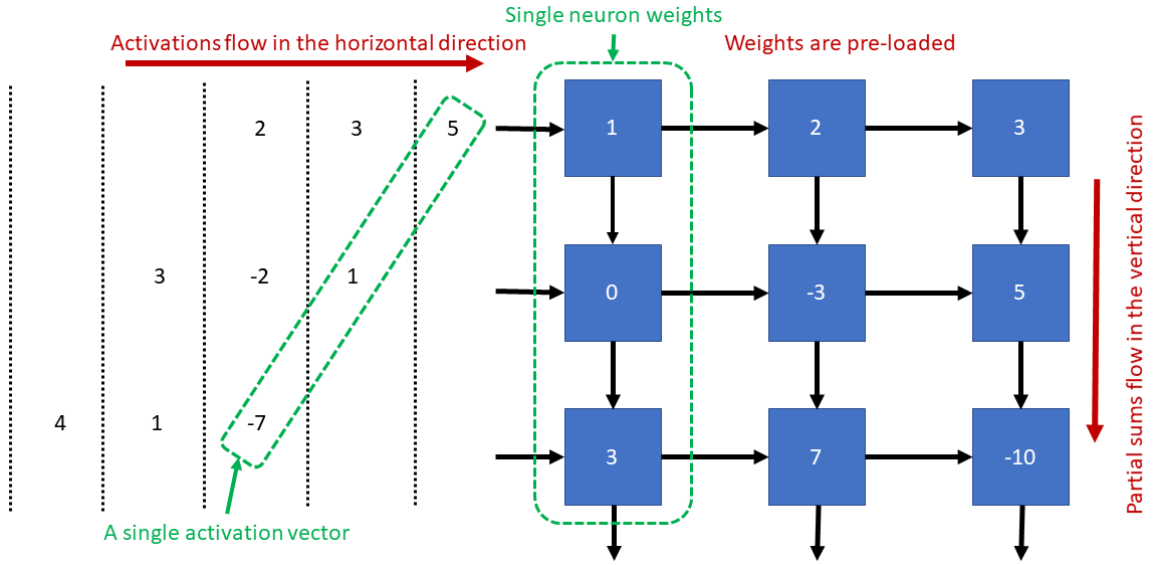

Figure 6.6. DNN accelerator with systolic array architecture.

6.3.3 Sanity-Check Implementation on Hardware

Sanity-Check requires additional computations; consequently, a pure software implementation may lead to a limited performance overhead under fixed-hardware resources as quantitatively characterized through performance measurements in Section 6.4. However, it is possible to eliminate the performance overhead with dedicated hardware extensions. We demonstrate this concept on a systolic array DNN accelerator. A systolic array DNN accelerator similar to Google's TPU [35] is shown in Figure 6.6. The architecture is an *S*x*S* MAC grid that can perform S^2 MAC operations in parallel. The weight values are first pre-loaded into the MAC units, and the input activations are shifted horizontally in the array. MAC units multiply the activations and the weight values, then update the partial sums. The partial sums are shifted vertically through the fabric to generate the final results.

We extend the MAC array with one extra column to balance the performance impact of additional neurons/convolutional filters, as shown in Figure 6.7. Output vector elements appear at the end of each column simultaneously or in the subsequent cycles based on the data-flow structure of the systolic array. We organize an adder tree to accumulate them concurrently to

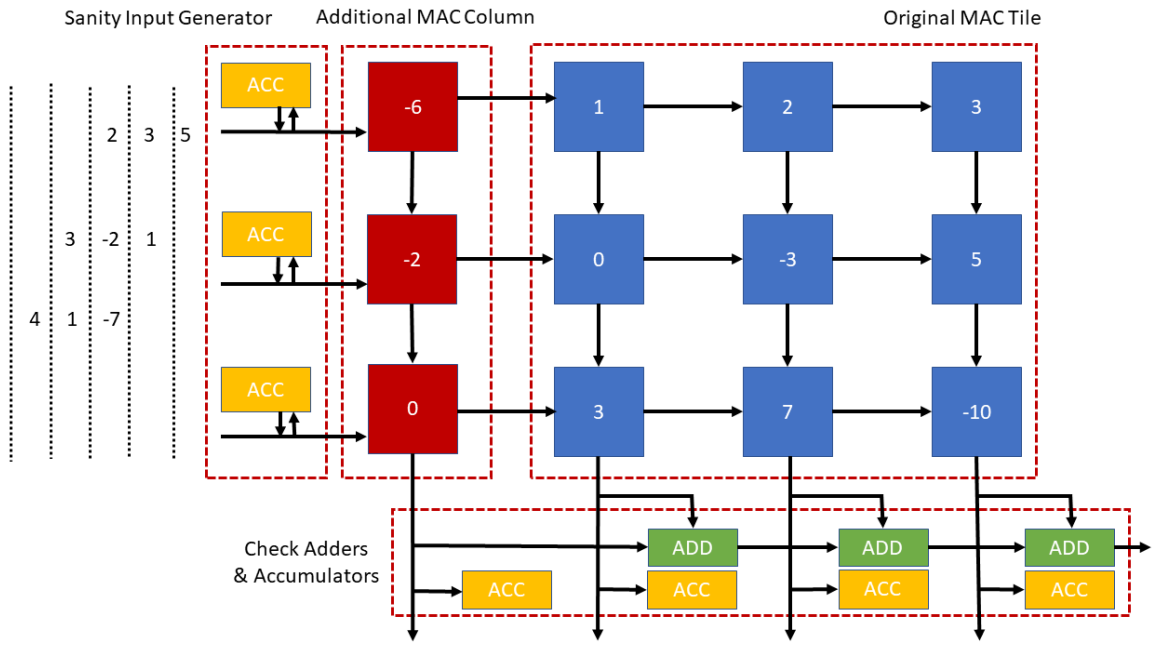

Figure 6.7. Sanity-Check hardware on systolic array architecture.

perform the checks for the spatial checksums embedded. A detailed diagram of the spatial checksum hardware is demonstrated in Figure 6.8. The proposed hardware synchronizes the output vector elements if they appear in the subsequent cycles (if the output vector elements are generated simultaneously, no synchronization step is needed), and the adder tree performs a sum-reduce operation to convert the output vector into a single value. Finally, the magnitude of the result is computed and compared with a small checksum threshold. If the calculated value exceeds the threshold, it causes the system to generate an error signal. The spatial checksum calculations do not result in throughput loss, but the latency has a negligible one cycle increase (e.g., 0.8% for the 64×64 array) due to the additional systolic array column embedded.

The temporal checksum hardware is demonstrated in Figure 6.9. Temporal checksum calculations are performed by accumulating input values and feeding the additive inverse of the accumulated value as the final input after a group of input vectors. The sanity input generation hardware (indicated with a blue background shade in Figure 6.9) keeps track of the number of processed input vectors through a shared counter and accumulates the same input positions across time through unique adders and registers. The module forwards the input activation vectors to

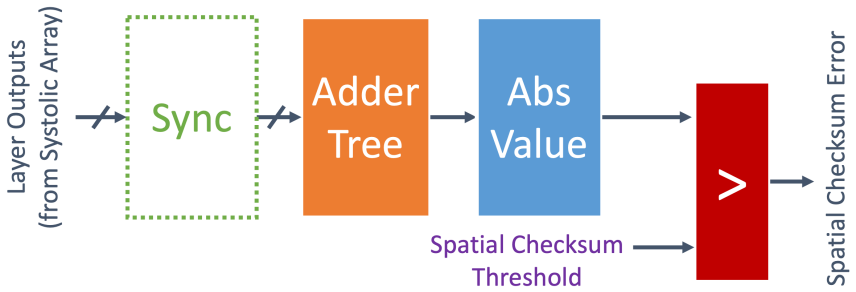

Figure 6.8. Details of the spatial checksum hardware implementation.

the systolic array in the normal operation cycles. After an input batch is processed, the additive inverse of the accumulation is provided to the systolic array as an additional input vector. If the vector elements need to be synchronized before feeding into the systolic array, a synchronization stage can be integrated before the systolic array connection. Check accumulators (indicated with a yellow background shade in Figure 6.9) keep track of the output count through the same shared counter, yet the control signals need to be delayed to accommodate systolic array latency. First, the systolic array output vectors are synchronized if they appear in the subsequent cycles. The same output positions are accumulated across time through unique adders and registers, and compared with the pre-determined constants for the temporal checksums. If any of the temporal check values deviate from the expected value by more than the specified threshold, an alarm signal is raised. The proposed implementation requires one extra prediction in each temporal batch, resulting in $1/(N+1)$ throughput reduction for input batch size N.

The checksum computations can be performed in residue arithmetic [93], obviating the need for larger adders and register files. This approach may introduce in rare cases aliasing; if these rare aliasing cases are not desired, larger bit-widths might be called for. Finally, assuming a MAC unit has n times larger area than that of an adder/accumulator, the area overhead can be approximated by the equation $(n+3)/(nS)$, resulting in a 2.5% overhead for a 64×64 systolic array for $n = 5$. The data width and the implementation type of arithmetic units will impact n, which would need to be determined for the target architecture. Section 6.4 verifies the minimal area and power overheads through hardware synthesis experiments.

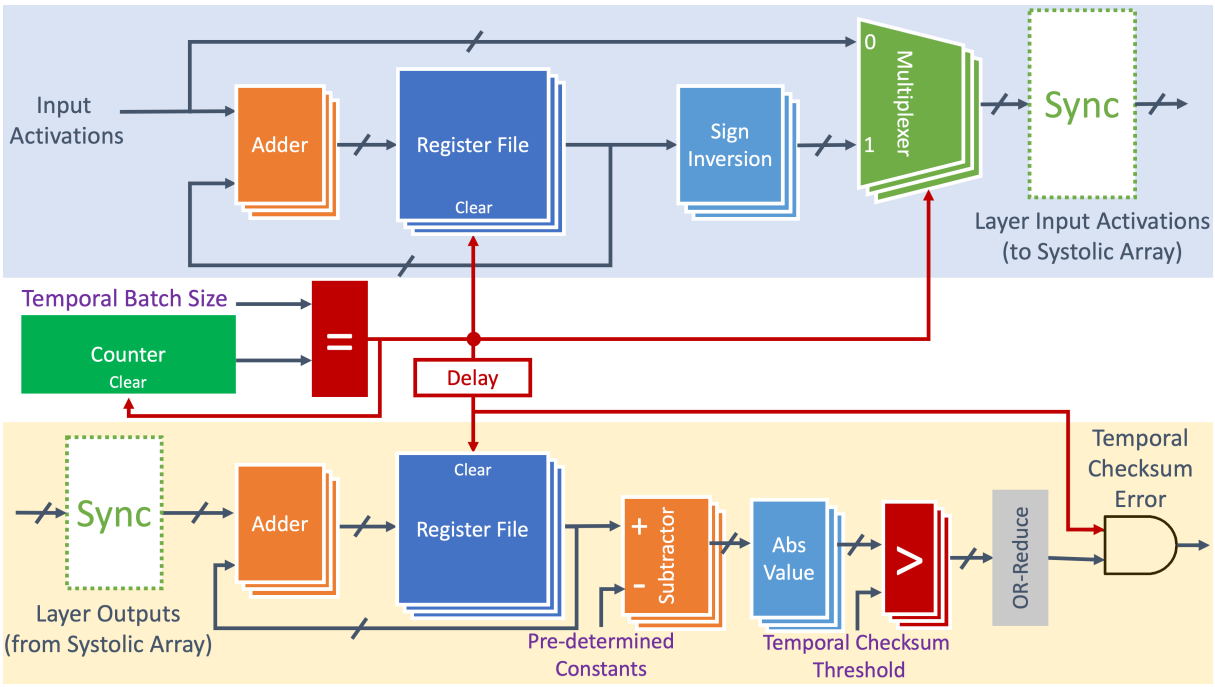

Figure 6.9. Details of the temporal checksum hardware implementation.

6.3.4 Baseline Methods for Comparison

We employ two spatially duplicated baseline methods to compare with *Sanity-Check*. Duplicated P.E. (Prediction Equality) checks if the end predictions are equal (e.g., same traffic sign) and indicates an error if they differ. Duplicated L.E. (Likelihood Equality) is stricter as it considers the likelihood equality at the outputs of the last *Softmax* layer, with every likelihood deviation notching up the error count. These methods offer an adequate resolution for detecting errors yet fall short of error correction unless further redundancy is employed.

6.4 Experimental Results

Figure 6.10 shows the error and error-caused misprediction coverage rates for single and multiple bit errors. We attain a perfect error-caused misprediction coverage with the combination of both checksums, even exceeding the coverage of Duplicated L.E., the strictest duplicated error detection method in our experiments. We also vary the checksum detection thresholds (from 2× to 100× of the maximum deviation seen in the training set) to tolerate the small errors that do

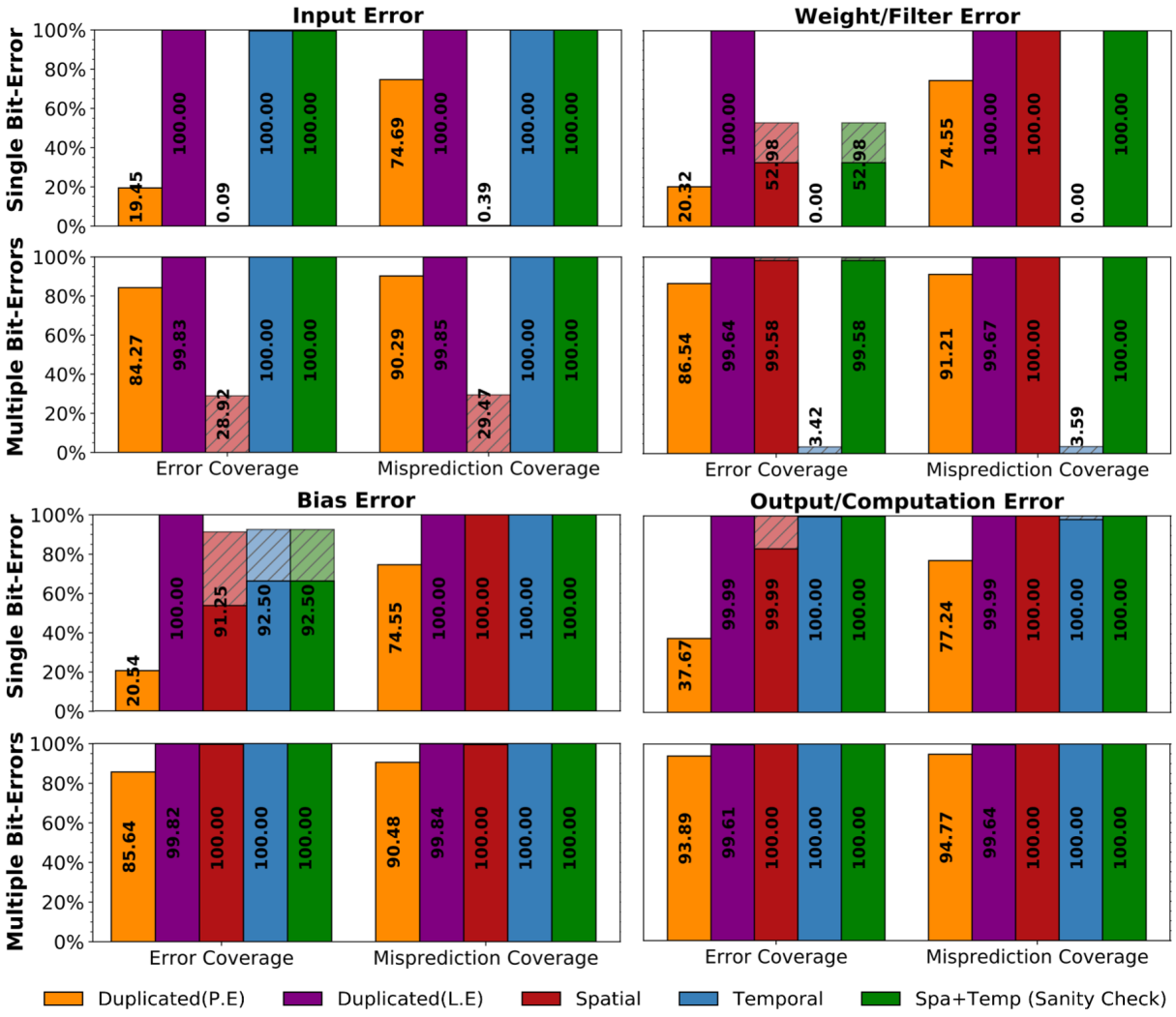

Figure 6.10. Error coverage and error-caused misprediction coverage rates for single and multiple bit-errors.

not result in misprediction; consequently, we reduce error coverage (as can be seen in the striped regions of the bars) but always obtain perfect error-caused misprediction detection rates. Finally, in the absence of an error, the outlined methods never lead to false alarms.

The misprediction coverage of our technique is particularly remarkable compared to the previous work in the literature. For instance, Li et al. [66] propose a symptom-based error detector where the activations are profiled after the training to determine the expected numerical range at each layer with the errors subsequently marked at inference time upon the violation of the expected range. Although their coverage for data types that incorporate a large numerical

Table 6.2. Memory and performance overhead for Sanity-Check checksums.

	Spatial	**Temporal**	**Spatial+Temporal**	**Duplicated**
# Parameters	0.09 %	0.00 %	0.09 %	100.0 %
# MAC Ops.	0.09 %	1.56 %	1.64 %	100.0 %
Meas. Perf. (CPU)	11.0 %	15.6 %	18.2 %	96.5 %
Meas. Perf. (GPU)	11.5 %	9.7 %	18.4 %	53.7 %

span is elevated (i.e., $> 95\%$), they state that coverage noticeably wanes on data types with limited range, such as a 16-bit fixed-point format with 10 fraction bits. In contrast, *Sanity-Check* attains perfect coverage on the same data type because of the precise checksums.

Duplicated P.E. correlates well with the actual mispredictions as no error is flagged until one network produces a different prediction. Interestingly, this feature hurts coverage rates as it is more likely for an error to escape detection by mutating into the same incorrect prediction rather than ending up with the exact wrong likelihood vector. Although the likelihood of mapping to the same wrong prediction at first glance promises to be rather low (e.g., $1/42 \approx 0.024$ for our 43 output network), our observations indicate that errors usually map to deviations from the correct prediction into only a limited subset of wrong predictions which in all likelihood neighbor the correct prediction in the decision space, thus reducing the detection rates of Duplicated P.E.

Table 6.2 reports the overhead of the suggested methods in terms of the number of additional parameters and the MAC operations. In addition, we measure the performance overhead of the pure software implementations on the CPU and GPU platforms. The proposed checksums have remarkably low overhead in terms of the extra parameters and MAC operations since *Sanity-Check* leads to an almost negligible 0.09% increase in parameters due to the extra neurons and filters, and the number of MAC operations grows by only 1.56% which is $64\times$ smaller than the cost of duplication. The measured performance overhead of the pure software implementation on a fixed hardware platform can still be $5.3\times$ smaller than that of duplication. As these platforms are not optimized for performing the required computations efficiently, a

Table 6.3. Area and power overhead of the Sanity-Check hardware modules.

	Accelerator	Systolic Array	Sanity-Check	SC/SA	SC/Acc
Area (mm^2)	2.922	1.559	0.048	3.08 %	1.64 %
Power (mW)	1170	238.3	13.1	5.50 %	1.12 %

Sanity-Check accelerator hardware design to attain perfect detection rates while shrinking the performance cost is further motivated.

We design the proposed hardware components (in Section 6.3.3) in Verilog HDL, integrate them into a systolic array-based DNN accelerator [37] and then perform hardware synthesis to characterize the area and power overheads experimentally. We configure the architecture to accommodate a 64×64 systolic array and 16-bit data types. The clock frequency is chosen as 1 GHz. We exclude on-chip data buffers (i.e., activation, weight) and perform synthesis with the Synopsys Design Compiler tool, using the Silvaco Open-Cell (15nm) technology and the Synopsys DesignWare libraries.

Table 6.3 demonstrates the area and power overheads of the proposed hardware modifications on the target accelerator. We report the area and power consumption values for the entire accelerator, 64×64 systolic array, and the *Sanity-Check* hardware. The *Sanity-Check* hardware includes the additional systolic array column, the adder tree for the spatial checksums, and the input and output accumulators for the temporal checksum calculations. In addition, we present the area and power percentage of the *Sanity-Check* hardware compared to the systolic array and the entire accelerator design. The synthesis results indicate an area footprint of 1.64%, and a power footprint of 1.12% for the *Sanity-Check* hardware when compared to the entire design. The area and power overheads are still quite minuscule when evaluated as a fraction of the systolic array, at 3.08% and 5.50%, respectively. We note that the rough area estimate of Section 6.3.3 is relatively accurate when compared to the synthesis results. The described hardware architecture will keep the overall performance overhead at around 0.08% in latency for

a 64×64 systolic array, and 1.5% in throughput for a batch size of $N = 64$. Overall, *Sanity-Check* can be integrated into a DNN accelerator as a hardware extension to deliver extensive error coverage at the cost of minimal area, power, and performance overheads.

6.5 Chapter Summary

The role of deep neural networks has expanded significantly in the safety-critical domains, including autonomous driving, healthcare, and industrial applications. While DNNs can perform accurately under minute numerical deviations, our analysis demonstrates that large numerical errors caused by hardware bit errors can critically impact accuracy and lead to unexpected system behavior. At the same time, the cost of traditional fault tolerance techniques is rarely palatable for DNN processing systems. We utilize the fundamental properties of DNN computations to enable real-time error detection in both fully-connected and convolutional layers through external invariants. The perfect critical error coverage of *Sanity-Check* is complemented by carefully designed hardware modules to minimize the area, power, and performance overheads of the *Sanity-Check* operations.

6.6 Acknowledgements

Chapter 6 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Low-Cost Error Detection in Deep Neural Network Accelerators with Linear Algorithmic Checksums," *Journal of Electronic Testing (JETTA)*, vol. 36, no. 6, pp. 703–718, 2020 ([1]). The initial conference version of this manuscript is published in Elbruz Ozen and Alex Orailoglu, "Sanity-Check: Boosting the Reliability of Safety-Critical Deep Neural Network Applications," in *Proceedings of the 28th Asian Test Symposium (ATS)*, pp. 7–12, IEEE, 2019 ([2]). The dissertation author was the primary investigator and author of both papers.

Chapter 7

Hardware Error Detection in DNN Accelerators via Internal Invariants

The abundant usage of DNNs (deep neural networks) in safety-critical domains such as autonomous driving raises concerns regarding the impact of hardware-level faults on deep neural network computations. As a failure can prove to be disastrous, low-cost safety mechanisms are needed to check the integrity of the deep neural network computations. This chapter introduces internal invariants in deep neural networks by introducing a custom regularization term in network training. We partition the outputs of each network layer into two groups and guide the network to balance the summation of these groups through an additional penalty term in the cost function. The proposed approach delivers twin benefits. While the embedded invariants deliver low-cost detection of computation errors upon violations of the trained equilibrium during network inference, the regularization term enables the network to generalize better during training by preventing overfitting, thus leading to significantly higher network accuracy.

7.1 Introduction

The external invariants proposed in the previous chapter are an effective solution to detect hardware errors in the linear computational stages of deep neural networks. On the other hand, the limited effectiveness of external invariants motivates us to use an alternative approach to embed internal invariants, which can operate across both linear and non-linear operations.

The learning process of deep neural networks facilitates the formation of internal invariants at various granularities for the purpose of coarse-grained detection and fine-grained localization of hardware errors. This chapter will primarily investigate coarse-grained internal invariants for the objective of hardware error detection.

The primary contribution of this chapter involves the introduction of a simple penalty term into the DNN loss function to train internal error-checking invariants and the utilization of these invariants for error detection. We first partition the outputs of each deep neural network layer into groups and form relationships across these groups by making use of additional penalty terms in the training process. The trained internal invariants are then utilized at inference time to check the anomalies and detect hardware errors since error incidents will result in the violation of the embedded relationships. Furthermore, we implement a comprehensive error injection framework and utilize it to compare and demonstrate the effectiveness of the checksums through exhaustive error injection experiments. Finally, we draw attention to our observations that indicate that the introduced penalty term delivers a rather useful and perhaps somewhat unexpected ancillary effect. It acts as a regularizer during DNN training and noticeably improves test set accuracy by reducing overfitting and improving generalization.

7.2 Error Checking with Computation Invariants

We aim to embed internal invariants into deep neural network computations and utilize them at inference time to detect discrepancies in the computations. The errors could be caused by various types of hardware-level faults, but independent of their provenance, these errors are detected as long as they violate the introduced invariant. Let us start our discussion by considering the following invariant for a particular DNN layer:

$$\left\| \sum_{i=1}^{S_l} h_i^l \right\|_F = 0 \qquad ,\forall l \tag{7.1}$$

In Equation (7.1), h_i^l denotes the output of the i'th computation unit in the l'th layer

and S_l represents the total number of computation units in the l'th layer. In the simplest case, Equation (7.1) can be considered to indicate that the outputs of a fully-connected layer always sum up to zero. As the summation will be a matrix for the convolutional layers, we use the Frobenius norm [204] to generalize Equation (7.1) for both fully-connected and convolutional layers by reducing the left-hand side to a single number. This invariant is quite useful for error checking since any single output error is always detected, or in general, any error pattern is detected as long as the individual errors do not cancel each other's footprint on the checksum by accumulating to zero.

The presented invariant requires the outputs of the computation units to span both positive and negative quantities so that they could potentially sum up to zero. While the computation units with a hyperbolic tangent (*tanh*) activation function produce both positive and negative outputs, the range of the very commonly used ReLU and Sigmoid activation functions is restricted to only non-negative quantities, thus precluding the direct utilization of the presented invariant.

We introduce an alternative invariant which allows us to perform a similar check for a set of non-negative quantities. Let us partition the computation units into two groups at each layer, sum each group separately, and set the norm of their difference to zero as in Equation (7.2). We refer to this invariant as the *balance checksum*. The balance checksum can detect all errors in a single computation unit; errors in multiple computation units are guaranteed to be detected as long as they violate the balance of the output partitions.

$$\left\| \sum_{i=1}^{\lfloor S_l/2 \rfloor} h_i^l - \sum_{i=\lfloor S_l/2 \rfloor+1}^{S_l} h_i^l \right\|_F = 0 \qquad ,\forall l \tag{7.2}$$

Although the balance checksum is useful for detecting errors, the introduction of such a mathematical property to neural network computations involves significant challenges. First, the balance checksum forces the layer outputs to adhere to a linear invariant without considering the norm operator. Linear invariants are straightforward to introduce in linear systems and are the key component for the error detection methodology achieved through external invariants in

Chapter 6. However, the scope of such an approach is strictly limited to the linear stages of the computation with data being left unprotected in the non-linear stages. In addition, the linear sessions are frequently interrupted by non-linear activation functions at the end of each layer. As the checksum generation and check operations should be performed within the boundaries of each linear stage, such an approach consequently incurs significant overhead. The mathematical introduction of such an invariant involves substantial challenges for non-linear systems, prodding us to follow a rather distinct approach to tackle this problem.

7.3 Training Balanced Output Partitions

Deep neural network training enables us to adjust the DNN parameters so that the network learns to perform the desired task correctly. The cost function plays an integral role in the training stage as parameter updates are continuously undertaken to minimize the cost function. One can set various training goals for the network through simple modifications on the cost function. An extra penalty term in the cost function is frequently used in the regularization methods to simplify the classifier complexity by guiding small weight values to zero. A simpler model obtained through regularization methods is less likely to suffer from overfitting to the training data and should exhibit superior generalization to future test examples. In a similar fashion, we integrate the balance checksum into our target DNN model by including it as a penalty term in the DNN cost function and guiding the network to minimize the checksum error while learning to identify the correct labels in the training data. Our overall cost function can be formulated as follows for a single training example:

$$-\sum_{c=1}^{M} y_{o,c}\log(p_{o,c}) + \lambda \sum_{l=1}^{L} \frac{1}{N_l} \left\| \sum_{i=1}^{\lfloor S_l/2 \rfloor} h_{o,i}^{l} - \sum_{i=\lfloor S_l/2 \rfloor+1}^{S_l} h_{o,i}^{l} \right\|_F^2 \tag{7.3}$$

The first part in Equation (7.3) is the categorical cross-entropy for the correct output labels. The second part is a mean square error for the group output differences. The function of λ is analogous to the coefficient used in the regularization methods. We use the hyperparameter

λ to control the impact of the penalty term on the cost function and prevent the penalty term from hindering the main learning task. In Equation (7.3), L denotes the number of layers and N_l the output size of each computation unit in layer l. $h^l_{o,i}$ corresponds to the output of the i'th computation unit in the l'th layer for input example o after the activation function. An exception needs to be taken for the outputs of the last layer, forcing us to extract $h^L_{o,i}$ before the application of the Softmax activation function. The sum of the Softmax activation function outputs is always normalized to 1 as shown in Equation (7.4) with the most likely class obtaining a rather elevated probability; therefore, it is not straightforward to integrate the balance checksum to the last layer after the Softmax function. We embed the balance checksum in the last fully-connected layer before the Softmax function, and as the Softmax layer outputs sum to 1, we check this property to protect data during the Softmax computation.

$$\sum_{c=1}^{M} p_{o,c} = 1 \quad ,\forall o \tag{7.4}$$

Finally, we partition the layers according to neuron (filter) indices; however, any equivalent partitioning scheme works as long as the trained scheme is used for error checking since the neurons (filters) are interchangeable before the training.

7.4 Error Checking at Runtime

Network training minimizes the additional penalty term in the cost function and consequently helps to balance the output partitions. As the network is required to satisfy the main classification task, the differences between groups typically fail to match zero exactly, instead slightly deviating from it. Our experimental observations indicate that as a result of training with the additional penalty term, the maximum deviation between the groups reduces by two orders of magnitude, thus providing us sufficient resolution to detect significant computation errors. We introduce individual threshold values for each layer and check if the difference between the groups deviates more than the determined threshold rather than focusing on a strict check of

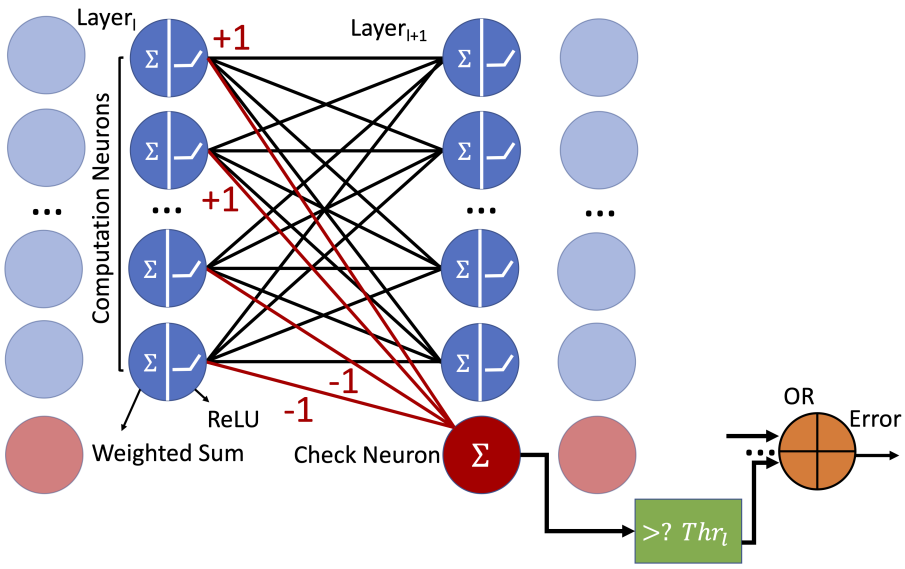

Figure 7.1. Checking the balance in fully-connected layers.

equality to zero. We learn the threshold values by profiling the training examples. We perform a full run on the training set to determine the maximum balance deviation at each layer and multiply these values with a small margin before setting them as the error thresholds.

To detect errors at runtime, we introduce some additional modifications in the DNN model. The correctness of the computations at each layer is checked by an extra computation unit in the next layer. For fully-connected layers, we employ an additional neuron in the next layer (check neuron) as shown in Figure 7.1 to check if the balance is satisfied. While the check neuron is connected to the outputs of all the computation units of the previous layer, we set the weight values for the first group connections as 1, and -1 for the other group. The check neuron accumulates both group outputs and calculates the difference at each prediction. We introduce 1×1 convolution filters (check filters) after each convolutional layer to check if the two group outputs are balanced. The channels of the check filter that convolve with the first group outputs are set to 1; the remaining channels are set to -1. In this way, the check filter group-wise accumulates the output channels of the previous convolutional layer and takes the difference. As the generated checksum is a matrix rather than a single value for the convolutional layers, we calculate the maximum value and use it for the threshold calculations and online error

checking. The outputs of the check neurons and check filters are forwarded to DNN outputs without being processed with the activation functions. The modified DNN produces a single check value for each layer together with the predictions, and we compare the check values with the thresholds to determine if an error has occurred.

7.5 Simulating Hardware-Level Faults on the DNN Graph

We design a comprehensive simulation tool to measure the effect of hardware-level faults on the DNN computation graph. Our simulation tools afford us the ability to inject bit errors into both activation values and weights during DNN execution. A bit error model is commonly employed to model the transient errors caused by SEUs (single event upsets) and also useful for simulating timing errors in sequential circuits. Our simulator injects activation errors via dedicated error injection layers embedded into the target network. Error injection layers produce error patterns dynamically for each prediction. We preprocess the weights to inject errors before runtime. The error-injected weights may affect multiple predictions until the weight values are refreshed by the correct ones. The simulator allows control of the error rate and the data width in the fixed-point format. We utilize our simulation framework to measure the DNN accuracy at different error rates and evaluate the performance of various error detection methods through exhaustive error injection experiments.

7.6 Experimental Method

We utilize a DNN model similar to AlexNet [49] to implement error detection methods and perform error injection experiments. Our target network model includes five convolutional and three fully-connected layers with the network being trained with the SGD (stochastic gradient descent) [205] optimizer (learning rate=10^{-2}, decay=10^{-6}, momentum=0.9, with Nesterov momentum) on the German Traffic Sign Recognition Benchmark Dataset (GTSRB) [50]. We up-sample GTSRB images, and adjust the output layer of the AlexNet network for

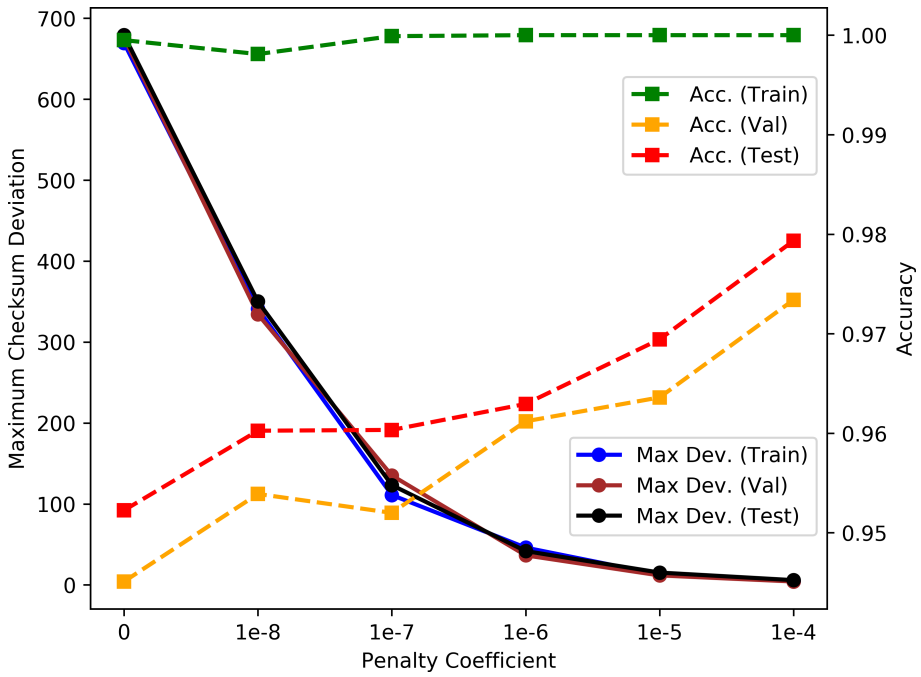

Figure 7.2. Penalty coefficient vs. maximum checksum deviation and accuracy.

GTSRB classification. We develop the target network, simulation tools, and error detection methods in Keras [202] and TensorFlow [203]. We use an NVIDIA GTX1060 GPU (graphics processing unit) for the experiments and Intel i5-8600K CPU (central processing unit) for additional performance measurements.

7.7 Experimental Results

We start by training the target model with a variety of penalty coefficients (λ) and observing the maximum difference between the layer output partitions. Figure 7.2 demonstrates that the penalty term is highly effective in balancing the output partitions through the entire network. The observed maximum deviation decreases by more than two orders of magnitude and delivers us sufficient resolution to detect even the small imbalances caused by errors.

We observe during network training another remarkable phenomenon; Figure 7.2 indicates that network accuracy tends to improve in line with an increase in the penalty coefficient instead of the penalty term limiting the learning efficiency. A 3% accuracy increase in both

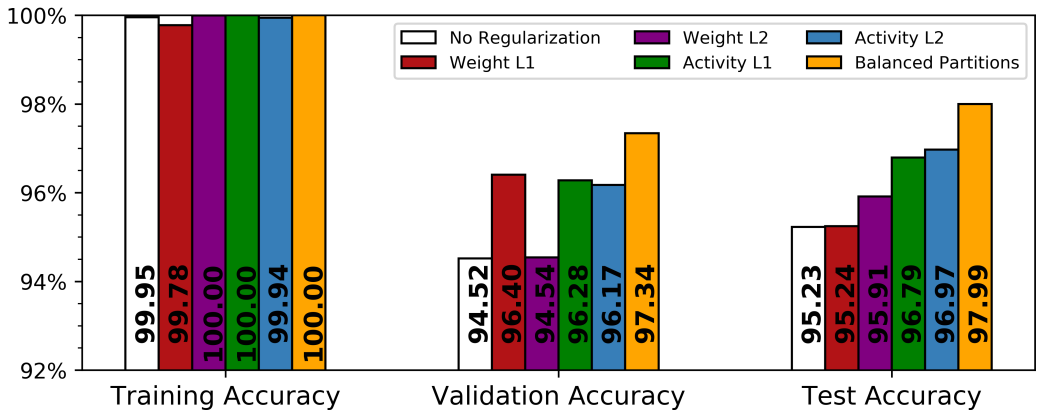

Figure 7.3. The comparison of regularization methods.

validation and test data compared to the base case without the penalty term can be observed. As we will explore shortly, the balance checksum acts as a network regularizer and helps to improve network generalization by ameliorating the overfitting problem.

We compare the accuracy increase of our methods to the conventional regularization methods used in practice. We train our model with L1/L2 weight and L1/L2 activation regularization methods and report the highest accuracy that we could obtain through a parameter sweep. We compare these results with the accuracy of the network under a balanced partitions constraint with $\lambda = 10^{-4}$, which delivers the best accuracy that we could obtain for our network. Figure 7.3 outlines the best accuracy values for training, validation, and test datasets. While almost all regularization methods have a positive impact on network accuracy, training the network with balanced output partitions delivers significantly higher accuracy than the standard regularization methods. If the layer outputs are considered as vectors, the introduced balance penalty term constrains the span of the generated output vectors by forcing them to be orthogonal to the vector $V_l = [1,1,1,....,-1,-1,-1]$ that has a length of S_l and populated by a sequence of $(\lfloor S_l/2 \rfloor)$ 1's and $(S_l - \lfloor S_l/2 \rfloor)$ -1's. As this constraint reduces the number of distinct features that can be extracted by one, we can expect a degradation in accuracy at a first glance.

To address this concern, we first extract the intermediate activations after non-linear functions at each layer for a randomly chosen set of 6400 test examples, then apply PCA [206]

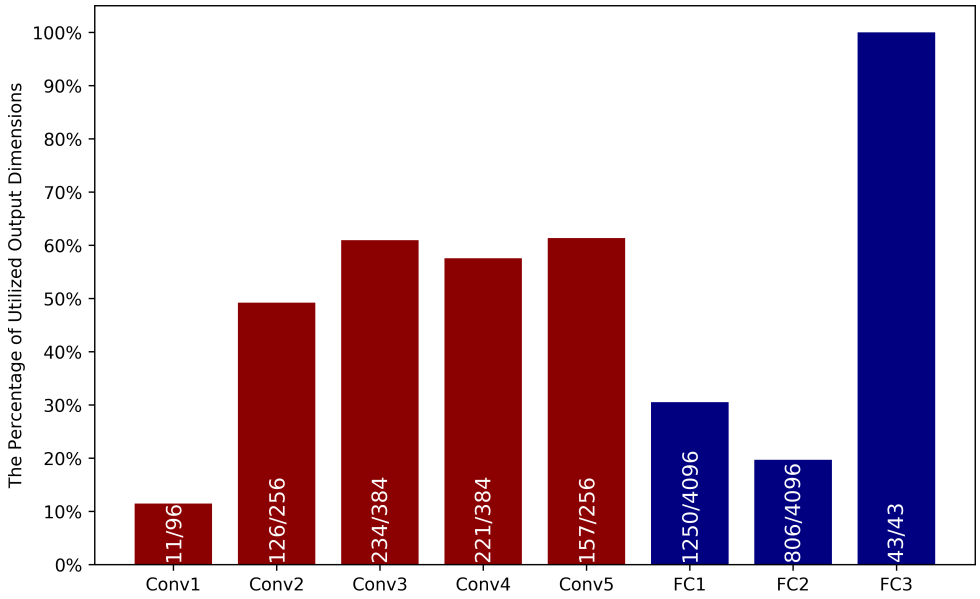

Figure 7.4. The percentage of utilized output dimensions at each layer.

(principal component analysis) to find the number of dimensions that account for 99% variance of the activation values at each layer output. The results in Figure 7.4 indicate that most layers are under-utilized as they extract much fewer features than the layer size (for instance, the first convolutional layer with 96 filters only extracts 11 distinct features). As the output data has much fewer useful features than the potential number of features that the layer can represent, reducing the maximum number of features that can be extracted by 1 proves to have no negative impact on accuracy.[1] On the contrary, the approach we propose auspiciously improves accuracy by reducing overfitting due to the following reasons. First, it restricts the activation range of the computation units and prevents the units from generating large activation values similar to Dropout [59] or activation regularizers since larger activation values will make it harder to balance the layer partitions. Second, correlating the layer outputs reduces model complexity and makes it harder to overfit to the training data.

We assess the error and error-caused misprediction detection capabilities of our approach in Figure 7.5. We utilize 4 different evaluation metrics: error detection precision, error detection

[1]The last output layer constitutes an exception as it is fully-utilized; we employ an extra neuron specifically for balancing the outputs and omit the additional neuron's output in the network decisions.

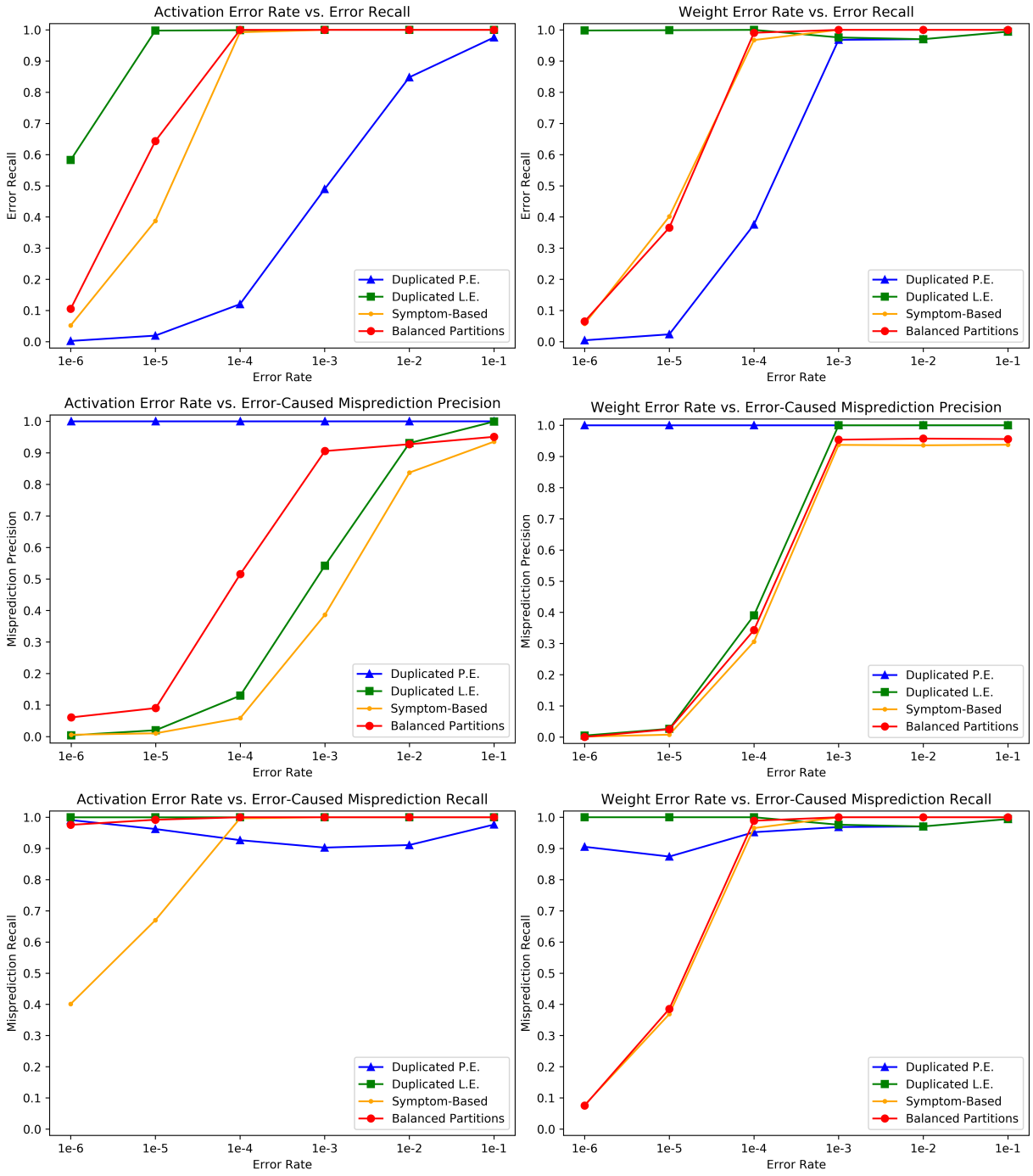

Figure 7.5. Error recall, error-caused misprediction precision, and error-caused misprediction recall for activation and weight errors.

recall, (error-caused) misprediction precision and (error-caused) misprediction recall. Precision and recall, commonly used measures of detector effectiveness, capture the accuracy of the detector among the positively identified examples and the detector coverage on the actual

positive examples, respectively. For instance, while error precision indicates the percentile of the cases the detector is correct among the identified error cases, error recall indicates the percentile of the error cases being detected. A mathematical definition of precision and recall is provided in Equation (7.5). TP, FP, and FN denote True Positive, False Positive, and False Negative examples, respectively:

$$Precision = \frac{TP}{TP+FP} \qquad Recall = \frac{TP}{TP+FN} \tag{7.5}$$

We implement three different error detection methods to compare with our approach. We design two duplicated models and employ the replicated DNN graph to check the consistency of the computations. The first method, *Duplicated P.E.* (prediction equality), checks if both replicated networks predict the same class (e.g., the same traffic sign) and it reports an error if the predictions differ. The second model, *Duplicated L.E.* (likelihood vector equality), checks the probability vector produced by the last Softmax layer and indicates an error if they are not equal. We also implement a *symptom-based error detector (SED)* outlined in [66] and compare it with our approach. The symptom-based detector profiles the range of the typical activation values, multiplies the range with a margin (e.g., 1.1 as reported), and detects the errors at runtime if an error causes the activation values to lie outside of these thresholds. Similarly, we experimentally observe that multiplying the differences with a margin (e.g., 1.4) before setting as error thresholds prevents the false alarm cases.

We inject bit errors into both weights and activations separately and report the indicated metrics for both activation and weight errors at a variety of error rates to comprehensively assess the delivered safety of the presented error detection methods. We use 16-bit data types for error modeling where 1 bit is the sign, and 11 and 8 bits are allocated to fraction bits in weight and activation values, respectively. We do not plot error precision rates as all error detection methods achieve perfect error precision (1.0) and never cause false alarms if no error is present.

The following points in Figure 7.5 merit further attention. First, our approach delivers

Table 7.1. Memory and performance overhead comparison.

	Duplicated	SED	Balanced Partition
Parameters	100.00%	0.00%	0.01%
Perf. (CPU)	100.00%	1.08%	1.00%
Perf. (GPU)	33.32%	2.95%	2.39%

consistently higher performance than a symptom-based detector in all metrics for all activation error rates. We observe a more significant advantage in low-error rate regimes. Our method almost always outperforms the symptom-based detector in weight error metrics as well. Duplicated P.E. always delivers perfect error-caused misprediction precision rates since no error is reported until at least one network experiences a misprediction; however, Duplicated P.E. has limited recall rates for both weight and activation errors. Duplicated L.E. detects almost all error cases due to its strict error detection criteria but consequently results in lower precision rates for error-caused mispredictions. Although both Duplicated P.E. and Duplicated L.E. have their advantages, the overhead of duplication restricts in practice its utilization as a safety solution for resource-intensive applications. Finally, we observe that our method is sensitive to the input distribution, as it also detects particular input errors.

We compare the memory footprint and performance overhead of the presented methods under fixed hardware architectures in Table 7.1 to provide a comprehensive overview. We report the overhead for Duplicated L.E. and Duplicated P.E. under a single heading as duplication dominates their overall overhead. While duplicated models require doubling of the network parameters, the symptom-based detector and our approach require almost zero memory footprint. SED achieves 93x smaller performance overhead compared to the duplicated models while the overhead of our approach can be 100x lower than the duplicated methods. Even on a GPU, appreciable overhead reductions can be observed by noting that SED is 11x and our method 14x smaller than that of duplication. The available hardware resources ameliorate the overhead of the duplicated models on the GPU, but such opportunities are rarely available on resource-limited

edge devices. As the experiments are performed on fixed hardware platforms, we do not report hardware overheads, yet one could expect duplication methods to neutralize the performance overhead at the cost of duplicated hardware resources. The small performance overhead of our method could be further reduced with much smaller overheads than duplication.

7.8 Chapter Summary

This chapter has presented an algorithmic error detection method for deep neural networks through internal invariants. These invariants are embedded into deep neural network computations by modifying the training process with an additional penalty term. The introduced invariants help us attain detection rates commensurate with state-of-the-art methods at only a fraction of their cost. Unlike the external counterparts presented in Chapter 6, internal invariants can operate across both linear and non-linear computational stages. The additional penalty terms are further observed to improve deep neural network accuracy by acting as a regularizer during training. The proposed technique in this chapter exemplifies how the unique characteristics of deep neural networks, such as redundancy and plasticity in the training process, allow us to embed computational structures such as invariants for hardware safety and reliability objectives.

7.9 Acknowledgements

Chapter 7 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Concurrent Monitoring of Operational Health in Neural Networks Through Balanced Output Partitions," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 169–174, IEEE, 2020 ([3]). The dissertation author was the primary investigator and author of this paper.

Chapter 8

Cost-Effective Rectification of Hardware Errors in DNN Accelerators

The cost of delivering safety and reliability through conventional fault-tolerance methods is oftentimes prohibitive for the deep learning hardware, as the available resources are stretched thin by the need to meet the computational requirements of modern deep learning algorithms. Fortuitously, the rules that govern the error resilience problem in DNNs (deep neural networks) deviate sharply from those in general-purpose computing. The inherent tolerance of deep neural networks to minor perturbations brings forth the possibility of embedding unparalleled resilience characteristics into deep learning hardware as long as the potential vulnerability caused by large-magnitude hardware errors can be mitigated. The accurate identification of errors through fine-grained internal invariants is complemented with approximate rectification techniques such as filtering or dropping of variables to maintain deep neural network accuracy even at excessive error rates. Unparalleled error resilience characteristics can thus be integrated into deep learning hardware while incurring costs that are a tiny fraction of those billed in conventional fault-tolerant designs.

8.1 Introduction

Innovative approaches for introspective error localization and approximate error amelioration are essential to boost the resilience characteristics of deep neural networks. Instead of

conventional and costly error detection techniques, deep neural networks can identify critical hardware bit errors via internal invariants learned through the training process. In this chapter, we demonstrate that error detection invariants can be crafted at finer granularities than the ones in Chapter 7 to allow precise error localization in the deep learning hardware datapath while necessitating no additional information redundancy.

In lieu of precise correction of the error perturbations, the impact of bit errors can be largely contained through approximate restoration techniques, which are carried out by dropping, clipping, or filtering variables that are contaminated with errors. The extreme effectiveness of these methods in maintaining model accuracy is experimentally established even at high error rates, while the implementation of the aforementioned techniques rarely incurs perceptible costs nor necessitates any information redundancy to deliver such error rectification. By simply snapping outsized error effects back to within the realm of minor numerical inaccuracies, procedures such as dropping erroneous values squash the error impact effectively and improve the bit error tolerance of deep learning algorithms by exploiting their inherent resilience characteristics to limited magnitude perturbations and complementing their inherent sparsity.

The approximate and resilient nature of deep neural networks yields the possibility of large-scale and efficient accuracy preservation by prioritizing the large magnitude bit errors and effectively attenuating them through the outlined approach. Such a perspective foreshadows fundamental breakthroughs for the error resilience problem in deep learning hardware and leads to strong safety and reliability characteristics at almost negligible costs.

8.2 Overview of Relevant Neural Network Characteristics

This section presents a brief overview of deep neural network computational characteristics that are of fundamental importance to the construction of novel error detection and mitigation techniques in deep learning hardware.

The behavior of deep neural networks is determined by the parameter configuration

learned through the training process. The flexibility of the training process in neural networks spawns a diverse set of models of comparable accuracy levels, even including ones that satisfy constraints of considerable strictness imposed on the computational graph. For instance, the forward pass of the model can be modified to rein in the propagated variable magnitude if the observed magnitude is unusually larger than the expected value. Neural networks can be trained effectively under graph constraints as long as the necessary information can be represented in the forward pass and the backward pass is able to be carried out efficiently. Invariants embedded into neural networks through the imposed graph constraints can then be utilized for error identification at inference.

A few other essential properties of deep neural networks can further introduce significant innovations to the error correction problem in deep learning hardware. Consider for starters that the final classification decision in a neural network is performed by finding the output position with the highest value in the last Softmax layer; an error in the intermediate variables is deemed therefore non-critical as long as the output position of the numerically largest value in the last layer remains constant. Secondly, the distribution of neural network parameters is observed to be clustered around zero, and to often span only a minute numerical range [28]. In a similar vein, not only are the majority of the observed activations restricted to small values as well but they exhibit high levels of sparsity furthermore.

The inherent resilience characteristics and the predictable numerical distribution of neural network variables bolster the feasibility and effectiveness of frugal approximate error correction. The value of an erroneous variable can be effectively estimated so as to minimize its impact on the output and maintain accuracy without necessitating perfect value restoration, thus significantly reducing the need for information redundancy for error correction operations.

The outlined properties of neural networks are fundamental to the context of our discussion as they catalyze innovation in the neural network fault tolerance problem to deliver superior resilience goals often with insignificant overheads.

8.3 Fine-grained Internal Invariants for Error Localization in Deep Neural Networks

The flexibility of the training process can be utilized to inject useful invariants in deep neural networks, which facilitate highly cost-efficient error detection and localization even across non-linearities. The embedding of such fine-grained invariant types is achieved by integrating custom weight or activation propagation rules in the neural network graph.

Critical bit errors are often associated with anomalously large numerical deviations in deep neural network variables. Large-magnitude spike errors are more likely to lead to a sizeable numerical deviation in the neural network outputs and result in an incorrect outcome. Therefore, a practical approach could interpret the anomalously large magnitude as a strong indicator of the critical bit error presence in a neural network variable.

Large-magnitude spike errors can be identified through the established numerical thresholds within a deep neural network. To illustrate, the expected numerical range of activations could be measured to determine numerical thresholds at each layer, and these thresholds can be utilized to locate erroneous variables as in [66]. Although the suggested technique requires minimal information redundancy, its performance strictly depends on how the numerical range of the data types is utilized at each layer. A single threshold per layer may not deliver sufficient resolution to locate all critical errors if the range of activations varies significantly within the layer. An alternative approach could be envisioned consisting of the utilization of unique threshold values for each activation, yet its significant cost in the number of additional parameters in the network dooms its practicality.

Our novel approach instead utilizes the numerical relationships among DNN activations to pinpoint anomalous deviations. Relationships such as equivalence or strong correlation could be contemplated, yet such relationships are hard to satisfy and further drastically reduce the overall information content of the involved variables. On the other hand, numerical order relationships among DNN activations could deliver a sound numerical bound while still retaining

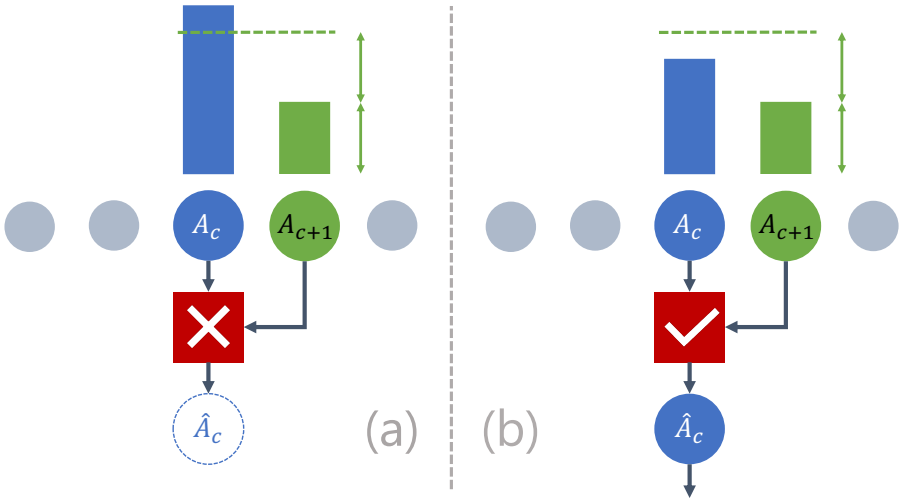

Figure 8.1. Anomaly detection with local magnitude comparison.

the flexibility to represent diverse information content. Such an approach resolves the problems mentioned earlier. First, each DNN activation is constrained with a unique bound, which leaves a smaller headroom for errors to escape detection compared to utilizing a global range for the entire layer. Second, no additional parameters are required as each DNN activation value is compared with another activation variable within the same layer.

The numerical order relationships facilitate a more precise localization of anomalous deep neural network variables compared to layer-wise thresholds. Furthermore, such relationships can be established within a neural network by imposing local variable propagation rules in the computational graph. For instance, an example propagation rule could impose a simple numerical order relationship across the neighboring variables where a weight or activation variable A_c is declared anomalous if its magnitude unusually exceeds the neighboring value A_{c+1} by a preset relationship (Figure 8.1-a), otherwise deemed normal (Figure 8.1-b) and propagated in the neural network graph with no modification.

Numerical order relationships can involve various design parametrizations and yield diverse expressiveness and resilience trade-offs. For instance, the anomaly limit established by the neighboring variable can be relaxed to boost the overall expressiveness yet at the cost of an increased likelihood of critical error escape. Furthermore, the anomaly limit set for a neural

network variable can be identified by multiple neighboring variables instead of one to provide additional flexibility in the numerical patterns that can be expressed within a layer.

The violation of these embedded invariants due to a hardware error can localize the erroneous variables at inference time with high precision. The fine-grained nature of such invariants engenders precise error localization even in the presence of multiple errors. After error localization, the novel error suppression methods to be detailed in the next section can be employed to contain error effects and maintain neural network accuracy gracefully even while suffering extreme bit error rates unimaginable in conventional fault-tolerant designs.

While the footprint of fine-grained invariants on the trained model is more noticeable than the coarse-grained counterparts introduced in Chapter 7 due to the increased number of imposed constraints, the inherent redundancy of modern deep learning models allows injection of such invariants into the model without requiring any additional information redundancy or impacting error-free model accuracy. Minor hardware extensions are necessary in the accelerator designs to check the invariant conditions and perform the mitigation actions efficiently, as discussed later in the next section. Fine-grained invariants can localize the errors with high precision, and when paired with novel error suppression methods, deliver complete algorithmic resilience for even extreme bit error rates of up to a few percent.

8.4 Maintaining Neural Network Accuracy with Approximate Rectification of Errors

We have illustrated that the problem of error detection could be resolved in an innovative manner by integrating computational invariants into neural networks and employing them for error detection in inference. The localization of the errors through the fine-grained invariants can be followed up by the suppression of the numerical distortion prior to the execution of each layer. Such suppression can be effected by dropping variables (setting to zero), clipping their magnitude to lie within the usual range, or passing the variables through filtering operations.

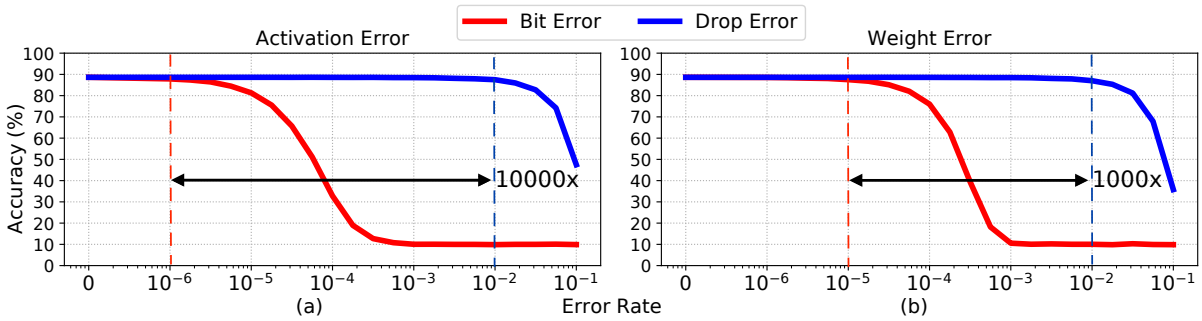

Figure 8.2. Bit-errors vs. drop errors on (a) activation and (b) weight variables.

Disproportionate error effects can thus be arrested at their tracks and reduced back to size before they have had a chance to diffuse in the network.

8.4.1 Error Rectification Through Dropping or Clipping Variables

The impact of bit errors on deep neural networks is shown to be asymmetric, with errors being mostly benign unless they lead to a significant numerical increase in variable magnitudes [66, 68]. We investigate this phenomenon in Figure 8.2, where we compare the accuracy of the identical DNN model (ResNet-18 [13] trained on CIFAR-10 [207]) by injecting bit-errors and drop errors into DNN variables. While bit-errors could have a variable magnitude effect, drop errors simply set the value of the impacted variable to zero. The reader will note that the DNN model exhibits a much more graceful response when the erroneous variables are dropped rather than when they suffer the effects of single-bit errors, with consequent resilience improvements of up to $1000-10000\times$. Previous studies confirm the resilience boosts under the drop errors [78, 103], while the relative sensitivity of DNNs to weight vs. activation errors shows variation [68, 78, 103].

The phenomenon in Figure 8.2 could perhaps be better explained if the algorithmic properties of DNNs are considered. First, previous studies [61] have established that the majority of DNN variables tend to cluster around zero. Such a characteristic is often boosted by certain design choices, including the common usage of Dropout [59], ReLU (rectified linear unit) activation function, and regularization techniques. As a result, correcting a value by setting

it to zero often introduces nothing but an inconsequential numerical discrepancy. Second, computation units (i.e., neurons or filters) are rife with redundancy in DNN layers [3], where the correlated units extract similar features. Thus, the drop of a particular variable rarely leads to a complete information loss of an extracted feature. Setting a variable imputed to be faulty to zero is consequently a safer option on average than allowing the bit-error to fester, although the numerical perturbation of the bit-error might be occasionally smaller than the perturbation introduced by the value drop.

The investigated phenomenon is extremely useful in practice. The erroneous values could be set to zero, or their magnitude could be restrained (i.e., clipped) to maintain accuracy even at high error rates without precise error correction, thus completely sidelining the use of information redundancy and expensive operations.

As an example, we demonstrate the proposed approach by combining fine-grained invariants with activation dropping in a simple suppression rule provided in Equation (8.1). We consider a layer l with n_l computation units (i.e., neuron or convolution filter). In Equation (8.1), $A_{(c,x,y)}$ and $\hat{A}_{(c,x,y)}$ signify the c'th unit output after the activation function and its propagated form through the suppression rule, respectively.[2] $U(i)$ is the step function where $U(i) = 1$ if $i > 0$; otherwise $U(i) = 0$. γ denotes a threshold coefficient that can be shared per layer or globally, with global sharing reducing the storage requirements commensurately. In simpler terms, the proposed rule outlined in Equation (8.1) propagates an activation value if its magnitude is smaller than the scaled version of its neighboring activation's magnitude; otherwise, it is dropped.[3] As the individual thresholds for each activation are determined through the magnitude of the neighboring activation, no additional value storage is necessitated.

[2]The x and y indices are utilized to indicate the spatial dimensions of the output feature maps in the convolution layers and not utilized in the fully connected layers.

[3]Indices out of bound are rolled over to the other end of the array in Equation (8.1), i.e., the first unit ($c = 0$) checks the last unit's ($c = n_l - 1$) output.

$$\hat{A}_{(c,x,y)} = A_{(c,x,y)} \times U(\gamma|A_{(c+1,x,y)}| - |A_{(c,x,y)}|) \qquad 0 \leq c < n_l, \quad \forall\{x,y\} \tag{8.1}$$

The presented suppression rule in Equation (8.1) is rather stringent since activation is immediately dropped after exceeding a single threshold, which is determined by the neighboring activation's value. This constraint could be relaxed further by introducing thresholds derived from multiple neighboring activations' values, and carrying out the dropping decision only when the majority (or all) of the conditions are violated. For instance, the suppression rule in Equation (8.2) is controlled by two gate conditions, and this rule allows a significant numerical variation among two neighboring activations (e.g., an edge type variation), yet precludes the propagation of single spikes when activation significantly exceeds both neighboring values.

$$\hat{A}_{(c,x,y)} = A_{(c,x,y)} \times \left[1 - U(|A_{(c,x,y)}| - \gamma|A_{(c+1,x,y)}|) \times U(|A_{(c,x,y)}| - \gamma|A_{(c-1,x,y)}|)\right] \qquad 0 \leq c < n_l, \quad \forall\{x,y\} \tag{8.2}$$

It could be seen that the suppression rules are applied in a channel-wise manner, as indicated by the iteration variable c in Equation (8.1) and Equation (8.2). Such an application order delivers a particular advantage against weight errors as well. An impacted weight value in a neuron or convolutional filter propagates into the entire channel of activations.[4] If the threshold activations are chosen from the adjacent channels, even the entire channel of anomalous activations in the output feature map could be easily filtered; thus, the weight error resilience of the network could be boosted in both fully connected and convolutional layers.

The proposed rules inhibit error propagation of critical bit-errors with a large magnitude, yet it is not guaranteed that non-erroneous activations in a pre-trained DNN model will escape

[4] A channel consists of a single neuron output in the fully connected layers.

unscathed through Equation (8.1) or Equation (8.2). Values that stand out will be squashed independent of whether they are erroneous or not; the consequent impact on non-erroneous activations needs to be ameliorated by incorporating these rules into the DNN model in the design phase. We follow the approach of integrating the suppression rules into DNN models through training, together with the ability to accomplish the desired classification task accurately. The unique characteristics of DNNs outlined in Section 5.2.1, including their inherent redundancy and plasticity in the training process, facilitate the embedding of suppression rules and the formation of alternative DNNs under such constraints without compromising their functionality.

8.4.2 Error Rectification Through Median Feature Selection

Noise mitigation is a well-investigated problem in the domain of image processing. An image could be subject to external noise (i.e., Gaussian or salt-pepper noise) in various real-life applications, yet even the most basic image filtering techniques perform remarkably well against noise by reducing its effect and maximizing the SNR (signal-to-noise ratio) [208]. Digital image filtering methods could be broadly categorized into linear and non-linear techniques. While linear filters (e.g., arithmetic-mean filter, Gaussian filter) could be implemented as convolution operations, they are commonly employed against noise models that have limited numerical impact, such as Gaussian noise. Order-statistics filters [208] of a non-linear nature (e.g., median filter) are demonstrated to exhibit a superior performance against large spike noise patterns, such as salt noise. For instance, the median filter can effectively remove error spikes, as its performance is impervious to a boost in error magnitude. Prior studies [66, 68, 69] indicate that errors that result in a sizable numerical increase in the activation/weight magnitude frequently prove critical. Such errors further exhibit a nature similar to the spike noise patterns in the image processing domain. Therefore, a similar filtering process in neural networks could dramatically inhibit critical bit-error propagation (as demonstrated in Figure 8.3) and translate into remarkable improvements in bit-error resilience.

Although non-linear filtering techniques have the potential to improve bit-error resilience,

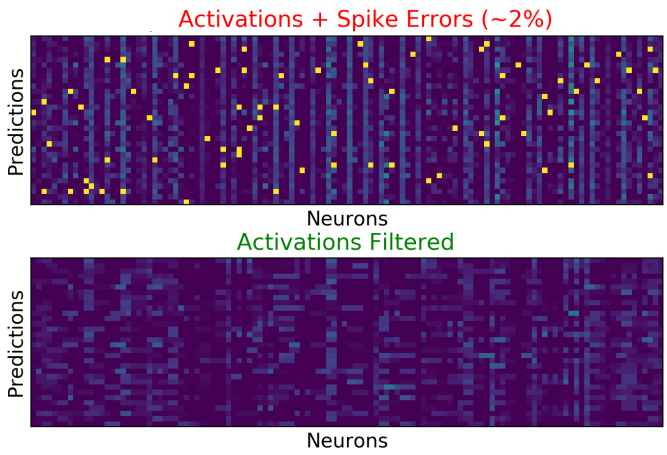

Figure 8.3. Spike error removal from DNN activations with median filtering.

several challenges should be resolved prior to the use of these techniques in neural network applications. First, image filtering techniques rely on the spectral difference between the image and noise patterns. The image patterns are required to be smooth so that they can be separated from noise, which inherently consists of high-frequency transitions. Unlike images, smoothness is not an inherently guaranteed property in the activation tensors; therefore, the direct integration of these filters into a previously trained model noticeably impacts accuracy by filtering out essential information content. Second, order-statistics filters are generally computationally costly (e.g., median filters require sorting operations), and they can not be merged with fully connected and convolution layers, unlike linear filter kernels. As a result, a naive integration of these functions is likely to lead to critical performance bottlenecks. In the remainder of this section, we try to address the first challenge by guiding neural network models to accustom to median filtering stages through adaptive training. Then we alleviate the performance bottlenecks of median operations with minimal, yet efficient hardware implementations on DNN accelerators.

We provide a conceptual description of the median feature selection technique in this section. It should be noted that the median filtering operations are performed prior to processing each DNN layer, and that the technique is applicable to input tensors of both fully connected and convolutional layers.

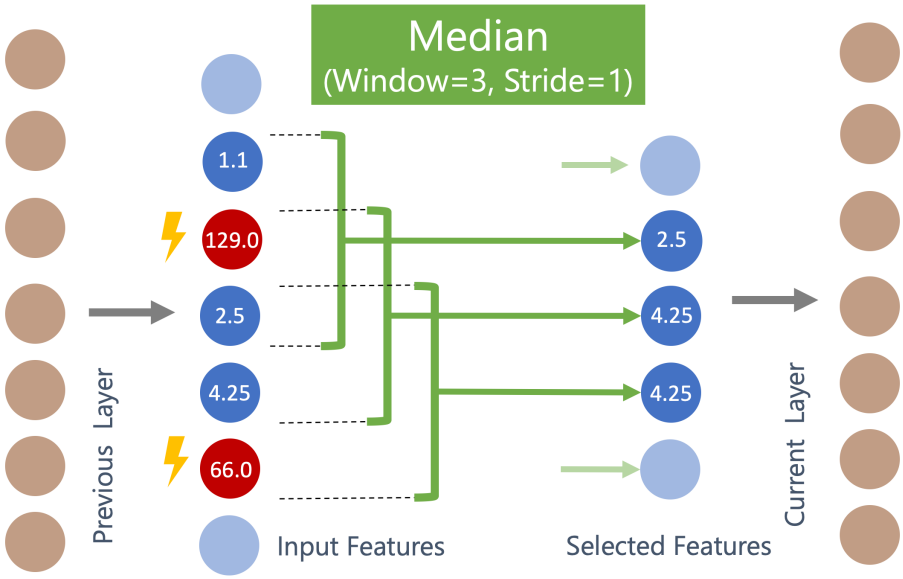

Figure 8.4. Median feature selection in the fully connected layers.

Application to Fully Connected Layers

Fully connected layers receive the inputs as a single-dimensional vector for each prediction. In fully connected layers, we apply the median feature selection operation as a sliding window on the input vector, then forward the selected features into the subsequent layer, as formulated in Equation (8.3) and visualized in Figure 8.4:

$$\hat{I}_i = med(I_{i-\lfloor w/2 \rfloor}, ..., I_i, ..., I_{i+\lfloor w/2 \rfloor}) \qquad 0 \leq i < n_{l-1} \tag{8.3}$$

In Equation (8.3), I and $\hat{I}$ are the original and selected input features of layer l. The utilized median filter size is denoted by w and the output size of layer $l-1$ by n_{l-1}. I is appended with $\lfloor w/2 \rfloor$ zeros in the corners before filtering so that $\hat{I}$ preserves the same length as I after the operation. We opt for small filtering window sizes (w) as they provide a good trade-off point between reliability improvement and resulting computational complexity of the median operation. For instance, the computation of the median value for $w = 3$ can be effected through three comparisons solely. Small window sizes further pose a minimal restriction on the layer outputs, and consequently, help us to attain accuracy levels competitive with the original model

wherein no error is present.[5] The second design parameter is the stride size (s), which indicates the amount of shift between consecutive median windows. For instance, when the stride size is equal to the filtering window size ($s = w$), the receptive fields of the median filters do not overlap. While the usage of non-overlapping windows has a positive impact on reliability by limiting the number of median window outputs that an error can propagate to, it also aggressively reduces the layer output size. We prefer to use single-element strides ($s = 1$) to preserve the output size. The single-element strides result in overlapping receptive fields and thus minimize the additional redundancy required for implementing the median filtering schemes by assigning more than one objective to each involved deep neural network variable. Although an error has a chance to propagate to multiple filter outputs due to overlapping receptive fields, a large error will still be filtered in all windows by the median function.

A distinct subset of neuron outputs could be selected by the median filters in different inference examples. This phenomenon ensures that each neuron can undertake a valuable role in the overall functional behavior of the deep neural network; thus the redundancy needs for implementing median filters are minimized.

Application to Convolution Layers

Convolutional layers receive inputs as a 3-dimensional tensor for each predicted example; therefore, the outlined median feature selection technique needs to be reconsidered before its application in the convolutional layers. Filtering could be performed in any dimension of the input or even on the multi-dimensional patches of the input feature map. However, some of these techniques may require different implementations than the case for the fully connected layers.

Convolutional layers are frequently carried out in the form of matrix multiplication in modern processing platforms. Dense linear algebra libraries provide efficient implementations for matrix multiplication (e.g., cuBLAS [209]), and certain data-flow architectures (i.e., systolic array) [35, 37, 210] are specialized in matrix multiplication operations. As a result, it is possible

[5]This concept could be appreciated by picking relatively small ($w = 1$) and relatively large ($w = n_{l-1}$) filter sizes, and observing the information loss at the filter outputs.

to attain a dramatic speed-up in the convolutional layers when they are processed as matrix multiplication operations. However, such a transformation requires input feature maps and the conversion of convolutional filters into suitable representations prior to the actual processing. The input feature map is first allocated into small 3-dimensional patches (with the patches possibly overlapping for small convolution stride sizes), flattened into single-dimensional vectors, and row-wise concatenated to form the feature matrix. Meanwhile, convolutional filters are also flattened into single-dimensional vectors and column-wise concatenated into a matrix. Finally, the convolution operations are simulated as matrix multiplication, and the output feature map is generated in the form of a matrix.

We apply median feature selection on the flattened version of the input patches before feeding them into matrix multiplication. First, our decision simplifies the filtering technique as it establishes an operational equivalence across fully connected and convolutional layers. As we will describe shortly, it allows us to facilitate the same hardware architecture to perform median filter selection operations in both fully connected and convolutional layer inputs. Second, it provides us freedom in regards to the application direction of the feature selection operations. Input patches and filter weights could be flattened in different orders of the dimensions by the DNN mapper (compiler) as long as they are consistent; therefore, the feature selection dimension could be easily changed on the software without necessitating any hardware modification.

We further observe that flattening the feature maps in a manner where the same pixels from different channels end up in adjacent positions offers a particular advantage against convolutional filter errors. A convolution layer filter is responsible for constructing a channel of the output feature map. As a result, a critical error will likely impact all entries in the corresponding output feature map channel, as demonstrated in Figure 8.5. Filtering in the spatial dimensions (x or y) of a single channel may not be as effective against weight errors because of the potential impact on all entries in the same channel. If the input feature map is flattened in a way that the same pixels from different channels reside in adjacent positions, it is guaranteed that no more than one element from a channel would be contained within a median window; therefore, the

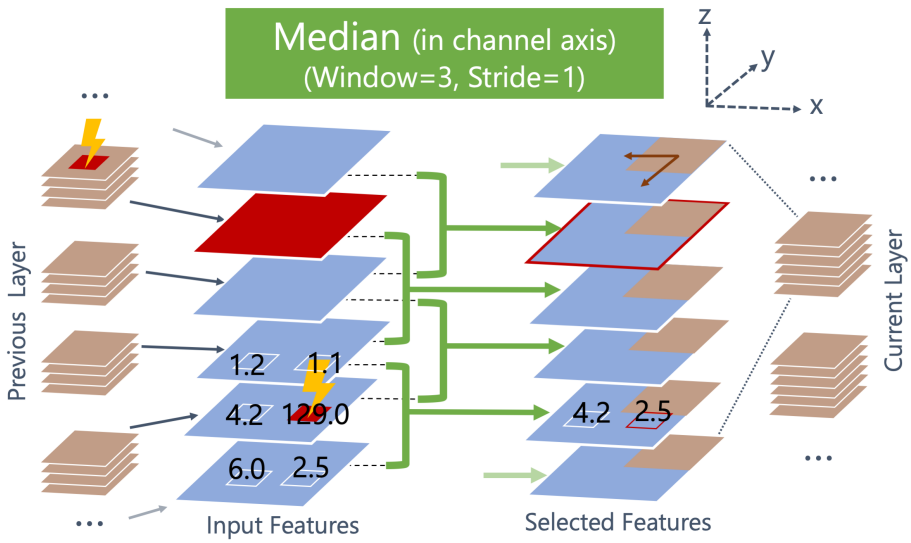

Figure 8.5. Median feature selection in the channel dimension of the convolutional layers.

impacted entries could be effectively filtered from the feature map. In the remainder of our analysis, we consistently perform median filtering in the direction of the channel axis (z) in convolutional layers due to the outlined advantage, as visualized in Figure 8.5. Fully connected layers can only be filtered in the channel axis since the neuron output is a scalar, thus inherently offering protection against the weight errors in neurons.

Moreover, performing filtering in the channel dimension (z) is not only preferable but also frequently necessary to attain competitive error-free DNN accuracy. Median filtering in the spatial dimensions is manageable when the spatial dimensions of each feature map channel are sufficiently large, and the input tensor exhibits a smoothness property in nearby pixels. DNN input (e.g., an image) and the output of the initial convolutional layers in a DNN do partake of such properties, yet modern DNN architectures minimize the spatial dimensions of feature maps quite aggressively throughout the network through the use of pooling techniques or strided convolutions [61]. Thus, it is common to see relatively small feature map channels (i.e., 4×4) in the later convolutional layer outputs. As a result, performing the median operation in such small spatial dimensions dramatically reduces the information content of each output channel in the later convolutional layers and prevents the DNN from attaining a competitive error-free

accuracy. On the other hand, the number of channels usually increases in the later layers in modern DNN architectures (e.g., ResNet [13]). A large number of channels in the later layers engenders effective median filter operations in the channel axis with no significant limitations on information capacity. For instance, when the median filters ($w = 5$) are placed in the channel dimension of LeNet-5 [211], the model exhibits no error-free accuracy loss after training over the case with no filters, yet the model trained with median filters in either of the spatial dimensions experiences around 10% accuracy loss.

The proposed technique could also be utilized to filter neural network inputs without any extra modification if the input images have sufficient resolution, thus protecting the system against input errors that could stem from low-cost input devices (e.g., camera noise). Unlike the intermediate feature maps, it is preferable to filter the input images in the spatial dimensions as neighboring pixels exhibit smoothness and the number of channels in an image is limited (e.g., only three channels for an RGB image).

If the neural network outputs require protection, it could be attained with a small modification in the network. As the output neurons are trained to be exclusive to attain a competitive classification accuracy, the proposed technique (with $s = 1$ and no layer extension) will fail to protect the output layer. The most straightforward solution involves the replication of output neurons and employing larger stride sizes to assemble a complete modular redundancy (i.e., for $w = 3$, each output neuron is replicated three times, and $s = 3$). This modification will introduce only a negligible number of additional parameters and operations when the number of output classes is small.

8.5 Deep Neural Network Training with Graph Constraints

Fine-grained invariants and error rectification rules can be embedded into neural network layers by imposing custom propagation rules in both the forward and the backward pass of training and ensuring that the deep learning model attains a competitive accuracy within the

confines of these rules. Invariant integration incurring neither additional information redundancy nor baseline accuracy degradation can be achieved through inherent model redundancy and training process flexibility.

8.5.1 Training DNNs with Anomaly Detection and Suppression Rules

We perform DNN training with the anomaly detection and suppression rules integrated after each convolutional and fully-connected layer to ensure a seamless adaptation to these operations. However, training DNN models with these rules involves particular challenges. We will describe some of these challenges and present various techniques to deal with the difficulty of training in this section. It should be noted that the proposed rules alter the gradient flow in the backward pass. A gradient on the output activation variable $\hat{A}_{(c,x,y)}$ needs to be back-propagated to all variables that take a role in the generation of $\hat{A}_{(c,x,y)}$, namely both $A_{(c,x,y)}$ and $A_{(c+1,x,y)}$ in the case of the rule specified in Equation (8.1).

The main difficulties in training stem from the flat regions and the discontinuities of the step function. The issue can be clearly demonstrated through the derivation of the back-propagation rules for Equation (8.1) (from $\hat{A}_{(c,x,y)}$ to $A_{(c,x,y)}$, and from $\hat{A}_{(c,x,y)}$ to $A_{(c+1,x,y)}$). Let us assume a single DNN output O, then utilize the chain rule to derive the corresponding gradients. We make simplifications based on the fact that the derivative of the step function is zero everywhere except for being undefined when the input is zero:

$$\begin{aligned}\frac{\partial O}{\partial A_c} &= \frac{\partial O}{\partial \hat{A}_c} \times \frac{\partial \hat{A}_c}{\partial A_c} \\ &= \frac{\partial O}{\partial \hat{A}_c} \times [U(\gamma|A_{c+1}| - |A_c|) + A_c U'(\gamma|A_{c+1}| - |A_c|)] \\ &= \frac{\partial O}{\partial \hat{A}_c} \times U(\gamma|A_{c+1}| - |A_c|)\end{aligned} \tag{8.4}$$

The back-propagation rule in Equation (8.4) is intuitive, as the gradient in $\hat{A}_{(c,x,y)}$ is back-propagated to $A_{(c,x,y)}$ as long as $A_{(c,x,y)}$ is not dropped; otherwise, the received gradient becomes zero. The gradient can be derived for the gate activation variable similarly as in Equation (8.5):

$$\begin{aligned}\frac{\partial O}{\partial A_{c+1}} &= \frac{\partial O}{\partial \hat{A}_c} \times \frac{\partial \hat{A}_c}{\partial A_{c+1}} \\ &= \frac{\partial O}{\partial \hat{A}_c} \times [A_c \times U'(\gamma \times |A_{c+1}| - |A_c|)] = 0\end{aligned} \tag{8.5}$$

Equation (8.4) and Equation (8.5) demonstrate that the gradient back-propagation imposes certain challenges on both input ($A_{(c,x,y)}$) and gate ($A_{(c+1,x,y)}$) activation variables. First, $A_{(c,x,y)}$ never gets updated if it always ends up being dropped. Second, $A_{(c+1,x,y)}$ receives no gradient at all through the value it has controlled; therefore, it is independently optimized despite its significant role in the generation of $\hat{A}_{(c,x,y)}$. The step function is desired in the forward pass because it effectively eliminates the propagation of the anomalous activations, yet hinders the gradient flow in the backward pass in its current form and precludes efficient training.

A similar problem has been previously tackled in previous work in BNNs (binary neural networks) to deal with the gradient back-propagation problem of the *sign* function. The sign function has zero gradients almost everywhere, except for the undefined gradient when the input is zero, as does the step function just being discussed. Hubara et al. [212] utilize the sign function in the forward pass, but they back-propagate the gradients from the sign function output to sign function input by approximating it as an *identity* function in the backward pass (straight-through estimator). In an analogous manner, we propose using a function that is structurally similar to the step function, yet exhibits better differentiability characteristics in the backward pass. The well-known *sigmoid* activation function fulfills the outlined criteria. While being structurally similar to the step function, the sigmoid gradients diminish more gradually as the magnitude of the input increases. As a result, gradients could back-propagate through the sigmoid if its input is close to zero. We utilize the step function in the forward pass, yet derive a back-propagation rule by replacing the step function with the sigmoid and enable a more efficient gradient flow.

We further observe that activation regularization usually has a positive effect on training. An activation variable could be stuck at significantly large values and may fail to be updated through gradients if it saturates the sigmoid inputs. Adding its magnitude as a penalty term in

the cost function, e.g., L1 or L2 norm of the activations (before suppression), reduces the large activation magnitudes back to the appropriate range, thus enabling their continuous updating through output gradients.

The proposed techniques improve the gradient flow and make it possible to train DNN models despite the presence of wide-spread discontinuities and plateaus that hinder the effective flow of the gradients. However, more iterations might still be required to train deeper DNNs in more challenging datasets when these rules are integrated. We further significantly speed up the process by training more complex models in a two-stage process. In the first stage, we enforce certain restrictions on DNN weights to allow seamless integration with the described suppression rules, although the first stage of training is carried without explicitly embedding these rules. The restricted model is faster to train, yet the imposed restrictions also limit its degrees of freedom. As a result, the baseline accuracy of the constrained model suffers in comparison to the one that can be attained through the unconstrained model. In the second stage, we first integrate the feature suppression rules into the model, relax the enforced weight constraints, and fine-tune the network until it fully recovers the baseline model accuracy.

Let us consider the two-gate rule provided in Equation (8.2). Each activation needs to satisfy at least one of the criteria imposed by its neighbor activations to propagate its value into the next layer. We allocate the neighboring neurons/filters into pairs at each layer and tie the weight connections of each pair so that both neurons/filters become equivalent. As a result, each neuron/filter pair produces the same output, and they further receive matching gradients in the backward pass. Although the imposed constraint effectively reduces the information content at each layer by half, it also ensures that the trained model will work seamlessly with the suppression rule in Equation (8.2) at the end of the first stage. As each neuron/filter output will be repeated by at least one of its neighbors, it is guaranteed that no information content will be suppressed in Equation (8.2). In the second stage, we first integrate the feature suppression rules after each layer, lift the equivalence restriction on the weights, and perform additional fine-tuning to augment the information content and recover the remaining accuracy while the

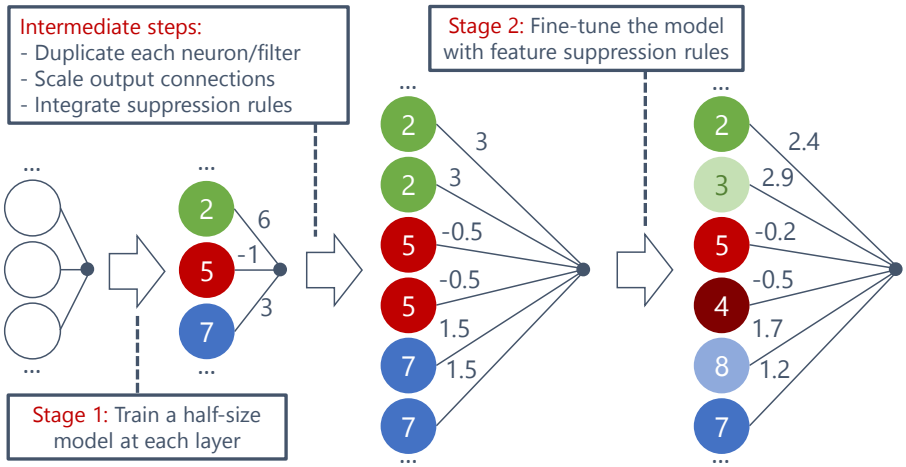

Figure 8.6. Training DNNs in a two-stage process.

initially equivalent neurons/filters are being differentiated. We observe that most of the accuracy content of the model could be attained in the first stage, and the second stage could quickly bridge the accuracy gap with a small number of fine-tuning iterations while assuring seamless compatibility with the feature suppression rules.

The memory and computational requirements of the first stage could be further reduced by 50% by merely training a model with half the desired size at each layer, then duplicating each neuron/filter and scaling the outgoing connections by half. This approach guarantees that the same outcome is attained at a smaller computational cost at the end of the first training stage. The overall design flow is visually demonstrated in Figure 8.6.

Finally, the output layer needs to be tackled differently than the other layers since the output layer is trained to produce results resembling a one-hot encoding format. We, therefore, do not integrate these rules into the last layer in training. Instead, output units are duplicated after training, and the suppression rules are applied at inference, which guarantees that specific outputs will form a majority within the filtering window and pass through these rules unaffected, and thus could be used to recover the classification result. This modification guarantees that the output layer will be protected against weight and activation errors while requiring a minimal increase in the parameters and operations.

8.5.2 Training DNNs with Median Feature Selection Rules

The median operation will suppress single spiking neuron outputs; therefore, the layer outputs should be locally smooth to work with median feature selection stages. The described smoothness property could be introduced in the target model by incorporating the median feature selection operation in the training phase. DNN libraries usually offer off-the-shelf implementations for the median [213], or other fundamental operators (i.e., min and max) that can be used to construct the median functionality [203]. The utilization of library functions ensures seamless training of the target model with the existing library routines.

We implement the median feature selection operations on the abstraction of the utilized deep learning framework [213], integrate it into target DNN architectures, then perform standard DNN training by simulating the median filter behavior in both the forward and the backward phases of the training. As a result of the introduced operators, only the median value in a window is forwarded to the subsequent layer in the forward pass, and a gradient on the median operator output is back-propagated only to the contributing input value.

The described median back-propagation update rule boosts correlations between layer outputs because of a striking "chase phenomenon" between the neuron outputs. This point can best be illustrated with a simple example on a median-3 window. When a median-3 output needs to be increased to reduce the loss function at the neural network outputs, the median input value of the filter (e.g., I_1) is updated through back-propagated gradients until it exceeds the maximum input value (i.e., I_2). When I_2 becomes the new median, it receives the output gradients and gets updated until it regains its position as the maximum value. When I_1 finally falls behind I_2, it continues to be updated, and so on. The reader will recognize that this process will result in a mutually reinforcing increase between neuron outputs until the gradient is stabilized at the filter output. As a result, neurons will learn to "fire together" and produce correlated outputs that could propagate through the median operator. The correlation behavior could be observed on the activation patterns of a fully connected layer in Figure 8.7 where distinct neuron

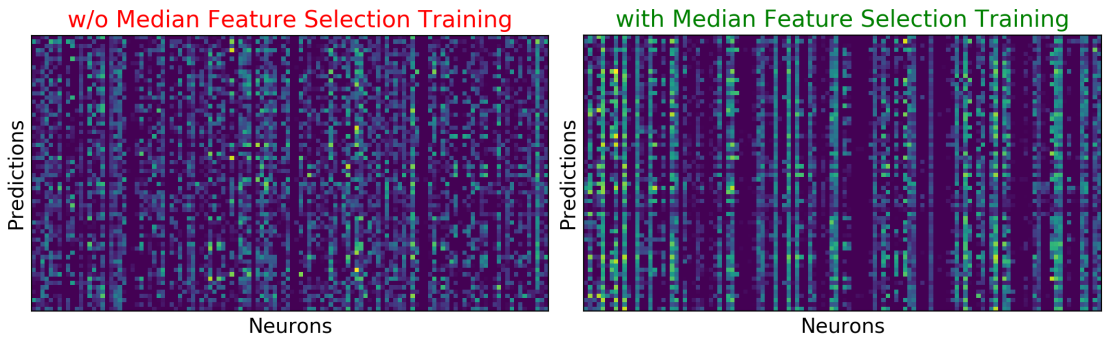

Figure 8.7. The impact of median feature selection training.

outputs (a column represents the outputs of a neuron) seem independent without such training, yet the median feature selection training leads to activated output bands that involve multiple neighboring neurons to propagate the features through the median functions as can be seen on the right-hand side of Figure 8.7.

Median models can be trained from scratch. In addition, we have observed that two initialization techniques help to reduce the training times of the median models since median filters might slow down the training process due to the additional operations. First, the median model can be initialized with the weights of the baseline model (trained without filters) or a median model with a smaller window size. As a result, the model can recover the required accuracy with fewer iterations since the model behavior is embedded through initialization, and only a fine-tuning step is required to adapt to the effect of median filters. Alternatively, a smaller template model with fewer neurons/filters at each hidden layer could be trained without filters, and its neurons/filters explicitly replicated to work with the median filters by forming a majority within each window. The constructed model attains the accuracy of the small template model, which is generally less than the baseline accuracy, yet a quick fine-tuning stage after filter integration enables us to bridge this accuracy gap.

We observe that the error-free accuracy of the median models is similar to and even sometimes may slightly exceed that of the baseline models, particularly if the baseline model contains sufficient redundancy. It is conceivable that the higher accuracy values occur due to

statistical variations in these scenarios, yet a more detailed experimental analysis is needed for any conclusions of certainty. Overall, median feature selection reduces the information capacity of the layers by encouraging neuron correlations; however, this constriction is rarely of practical import in modern neural networks, thus lending credence to arguments in Section 5.2.1 regarding the redundancy characteristics of deep neural networks. As a concrete example, an intriguing observation in Chapter 7 indicates that modern DNNs extract a small number, rarely exceeding a fraction of the layer size, of unique (orthogonal) features at each layer. Median feature selection utilizes the redundancy of the layer output space to construct a redundancy scheme at no cost while keeping all the essential information content.

After the desired accuracy is attained through training, the model is deployed on the target accelerator where the median filtering operations are performed more efficiently with dedicated hardware units.

8.6 Hardware Design for Efficient Error Detection and Rectification

Introduced fine-grained invariants can be checked and hardware errors can be rectified efficiently at inference time through dedicated hardware extensions in deep neural network accelerators. These extensions can be implemented at minimal hardware cost with basic hardware components such as comparators and multiplexers.

8.6.1 Hardware Design for Efficient Anomaly Detection and Suppression

We carry out anomaly detection and suppression operations entirely on hardware at inference time, which allows us to implement these operations with minimal hardware resources and with no consequent performance issues. First, we will look into the design of a hardware unit that can efficiently carry out the anomaly detection and feature suppression operations with minimal resources. We will then describe how these operation stages are positioned within the data flow of a typical DNN accelerator to deliver extensive coverage against a variety of bit-error

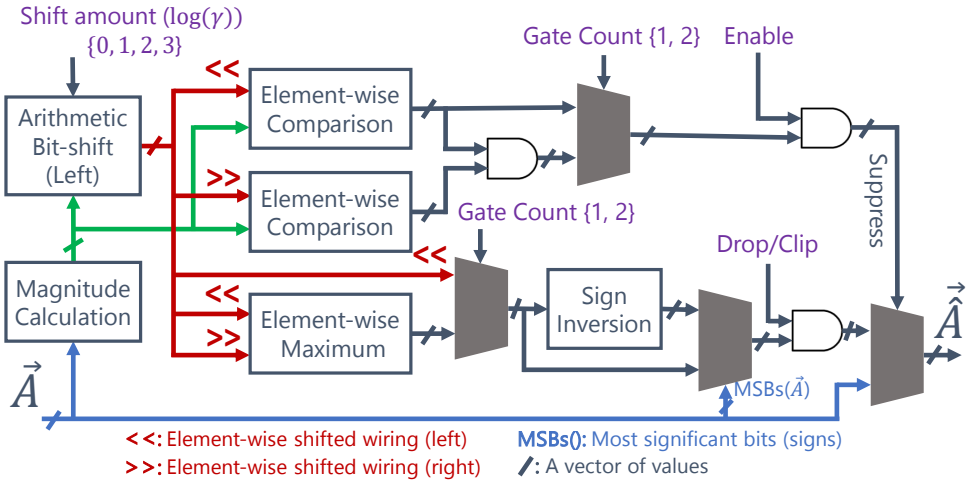

Figure 8.8. Hardware implementation of anomaly detection and suppression operations.

cases that occur in both the accelerator buffers (e.g., weight, activation) and the computational fabric (e.g., systolic array).

The required hardware operations involve magnitude calculation, multiplication with a scaling factor (γ), comparison, and a selection circuitry (multiplexers and AND gates) for dropping or clipping the desired features. Within the basic components in the list, multiplication operations particularly stand out in terms of expense; therefore, we reduce the need for explicit multiplication operations by limiting the scaling factor (γ) into powers of two, obtainable merely through arithmetic shifting. The vectorized hardware implementation is demonstrated in Figure 8.8. The hardware unit first calculates the enforced thresholds on the fly by finding the magnitude of each element, then scaling with γ through arithmetic shifting where the shift amount is controlled by an external signal. The simplicity of the required operations makes the threshold calculations extremely efficient. The calculated threshold vectors are aligned with the activation magnitude vectors through element-wise shifting and numerically compared to determine if the threshold is violated for each activation value. An external signal (*Gate Count*) determines whether the suppression signal for each activation is generated by considering the violation signals for either one or both of the thresholds calculated through the neighboring activation values. If the activation values need to be dropped, they can be replaced directly

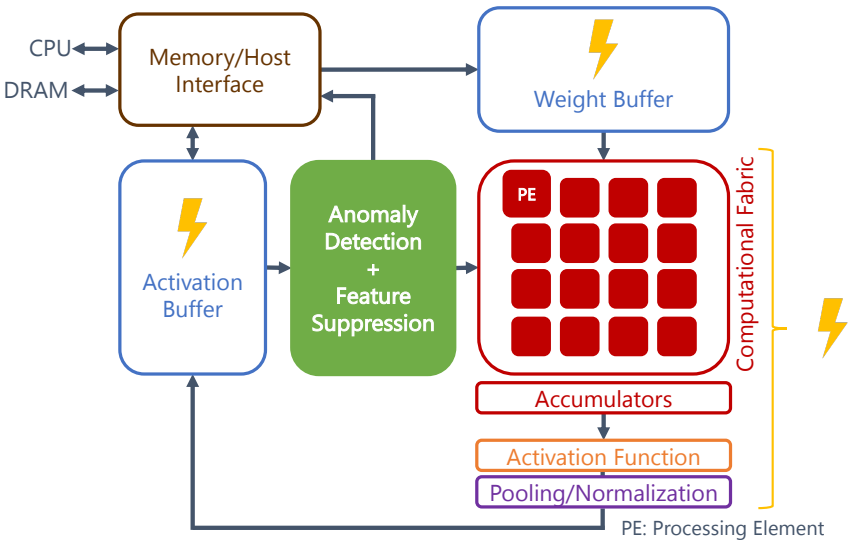

Figure 8.9. Anomaly detection and suppression unit integration into a DNN accelerator.

with zero. Clipping necessitates a slightly more complex maneuver in that it relies on the identification of the largest threshold value violated while preserving the sign of input activations in the suppressed values. Finally, the generated suppress signals determine if the original or suppressed values are reflected to the module output.

The proposed hardware module could be integrated into the outputs of the activation buffer, as in Figure 8.9. The demonstrated architecture[6] loads weights into the computational fabric (i.e., systolic array) through its weight buffer. The processing of each layer is performed by fetching the layer inputs from the activation buffer, performing multiply-accumulate operations, accumulating the results if necessary, and finally generating the layer output by passing the results through the activation pipeline. We observe that positioning the proposed stage just before processing each layer delivers advantages against errors occurring in various locations of the accelerator. First, any bit-error in the activation buffer will be checked and suppressed before getting involved in further computations. Second, any bit-error in the computational fabric or the activation pipeline will be stored in the activation buffer and then checked prior to processing the next layer. The output layer will be protected through the introduced small redundancy when the

[6]Architecture diagram is inspired from [35].

outputs are checked by the proposed hardware unit before transferring to the host device memory. Third, the proposed channel-wise checking scheme is useful, as an error in the weight buffer will impact a single neuron output or feature map channel, and it can be effectively checked and suppressed by the additional hardware before processing each layer.

8.6.2 Hardware Design for Efficient Median Feature Selection

The median feature selection could be carried out efficiently on the hardware at inference time with minimal effect on system performance. In this section, we first iteratively build efficient median selection hardware, describe how these components could be integrated into an embedded DNN accelerator, then finally suggest various optimizations to minimize the design costs. Although we utilize a systolic-array architecture for demonstration, the proposed hardware plug-in can be combined with any DNN accelerator in a modular fashion without involving intrusive design modifications.

We refer to the earlier work in sorting networks [214] to construct efficient median filter implementations that are demonstrated in detail in Figure 8.10. Sorting networks are made of one simple building block whose functionality is merely sorting two numbers. A *Sort-2* unit (Figure 8.10-a) can be constructed with a comparator and two multiplexer blocks.

As a next step, we construct a *Median-3* unit using previously designed *Sort-2* blocks. We first design a module that can fully sort 3-inputs by using three *Sort-2* blocks, then prune two multiplexer units whose outputs end up unused (dotted connections) to obtain an optimized design for *Median-3* calculation, as shown in Figure 8.10-b. A *Median-5* unit (Figure 8.10-c) could be constructed with four *Sort-2* blocks and a previously designed *Median-3* unit, then pruning the multiplexers whose outputs are unused. The methodology for median calculation relies on the fact that neither the largest nor the smallest number in a 4-element subset of the 5 numbers can be the median of these five numbers. *Median-5* units first eliminate these two extremal numbers with four comparisons, then search for the median among the remaining three elements.

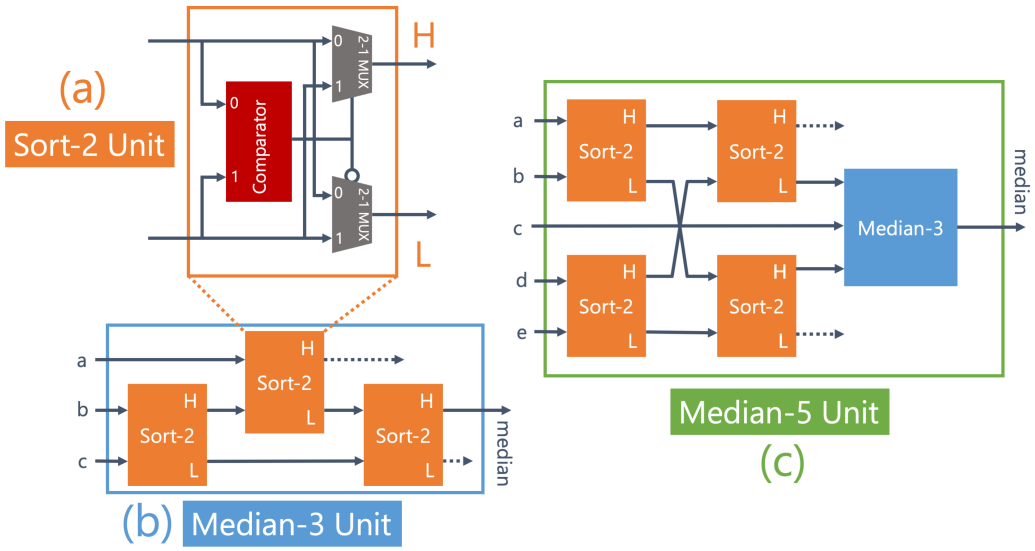

Figure 8.10. Sort-2, Median-3, and Median-5 hardware units.

The designed units could be integrated into a systolic array with ease. A systolic array is a two-dimensional grid of MAC (multiply-accumulate) units with each MAC unit receiving the partial sum from the previous MAC unit in the same column, multiplying the provided inputs, accumulating the result with the received sum, then forwarding the new partial sum to the next MAC unit. The entire architecture is implemented as a pipeline so that an $N \times N$ grid performs N^2 MAC operations at each cycle. The inputs of the systolic array are provided as a vector, and an input synchronization stage delays the input activations properly so that the partial sums are updated with the correct multiplication results. We can integrate an array of median units into the input connections (before the synchronization stage) to perform the designated median operations among the neighboring input values, as demonstrated in Figure 8.11. Each median window is processed in parallel by a distinct unit so that the filtering operation on the entire vector is performed within a single clock cycle, and integrated as a pipeline stage to preclude any performance bottlenecks. If the input is fetched partially due to bandwidth constraints in buffers, median operations could be performed on each portion without needing the entire vector.

Finally, the area and power overheads could be further reduced by sharing the result of overlapping computations. For instance, when two *Median-3* units operate on consecutive

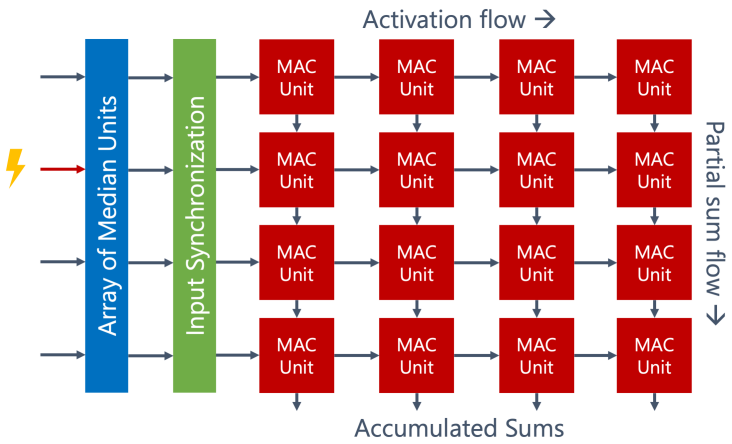

Figure 8.11. Median unit integration into systolic array architecture.

three-element groups of $\{I_0, I_1, I_2, I_3\}$, $\{I_1, I_2\}$ needs to be sorted in both $median(I_0, I_1, I_2)$ and $median(I_1, I_2, I_3)$ operations. As a result, the corresponding *Sort-2* unit can be overlapped. However, only one *Sort-2* unit can operate directly on the module inputs (i.e., a-b or b-c) in Figure 8.10-b, limiting the optimization to the boundaries of a *Median-3* pair and thus providing a limited reduction from 6 to 5 *Sort-2* units.

While the outlined techniques deliver significant implementation benefits, a *Median-3* unit design could be attained that seamlessly integrates to the systolic pipeline nature of DNN computations by providing a regularized sharing across overlapping windows. Each *Median-3* window after the initial one could be implemented with only two comparisons by sharing the overlapping comparison. The regularized sharing can be attained by decoupling the sequential dependency and simplifying the *Sort-2* units into the constituent comparators and multiplexers while inducing a regular iterative execution of the median-3 operation. The consequent design shown in Figure 8.12 reduces the number of comparators from 3 to 2 at each module by enabling comparator sharing between modules iteratively, reduces the number of multiplexers from 4 (remaining multiplexers after pruning) to 2, minimizes the module latency by eliminating the sequential dependency between the comparisons and lends itself to a regular, pipelined implementation. These pipelined, hardware sharing architectures can be generalized through induction to *Median-5* units that can be implemented with only 4 comparators (after the first

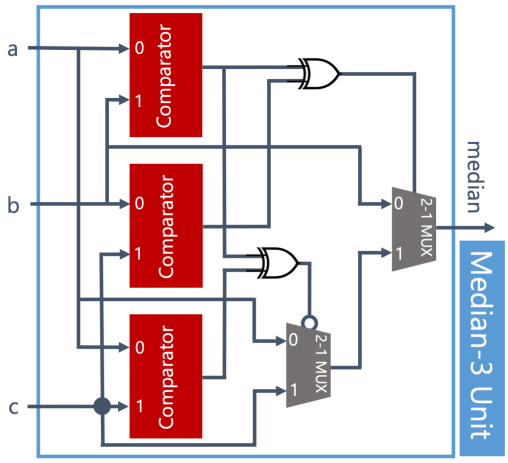

Figure 8.12. Optimized Median-3 unit.

Median-5 operation) instead of 7; a minimal multiplexer structure can route the median value to output by utilizing the ordering information.

8.7 Experimental Method

8.7.1 Anomaly Detection and Suppression Experiments

Anomaly Detection and Suppression Simulation in Software

We demonstrate the reliability improvements on three DNN models that are trained on three different datasets: LeNet-5 on the MNIST dataset [211], SqueezeNet [215] on the GTSRB dataset [50], and finally ResNet-18 [13] on the CIFAR-10 [207] dataset. The accuracies of the baseline models are 99.31%, 94.57%, and 88.63%, respectively. All models and experiments are implemented in PyTorch [213].

We design the proposed feature suppression rules as a custom DNN layer. The layer supports the definition of the number of gate activation variables. Namely, we utilize a single gate for each activation (Equation (8.1)) in LeNet-5 and two gates (Equation (8.2)) in SqueezeNet and ResNet-18 models. We explore the options of both feature dropping and feature clipping to suppress anomalous activations with various threshold coefficients (γ). An activation is immediately dropped upon a violation in the first option. The latter option allows the magnitude

of the activation to be clipped to the violated threshold. When two gate variables are utilized, the clipped magnitude is adjusted to the largest threshold violated. The models trained with the suppression rules exhibit comparable accuracy values compared to baseline models, where the test accuracy values are within $0.2-0.5\%$, $0.3-0.4\%$, and $1.3-2.4\%$ for LeNet-5, ResNet-18, and the highly irredundant SqueezeNet model, respectively. The LeNet-5 model can be trained directly with the integrated suppression rules in 100 training epochs (identical to the baseline model). The SqueezeNet and ResNet-18 models are trained in a two-stage process. We first train half-sized models with the same number of iterations required for the baseline models (100 epochs for SqueezeNet, 200 epochs for ResNet-18). We perform layer modifications, integrate the suppression rules, and perform further fine-tuning (100 epochs for SqueezeNet, 50 epochs for ResNet-18) in the second stage. The highest reported accuracy values are attained by utilizing the step function in the forward pass of the suppression layer and performing the backward pass with the sigmoid approximation. Utilizing the step function or the straight-through estimator in the backward pass often leads to instability in the training process. The straight-through estimator is oblivious to the forward pass behavior of the suppression layer, while the step function does not provide an efficient gradient flow.

Error Injection Method and Bit-Error Model

Error injection is commonly utilized in both academia and industry to verify the safety claims of safety-critical designs. Error injection experiments need to be repeated exhaustively to reach statistically significant conclusions. However, full hardware simulation of a system is computationally expensive, and runtime constraints often limit the applicability of exhaustive error injection experiments when practical designs are considered. We tackle this problem by first generating hardware-specific bit-error models, expressing them on the abstraction of the DNN graph, and performing exhaustive simulations through our error injection framework in PyTorch, at the speed of DNN inference. The data-flow nature of DNNs reduces the potential mismatch between hardware errors and bit-error model, thus boosting confidence in the analysis.

Our simulation environment can inject errors into DNN weights and activations. The data types are assumed to be quantized to the 2's complement fixed-point format. We consider both 8-bit (LeNet-5) and 16-bit (SqueezeNet, ResNet-18) data types for various DNN models. The data range for each network is profiled to determine the required integer and fraction bits. After the desired error rate is provided, the framework randomly determines the erroneous variables, impacted bit positions, and performs inference to measure the accuracy with the applied perturbations. The injection is performed before the prediction for the weight errors, and during the prediction for the activation errors. The experiments are repeated at the desired error rate until we obtain statistically consistent results.

Hardware Integration and Overhead Measurements

We design the hardware modules in Verilog HDL, integrate them into the DNNWeaver v2.0 [37] open-source DNN accelerator design, and perform hardware synthesis with the Synopsys Design Compiler to measure the area, power, and timing overheads. On-chip buffers (i.e., input, weight, bias) are not included in the synthesized design. We utilize the Silvaco Open-Cell (15nm) and the Synopsys DesignWare libraries in the synthesis process. The target frequency is chosen as 1 GHz for both 8-bit and 16-bit designs.

8.7.2 Median Feature Selection Experiments

Median Feature Selection Simulation in Software

We first implement the proposed median feature selection method as a custom DNN layer in PyTorch [213] for use in the training and error injection experiments. Median feature selection layers are parameterized by the median window size (w), the stride size (s), and the systolic array input size (N) to simulate the behavior of the outlined hardware filters. The systolic array size is important for modeling the hardware behavior precisely because if the layer input size exceeds the systolic array input size, the systolic array processes the input in smaller chunks; thus, each chunk is individually zero-padded and filtered by the proposed hardware architecture.

A custom convolution layer implementation enables us to precisely control the flattening order of the input feature maps and the filter weights. We first flatten the input feature map patches and convolution filters, then apply median feature selection, and finally process the convolution layer by simulating it in the form of matrix multiplication. While we could perform flattening in the various enumerations of the tensor dimensions, placing the same pixel locations from neighboring channels into adjacent locations is essential for median feature selection, as previously discussed in Section 8.4.2.

In addition, the off-the-shelf median function implementation in PyTorch creates a significant performance bottleneck in the training and error injection experiments as it is not optimized for fixed window sizes. We implement our fixed median-3 and median-5 routines using minimum and maximum functions in PyTorch and speed up these operations more than $10\times$ compared to the standard median implementation.

Error Injection Method and Bit-Error Model

The data-flow nature of the DNN computations and the considered error types allow us to model the DNN error effects accurately through graph-level error injection. Conventionally, a significant concern in high-level error injection techniques has been the accuracy of the analysis [216, 217]. First, we are concerned with the transient errors in DNN variables (e.g., weights, activations). Modern DNN frameworks provide complete access to weight and intermediate activation tensors; thus, the potential error locations could be accurately sensitized. Second, the data-flow nature of the computations guarantees that the manifestation and propagation of variable errors could be modeled precisely with a mathematical formulation on the DNN graph. For instance, any error in the data buffers will manifest itself as a numerical perturbation in the variables. Similarly, a timing error in a MAC unit could be modeled as a numerical perturbation in the accumulated sums. These errors could be propagated to DNN output by performing inference with the introduced perturbations.

We focus on data-path errors solely in our analysis for two main reasons. First, voltage

scaling techniques usually apply to the data path portion of the DNN accelerator (SRAM (static random-access memory) weight/activation buffers, MAC units in the systolic array); therefore, the bit errors that are caused by such techniques manifest and stay confined to the data path. Second, while the SEUs (single-event upsets) may impact both the data path and the control signals in a safety-critical design, the data-flow oriented nature of the DNN accelerators results in only a minimal amount of control circuitry. For instance, Google's TPU [35] allocates only 2% of the die area for control and essentially all the rest for the data path. As a result, protection of the control circuitry through traditional techniques such as duplication could be palatable even if the associated techniques may prove rather costly.

We utilize three different neural network models that are trained on three distinct datasets for the error injection experiments: LeNet-5 on the MNIST dataset [211], SqueezeNet [215] on the GTSRB dataset [50], and ResNet-18 [13] on the CIFAR-10 dataset [207]. We investigate the impact of bit-errors on both network weights and activations. The weight errors are injected statically before the neural network inference, while activation errors are injected dynamically during the inference by the error injection layers in the target DNN models.

The bit-errors that are injected into DNN weights and activations accurately cover a wide range of error effects that can occur in the memory, storage buffers, and other sequential elements in the systolic array. First, the described bit-error scenario could occur due to single-event upsets (i.e., caused by high-energy particles) in the buffer SRAM cells or systolic array flip-flops. Second, a marginally low selection of the supply voltage could noticeably increase the bit-failure rate in the buffer SRAM cells despite its advantage in power consumption reduction [171]. Finally, voltage under-scaling [78] or over-clocking [173] could deliver energy/performance benefits in the systolic array, yet result in an increased rate of timing violations. Our simulation environment could be utilized to investigate all of these scenarios with the main difference of the error rates in the latter two scenarios being significantly higher than the first as a side effect of aggressive hardware optimizations in the embedded DNN accelerators.

DNN accelerators utilize low-precision fixed-point data types for inference; therefore,

we model bit-errors on the 2's complement fixed-point data type while estimating the numerical impact. The desired bit-width of the data types is provided as an external parameter to the simulation, and the allocation of integer/fraction bits is performed according to the dynamic range of the weights/activations. Specifically, we consider 16-bit data types for both weights and activations (with 11 and 6 fraction bits, respectively) in the conducted error injection experiments.

Hardware Integration and Overhead Measurements

We design the median feature selection hardware in Verilog HDL and integrate it into DnnWeaver v2.0 [37], which is a representative open-source design for an embedded DNN accelerator. Hardware synthesis is performed with the Synopsys Design Compiler to characterize the area and power overheads, and identify potential timing bottlenecks. We exclude the large buffers (input, output, weight, bias) from the design before synthesis and utilize the Silvaco Open-Cell (15nm) and the Synopsys DesignWare libraries in the synthesis process. The target frequency is chosen as 200MHz. We report the area and power share of the median feature selection hardware in 8-bit and 16-bit accelerator designs for overhead characterization.

8.8 Experimental Results

This section first demonstrates the error resilience improvements and compare the results to other error tolerance methods. Second, we characterize the area and power footprint of the hardware module that carries out the required operations when integrated into a DNN accelerator.

8.8.1 Error Resilience Improvements

Anomaly Detection and Suppression Experiments

Figure 8.13 demonstrates the accuracy of three distinct DNN models separately under the weight and activation errors. We consider various versions of each model that are equipped with different error tolerance techniques. *Baseline* represents the bit-error resilience of a model with no additional safety features. *TMR* is the experimentally measured bit-error resilience of

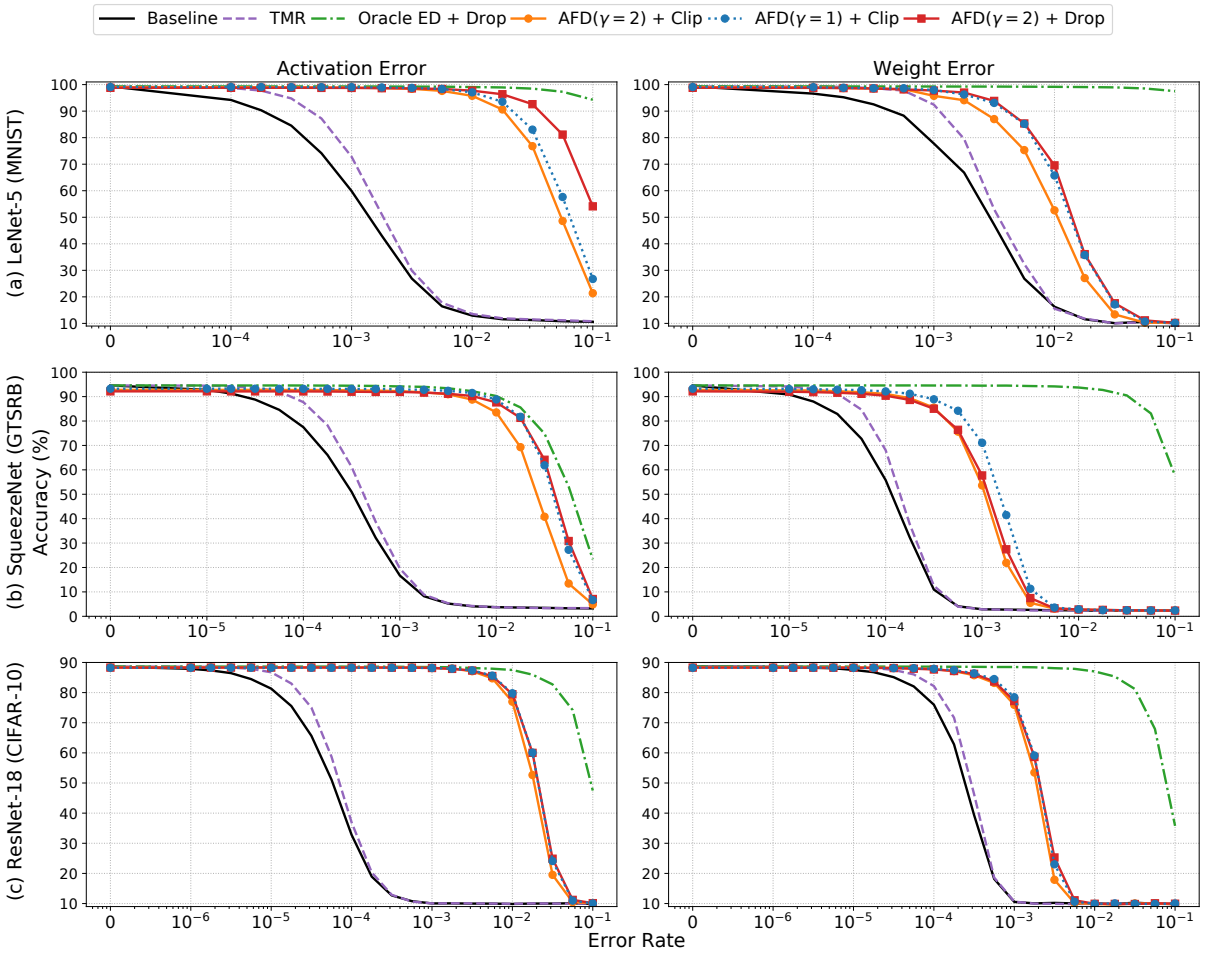

Figure 8.13. Resilience improvements delivered by anomaly detection and suppression.

a coarse-grained TMR (triple modular redundancy) scheme where three instances of the same network are simultaneously executed, and the decision is made through majority voting. *Oracle ED + Drop* signifies the result for the case where the erroneous weight and activations could be precisely located through an oracle error detection technique and dropped by setting to zero. The remaining methods utilize our *AFD* (anomalous feature detection) technique with the given threshold coefficient (γ) and tackle the errors either by dropping or clipping to the enforced threshold. In Figure 8.13, the relative criticalities of the weight and activation errors vary among the models and depend on the numerical range allocated to each type in the quantization schemes. The graphs are expected to show a higher accuracy drop if the incidence of errors were boosted through the simultaneous presence of both weight and activation errors.

The LeNet-5 model can tolerate up to 1000× more errors with the same accuracy loss if all erroneous activations could be precisely located and dropped. Although precise detection of errors requires additional information redundancy, the *AFD* technique provides error detection for free, yet the comparable precision of our technique enables the model to tolerate up to 300× more errors when it is coupled with suppression through feature dropping. The precision of *AFD* is similar to oracle error detection in particular networks, especially at lower error rates. *AFD* delivers resilience at up to 500× higher activation error rates with the same accuracy loss compared to the baseline SqueezeNet model when combined with feature dropping. The error rate where appreciable accuracy loss (e.g., ∼1% loss) commences could be delayed to almost 10000× larger error rates in ResNet-18 if the erroneous activations are to be precisely located and dropped. The majority of errors could still be localized by *AFD* at no additional cost and dropped; thus, the same accuracy drop point could be delayed to $\sim 2000\times$ higher error rates compared to the base model. The superiority of feature dropping over feature clipping in terms of fault resilience for the same threshold coefficient, γ, values is moderated when feature clipping utilizes a lower threshold coefficient; feature clipping with ($\gamma = 1$) delivers comparable results to feature dropping (with $\gamma = 2$), while retaining comparable baseline accuracy.

In addition, the impact of weight errors could be significantly inhibited without requiring precise error correction if the erroneous weights could be located with high accuracy, and set to zero before the execution. This approach is observed to provide resilience at up to $\sim 1000-3000\times$ more errors with the same accuracy loss when compared to baseline in Figure 8.13. Our technique does not directly operate on the weights but instead monitors and suppresses the activations, yet delivers particular benefits against weight errors, mainly when applied in a channel-wise manner as described in the previous sections. The anomalous feature detection combined with suppression allows the network to operate with the same accuracy loss at 20× larger error rates in the weights than the baseline model. The benefits could be improved further by operating directly on the weights if the anomalous weights are monitored and suppressed in the buffers or before loading into the computational fabric.

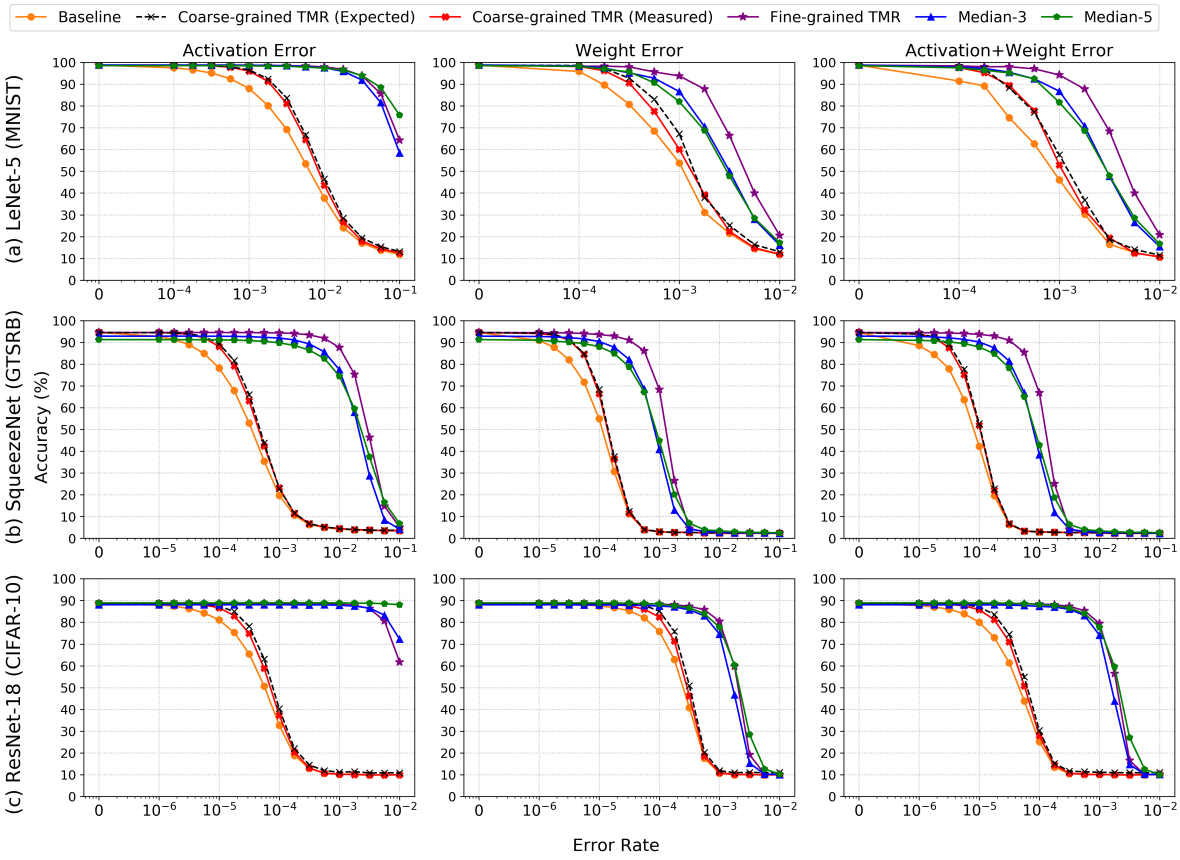

Figure 8.14. Resilience improvements delivered by median feature selection.

Median Feature Selection Experiments

Figure 8.14 demonstrates the accuracy of various neural network models that are subject to weight and activation errors under various error rate scenarios. We include a wide span of error rates to observe the entire picture of the accuracy drop for the investigated models. Even though certain error types such as SEUs caused by high-energy particles might occur rarely, aggressive optimizations such as voltage scaling could result in relatively high error rates (e.g., 10^{-2}) [78, 103] when the design is aggressively scaled to reduce power consumption. *Baseline* results indicate the neural network response against errors with no safety mechanism thus devoting all allocated resources to accuracy boosting. For comparison, we have designed a coarse-grained TMR system where three instances of the network are executed concurrently (with exclusive resources and parameters), and the final decision is made through majority voting. As we

perform error injection on the TMR system and report the accuracy results (*Coarse-grained TMR - Measured*), we further mathematically estimate the TMR system accuracy (*Coarse-grained TMR - Expected*) from *Baseline* accuracy values through Equation (8.6). In Equation (8.6), a_{exp} and a_{base} refer to *TMR-Expected* and *Baseline* accuracies. a_{ef} and n_c signify the error-free accuracy of the baseline model and the number of output classes, respectively. The normalization with a_{ef} allows us to convert the accuracy values into probabilities that can be used in the reliability calculations. The three summed terms in Equation (8.6) refer to the conditions of all three networks being correct, only one network mispredicting due to bit-error, and finally, only the first network being correct with the other networks predicting different wrong decisions. The last term is due to the fact that we accept the decision of the first network when all networks disagree due to errors. The $(n_c-2)/(n_c-1)$ term allows us to exclude the case where the second and third networks end up with the same wrong prediction.

$$\frac{a_{exp}}{a_{ef}} = (\frac{a_{base}}{a_{ef}})^3 + 3(\frac{a_{base}}{a_{ef}})^2(1-\frac{a_{base}}{a_{ef}}) + (\frac{a_{base}}{a_{ef}})(1-\frac{a_{base}}{a_{ef}})^2(\frac{n_c-2}{n_c-1}) \tag{8.6}$$

In addition, we construct an additional fine-grained modular redundancy technique for comparison (*Fine-grained TMR*) where each neuron and convolution filter is triplicated at every layer, and one majority-voted output from each triplet is propagated to the next layer. The voting is performed for each pixel location individually in the convolutional layers. Since each layer incorporates three copies of the same neuron/filter, the memory requirements and the computational cost of the DNN model are triplicated in the fine-grained TMR technique. An alternative approach might start with a down-scaled network (i.e., $1/3$ of baseline model at each layer) to mitigate the overheads when a fine-grained TMR is constructed; yet down-scaled models do not usually display a competitive accuracy when compared to the baseline designs. For example, the test accuracy of the baseline SqueezeNet model drops from 94.52% to 82.91% when each layer size is scaled by $1/3$, making such an approach an unviable proposition. Finally, we report the results for our proposed technique for two distinct median filter sizes (*Median-3* and

Median-5) to demonstrate the effectiveness of median feature selection against both weight and activation errors. Both *Median-3* and *Median-5* feature selection utilize single step strides and thus require no additional redundancy in the DNN layers while providing extensive resilience.

Figure 8.14 demonstrates that resilience tends to vary among DNN models considerably. For instance, while LeNet-5 can tolerate activation error rates of up to 10^{-4} with negligible accuracy drop (0.2%), the SqueezeNet model experiences a noticeable reduction (16.3%), and ResNet-18 is severely impacted with almost 55.9% accuracy drop at this error rate.

Moreover, coarse-grained modular redundancy is helpful only at low error rates, and even then, only as long as its tremendous overheads are deemed palatable since the coarse-grained TMR system accuracy drops hand-in-hand with the baseline accuracy at high error rates.

It could be seen that median feature selection is extremely effective against activation errors. Accuracy drop starts early when the error rate hovers around 10^{-4} for the baseline LeNet-5 model, yet the *Median-3* and *Median-5* models experience the same levels of accuracy loss quite late and not before the error rate exceeding 10^{-2} (resilient to $100\times$ more errors). The improvement is more dramatic for the ResNet-18 model, where the baseline model starts to lose accuracy at error rates as low as 10^{-6}, whereas the *Median-3* and *Median-5* models delay the onset of accuracy loss to the much higher error rates of 10^{-3} and 10^{-2}, offering resilience against $1000\times$ and $10000\times$ larger number of errors, respectively. The accuracy difference between the median and baseline ResNet-18 is maximized at certain error rates (e.g., 10^{-3}) when median models suffer no accuracy loss, but baseline model decisions are reduced largely to randomness.

Median feature selection further provides a significant resilience improvement against neuron/filter weight errors. The proposed channel-wise median feature selection technique allows us to filter out the output of significantly impacted neurons/filters due to weight errors, thus improving reliability. While the SqueezeNet *Baseline* starts to drop accuracy at weight error rates starting at 10^{-5}, median models exhibit the same accuracy drop when almost $10\times$ more errors are present (an error rate of 10^{-4}). For ResNet-18, the *Median-5* model exhibits the same small accuracy drop ($\sim 1\%$) as the *Baseline* but not until weight error rates are $20\times$ larger.

The *Median-3* and *Median-5* models usually deliver similar performance while the results for *Median-5* are frequently better at the higher error rates. We also inject activation and weight errors simultaneously at the same error rate. Although the combined effect leads to a more rapid accuracy drop, the results may look similar to either weight or activation error graphs depending on the relative criticality of the error types.

Fine-grained TMR offers additional advantages over the *Median-3* and *Median-5* models since the variable replication guarantees that a single error within a TMR window will be corrected precisely. However, it introduces the sizable cost of triplication in the number of parameters and multiply-accumulate operations. For the activation errors, the reliability improvements offered by the median models and the fine-grained TMR technique are observed to be comparable, with the fine-grained TMR bested even sometimes due to the large window sizes of *Median-5*. The advantages of fine-grained TMR are more pronounced for weight errors; however, the utility of these improvements is arguably still marginal, mainly when the additional overheads are considered. On the other hand, *Median-3* and *Median-5* deliver bit-error resilience similar to fine-grained TMR across the given models while incurring no additional redundancy.

8.8.2 Hardware Overhead Characterization

Anomaly Detection and Suppression Experiments

Table 8.1 demonstrates the area and power footprints of the entire accelerator together with the share of the proposed hardware for 8-bit and 16-bit designs. The area footprint of the proposed hardware module is around $0.2-0.3\%$ of the entire design, a minuscule fraction of the chip area budget. Likewise, the power overheads are bounded to within $0.1-0.15\%$ of the accelerator's power consumption, paling in comparison to process variation driven chip power deviations. The operations could be implemented as a pipeline stage, which seamlessly fits into the design without incurring any throughput loss, but only an extra cycle in latency. The synthesis experiments validate that the combinational delay of the proposed module fits into a single clock cycle at 1 GHz frequency and does not create a timing bottleneck.

Table 8.1. Hardware area and power footprint for anomaly detection and suppression.

8-bit Design

	Accelerator	AFD + Suppression	%
Area (mm^2)	1.744	0.004	0.23
Power (mW)	955.5	0.9	0.09

16-bit Design

	Accelerator	AFD + Suppression	%
Area (mm^2)	2.907	0.009	0.31
Power (mW)	1160.0	1.7	0.15

Median Feature Selection Experiments

We report the synthesis results for both 8-bit and 16-bit accelerator designs with an array of integrated *Median-3* and *Median-5* unit options. Figure 8.15 demonstrates the area and power consumption shares of the median unit array, systolic array, and the rest of the accelerator (i.e., control circuitry, SIMD (single instruction, multiple data) core). The area and power consumption costs of the *Median-3* array constitute merely a $0.19-0.23\%$ and $0.07-0.13\%$ of the entire design, respectively. The cost of the *Median-5* array is similarly minuscule, ranging between $0.39-0.48\%$ for area and $0.13-0.19\%$ for power consumption. The reader will note that our estimation is somewhat pessimistic as the synthesis results do not consider on-chip buffers, which consume a significant portion of the chip area and power budget. As a result, the presented overheads are bound to wane further when the buffers are included in the analysis.

A coarse-grained TMR system requires the systolic array to be replicated with up to $64-108\%$ area and $30-42\%$ power overheads in the presented designs. If the bit-errors are a concern in the buffers, additional safety measures are necessary, such as parameter replication or ECC (error correction codes) presented in [72]. While the replication of the buffers is costly, a more cost-effective solution, i.e., ECC, approximately requires 50% and 30% expansion in the buffers for 8-bit and 16-bit data types, respectively. When the overheads of the standard safety measures are taken into account, the insignificant cost of median feature selection turns it almost

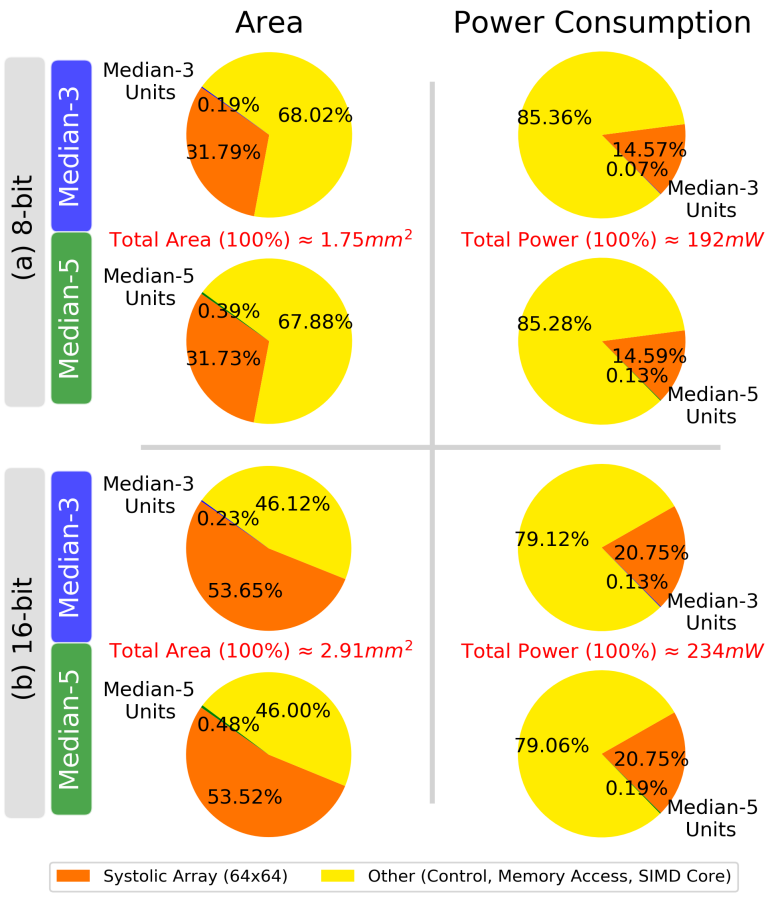

Figure 8.15. Area and power consumption of median feature selection.

into a free lunch for delivering safety in resource-constrained embedded DNN accelerators.

Finally, the performance overhead of the median feature selection is minimal. We opted to allocate a single cycle and implemented it as a fully integrated pipeline stage with the optimal hardware units presented. This approach increases the operation latency by one cycle, yet the throughput of the system is not impacted. The batch latency of a systolic array is proportional to $2N+M$, where N is the systolic array dimension, and M is the batch size; therefore, an extra cycle corresponds to $1/(2N+M) \approx 0.74\%$ increase in latency when $N = 64$ and $M = 8$. Alternatively, the minimal delay of the optimized units may allow combining them with the existing logic without incurring an extra cycle, if there is slack in the timing.

8.9 Discussion

Overall, the two outlined novel mechanisms of error localization and suppression can be coupled to deliver highly resilient neural network processing systems. Potential error locations are pinpointed through computational invariants injected through the learning process, and anomalous variables are snapped back outright before they have had a chance to propagate and influence the neural network decisions. In contrast to the outright effect of the large deviation bit errors encountered, the impact of suppression, which accords with the inherent distribution of neural network variables, on model accuracy is highly muted. The described approach is a potent strategy to engender approximate error resilience methods, revolutionizing our perspectives on functional safety for deep learning hardware. As a result, strict safety goals can be attained at minimal additional cost in DNN applications.

The novel techniques we outline focus primarily on the data path and buffers where the majority of hardware resources are allocated, and the cost of delivering functional safety through conventional fault tolerance methods proves to be exceedingly high. While control path integrity is just as important, the inordinate cost of traditional techniques can be easily borne for the small footprint of control circuitry in DNN accelerators that may necessitate absolute resilience.

We attain strong error resilience and competitive accuracy through the inherent flexibility and redundancy of neural networks. Neural networks embed redundancy in various dimensions, and the redundancy types that cannot be effectively squeezed through model compression can be utilized for boosting error resilience at no additional cost. Our preliminary investigation indicates that model compression methods such as pruning can be applied to the proposed models without impacting their outstanding error resilience characteristics. On the other hand, introduced invariants can lead to dependencies across neighboring variables, which need to be taken into consideration during the model compression process.

While the outlined analysis focuses on convolutional and fully connected layers, the proposed techniques are expected to generalize to a wide range of DNNs, such as recurrent

models and other emerging neural network architectures.

Overall, the computational characteristics of neural networks can enable significant breakthroughs for the error resilience problem in deep learning hardware, delivering highly effective solutions at imperceptible overheads.

8.10 Chapter Summary

The error resilience of deep neural networks can be boosted noticeably by restricting and containing the numerical contribution of the errors without necessitating explicit error correction steps. The proposed novel error detection and remediation techniques can complement each other seamlessly to tackle errors with high precision while neither incurring additional information redundancy nor having a noticeable impact on the error-free classification accuracy. The proposed approach innovatively redefines the error resilience problem in the context of deep neural networks thus unlocking effective opportunities for efficiently embedding functional safety into the next generation of machine intelligence hardware.

8.11 Acknowledgements

Chapter 8 contains re-organized reprints of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Shaping Resilient AI Hardware Through DNN Computational Feature Exploitation," *IEEE Design & Test (D&T)*, vol. 40, no. 2, pp. 59–66, 2023 ([4]), Elbruz Ozen and Alex Orailoglu, "Boosting Bit-Error Resilience of DNN Accelerators Through Median Feature Selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3250–3262, 2020 ([5]), and Elbruz Ozen and Alex Orailoglu, "Just Say Zero: Containing Critical Bit-Error Propagation in Deep Neural Networks with Anomalous Feature Suppression," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, IEEE/ACM, 2020 ([6]). The dissertation author was the primary investigator and author of all three papers.

Chapter 9

Designing Error-Resilient Deep Neural Networks

The inherent resilience characteristics of deep neural networks against small numerical perturbations facilitate proactive avenues for improving the safety and reliability of embedded deep learning applications. This chapter demonstrates the possibility of such exploitation by juxtaposing the reduction of the vulnerability surface through the proper design of the quantization schemes with shaping the parameter distributions at each layer through the guidance offered by appropriate training methods, thus delivering deep neural networks of high resilience essentially through algorithmic modifications. Unequaled error resilience characteristics can be thus injected into safety-critical deep learning applications in a proactive manner to tolerate bit error rates of up to 10% at absolutely zero hardware, energy, and performance costs while improving the error-free model accuracy even further.

9.1 Introduction

The outlined resilience characteristics of deep neural networks in Section 5.2.1 promise novel opportunities for tackling the functional safety problem in embedded deep learning applications at no additional cost. As long as it does not lead to a misclassification at the neural network outputs, an error in the neural network variables can be classified as non-critical. Furthermore, neural networks are known to endure mild and bounded inaccuracies gracefully,

including numerical errors introduced in the quantization process. Similarly, limited noise effects are well tolerated despite the fact that nearly all neural network variables are impacted to a certain degree by such perturbation patterns [186].

Unfortunately, the described resilience characteristics of neural networks fall short of conclusively establishing the safety of deep learning applications. Previous studies [66, 68, 70] have repeatedly demonstrated that models often start to lose accuracy even at low bit error rates (i.e., 10^{-6}). The impact of hardware errors is bound to be substantial if a numerically significant bit position is impacted. Moreover, if the numerical range of variables in the executed model does not fully utilize the allocated numerical range in the employed number representation scheme, the numerical effect of the potential bit errors can easily exceed the expected scale of the original neural network variables by a few orders of magnitude even. As a result, even a small quantity of such large error patterns can substantially alter neural network outputs and thus critically impair system accuracy.

The large numerical range in the number representations constitutes a significant source of vulnerability against critical bit error effects in deep neural networks. First, we observe that the outlined vulnerability can be alleviated proactively through layer-wise quantization techniques by tightening the quantization margins to match the utilized range at each layer. Furthermore, we notice that the numerical distribution of model parameters at each layer often contains *outlier* values which can significantly exceed the rest of the distribution. The stretched quantization range imposed by such outlier variables exposes the neural network model to large magnitude errors, with deleterious consequences for model accuracy. We propose a novel regularization method, *outlier regularization*, in the training process to tighten the numerical range and cluster the parameter distributions, thus curbing the impact of potential bit errors more effectively. As a result of the combined employment of layer-wise quantization and outlier regularization techniques, deep neural networks can inherently tolerate extreme bit error rates (up to 10%) with no reliance on additional error tolerance mechanisms and at essentially no additional cost.

Experimental analysis reveals yet another surprising phenomenon in our studies, as the

models trained with the proposed outlier penalty terms consistently exhibit a noticeably higher accuracy than the baseline models in both full-precision and quantized formats due to the model regularization effect in training and improved quantization quality.

While our approach bears similarities to activation range restriction methods [99, 100], it differs in that we embed such restrictions indirectly through the layer-wise quantization process instead of employing explicit range limitations. As a result, the restrictions are implemented implicitly without necessitating any additional operations. Moreover, we perform range restrictions on the model parameters in a similar manner through layer-wise quantization, in addition to the activations. The effectiveness of parameter range reductions is shown to be quite substantial against model parameter (i.e., weight, bias) errors, which are the primary threat to system safety due to their permanent impact on model accuracy. The remarkable benefit of this approach is also validated through experimental comparison with [99]. Yet again, our approach differs in that we actively shape the parameter distributions through dedicated regularization terms, attenuate the scale of bit error effects, and boost the error resilience of the models remarkably further.

This manuscript starts off with an introductory overview of model quantization in Section 9.2. Section 9.3 discusses the design choices and our novel regularization technique to improve the error resilience of deep neural networks. A detailed experimental analysis is provided in Sections 9.4-9.5. Finally, we discuss the future implications and directions in Section 9.6 prior to presenting our concluding remarks in Section 9.7.

9.2 Overview of Model Quantization

Deep learning inference rarely requires the high-precision offered by the floating-point representations. Neural network variables can be represented in the fixed-point format and even quantized to smaller bit widths (i.e., 8-bit and often less) with small or even imperceptible impact on model accuracy. Quantized models offer a noticeable reduction in memory footprint and a significant decrease in energy consumption due to reduced data movement. Furthermore, smaller

fixed-point arithmetic units that are faster and more efficient than their floating-point counterparts [26] cut down the processing cost and boost system performance further. Quantization is a commonly employed technique in deep learning inference, including embedded neural network accelerators, because of the outlined benefits.

The quantization process maps a given continuous distribution into a discrete range through a linear projection of the variables and rounding the final result to the nearest quantization point. A quantization scheme is termed *symmetric* if the representation is symmetric around zero. Namely, a symmetric quantization scheme can be considered as a function which maps a floating-point range into a fixed-point range (Equation (9.1)) merely through scaling and rounding (Equation (9.2)):

$$[-m_{float}, m_{float}) \longrightarrow [-m_{fixed}, m_{fixed}) \tag{9.1}$$

$$x_{fixed} = round\left(\frac{m_{fixed}}{m_{float}} \times x_{float}\right) \tag{9.2}$$

With knowledge of the scale factor (m_{float}/m_{fixed}), the quantized variables can to a large extent be restored to their original form. A simplistic fixed-point representation may consist of a mantissa and utilize a single global scale factor. A scheme with a single global scale factor, while straightforward, may significantly exceed the minimal quantization error, failing to display a competitive model accuracy as deep neural network variables often span diverse dynamic ranges at each layer. The diversity of the dynamic ranges can be observed in an example shown in Figure 9.1 for the weight distributions of the ResNet-18 model [13] trained on the CIFAR10 dataset [207]. The global range may not be covered with a small *quantization step size*[7]; on the other hand, employing a larger quantization step size might incur noticeable information loss and degrade model accuracy.

[7] *Quantization step size* refers to the numerical distance between two adjacent quantization points. This distance is constant between two adjacent points in uniform quantization schemes.

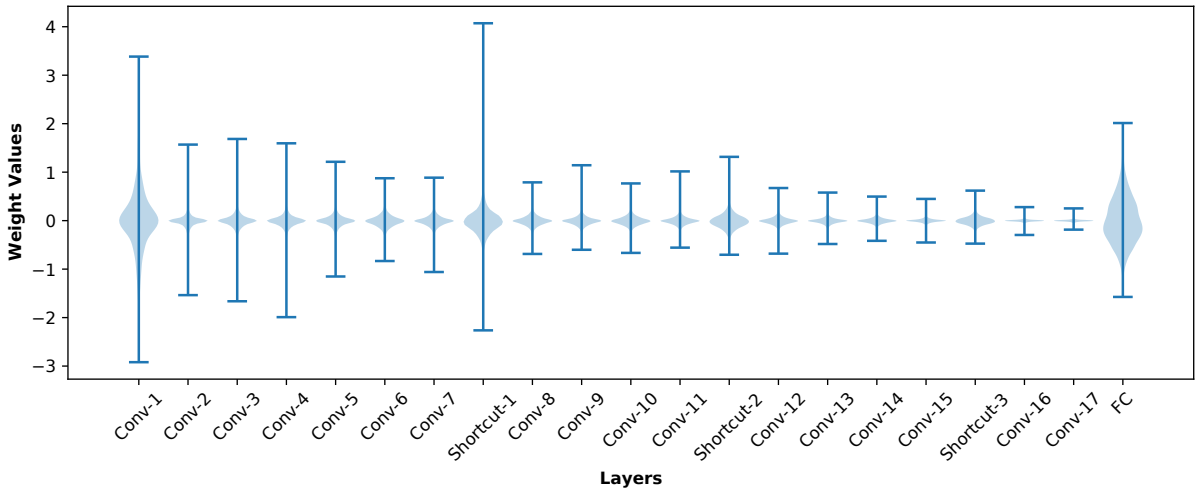

Figure 9.1. Weight distribution across neural network layers.

A quantization scheme with unique scale factors for a group of variables [218, 219], i.e., unique scale factors per layer, and distinct scale factors for weights and activations, can express values from quite diverse distributions accurately, improve quantization quality, and consequently allow model accuracy maintenance even at low bit-width representation.

Moreover, the weights and activations at each layer usually follow a distribution that embeds *outlier* values of significantly higher magnitude than the rest. An illustrative distribution pattern is shown in Figure 9.2 for the weights of the first convolutional layer in the same ResNet-18 model trained on the CIFAR10 dataset.

Large-magnitude variables may necessitate an increase in the quantization step size to accommodate the entire numerical range; otherwise, such values need to be clipped. Unfortunately, both increases in quantization step size and clipping large values can each lead to a larger quantization error and consequently reduce quantization quality. Solutions such as activation clipping have been proposed [32, 33] to partially address this problem in the context of activation quantization.[8]

[8]It should be noted that the usage of the term "activation clipping" in [32, 33] differs from the clipping concept that is introduced as an approximate error rectification technique in Chapter 8.

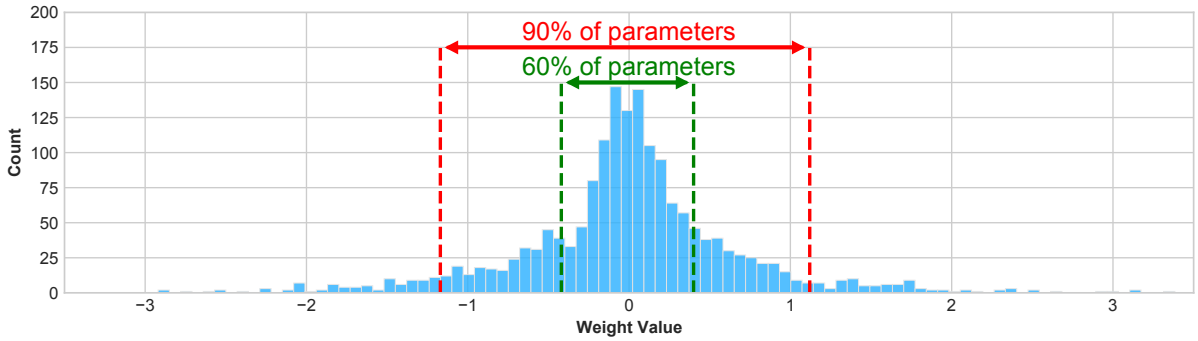

Figure 9.2. Sample weight distribution in a neural network layer.

9.3 Designing Error-Resilient Deep Neural Networks By Tightening Numerical Range

This section focuses on the represented numerical range in neural network layers and its significant impact on the error resilience of deep neural networks. For starters, we expound the dependence of the numerical range at each layer on the characteristics of the employed quantization methods and the dramatic improvements in system resilience that can be garnered through proper design choices in the quantization schemes. We proceed to propose a practical approach to shape the numerical distribution of the neural network parameters with proper regularization methods so that the numerical impact of bit errors becomes less pronounced in the quantized form of the obtained parameter distributions.

9.3.1 Tight Quantization Bounds with Layer-wise Quantization

Deep neural network layers span diverse numerical ranges at each layer; therefore, the accuracy of the quantization scheme can be improved through layer-wise quantization, which employs unique scale factors at each layer. Although this is a common practice for improving the precision of model quantization, the error resilience and reliability implications for layer-wise quantization have not been investigated in the literature. Our analysis demonstrates that the use of unique scale factors at each layer not only yields benefits in terms of quantized model accuracy but also boosts the bit error resilience of deep neural networks rather noticeably.

While the number of allocated hardware bits is fixed on the hardware for a variable, the same binary representation can span diverse numerical ranges at each layer in a layer-wise quantization scheme because of the scale factors. The proper range adjustment with the layer-wise scale factors can guarantee that the parameter distributions at each layer fully span the provided binary representation on the hardware (e.g., from -128 to 127 for an 8-bit binary representation). As a result, the numerical impact of the hardware bit errors is guaranteed to be within the layer distribution and not significantly larger than the layer parameters.

The impact of quantization range on model resilience can be explained with an illustrative example. The maximum weight magnitude in the last convolutional layer in Figure 9.1 is observed to hover around 0.25, and the global maximum weight value comes in at slightly above 4. If the last convolutional layer is represented with a quantization scheme that covers the entire global range, the numerical impact of bit errors becomes proportional to $(4/2^{b-1}) * (2^i)$ where b is the quantization bit-width, and i is the erroneous bit position. It can be seen that the error magnitude could easily exceed the expected distribution of the weights in the last convolutional layer, even for errors manifesting at the least significant bit positions. On the other hand, the numerical error magnitude can not exceed $(0.25/2^{b-1}) * (2^i)$ for the last convolutional layer if a layer-wise quantization scheme is applied. The error magnitudes would often be smaller than the original layer weights in the layer-wise quantization scheme, leading to such muted error effects being tolerated better by deep learning models, which are known to exhibit resilience against minor and limited-magnitude perturbations.

Alternatively, the effect of the layer-wise scaling factors can be considered as boosting the signal-to-error ratio at each layer by guiding layer variables to span the provided fixed-point representation range fully through the proper scaling factors. As a result, the numerical impact of bit errors is diminished in comparison to the value of the original neural network variables.

The described scheme can be applied to both model weights and activations with dedicated scale factors at each layer to improve the resilience of the model against the bit errors in the weights and the activations.

9.3.2 Squeezing Layer-wise Bounds with Outlier Regularization

Layer-wise quantization improves model resilience by precluding error effects that dwarf the actual variable distribution at each layer. However, we observe that the distribution of the variables at deep neural network layers frequently contains large magnitude variables, *outliers*, that increase the numerical range, thus exposing the model to error effects that can dwarf a large fraction of the distribution.

When the quantization step size is increased to accommodate such large magnitude variables, the ratio of a non-outlier variable magnitude compared to the quantization step greatly diminishes. We have previously concluded that the numerical impact of bit errors is proportional to the quantization step size ($2^i \times step\_size$) for impacted bit position i; therefore, the bit error effects can grow disproportionately when compared to the magnitude distribution of non-outlier variables in the corresponding layer.

The described problem can be alleviated by constricting the quantization range. A primitive solution could involve the setting of a smaller quantization range and the clipping of the large magnitude variables after training. However, this approach might result in an increase in the quantization error for the large magnitude variables and consequently impact model accuracy. An alternative approach might involve clipping the range during training to preclude the formation of a distribution with large magnitude variables; however, the optimal selection of the clipping threshold is rather difficult to establish when performed manually at each layer.

We utilize the flexibility of the training process to form clustered variable distributions at each layer and allow the target model to adjust the required numerical range at each layer automatically by discouraging significantly large magnitude parameters. Such modifications can be encouraged by designing additional goals for training and optimizing these goals in the learning process.

An additional regularization term in the standard cross-entropy loss function described in Equation (2.2) can penalize the outlier variables with large magnitude and shrink the numerical

range of the distribution effectively at each layer. L1 or L2 norm regularization can be utilized for this purpose, but such regularization schemes are not selective and penalize all weight values. Such an approach can fail to yield the expected benefits as the resultant scale changes may retain the large scale magnitude disparateness across the variable space. Instead, we introduce a novel regularization term, *outlier regularization*, which focuses the penalty term on the large magnitude parameters so as to penalize large magnitude variables while keeping the rest of the distribution intact.

We consider two different regularization terms to encourage the desired behavior in model training. The first alternative, *max-magnitude regularization*, computes the penalty term by accumulating the maximum magnitude values from each parameter group as in Equation (9.3). We treat the weight and bias values of a layer as separate groups in our implementation. In Equation (9.3), L denotes the total number of layers, and W_l and B_l refer to the weights and biases at layer l. The magnitude is computed element-wise, and the maximum function returns the maximum entry at each computed magnitude tensor.

$$\sum_{l=1}^{L} \left(max(|W_l|) + max(|B_l|) \right) \tag{9.3}$$

Max-squared regularization has a similar implementation except for the fact that we accumulate the maximum square values from each parameter group as in Equation (9.4) to construct the penalty term. The square function in Equation (9.4) is element-wise, similar to the magnitude function in Equation (9.3).

$$\sum_{l=1}^{L} \left(max(W_l^2) + max(B_l^2) \right) \tag{9.4}$$

We include the computed penalty term in the standard cross-entropy loss function in Equation (2.2). The overall loss function is demonstrated for maximum magnitude regularization in Equation (9.5):

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\ log(p_{i,c}) + \alpha \sum_{l=1}^{L} (max(|W_l|) + max(|B_l|)) \tag{9.5}$$

We further employ a regularization coefficient in the cost function (α) to exert precise control over the penalty term. The proper selection of the regularization coefficient empowers the deep learning model to penalize the largest magnitude parameter at each group sufficiently while incurring no adverse effect on the learning process so that competitive model accuracy can be attained. The parameter distributions are expected to show a stronger clustering as a result of regularization; when such parameter distributions are quantized with a proper scale factor to cover the squeezed bounds, the magnitude of the bit errors that can be suffered would diminish vis-à-vis the value of the quantized parameters.

The reader will note that the described regularization scheme targets the model parameters (i.e., weight, bias) in the given description. Model parameters are leaf nodes in the gradient back-propagation tree for the convolutional and fully connected layers; thus, the maximum magnitude entries can be penalized with the specified regularization methods with no impact on the other weights, even those that lie within the same layer. The described independence property across the weights underpins our technique as it enables the precise manipulation of the desired parameters with no unintended consequences in the parameter distributions.

The application of regularization methods to the activations, while feasible, does introduce far more complex interactions in model training. First, activations are derived through the multiply-accumulate operations that involve the layer weights; thus, the gradients accumulated on the activations eventually end up being back-propagated and used to update the weight entries. Such indirect effects on the model weights might introduce interference in training, reduce training efficacy in forming desired parameter distributions in the target model, and diminish system reliability against weight errors. Furthermore, penalizing a particular activation at a layer inevitably affects the other activations in the same layer in case of weight sharing across the units, particularly for the convolutional layers where the filter weights are shared to construct an entire

channel of the output feature map. As a result, the direct manipulation of the activations through regularization necessitates a more profound theoretical analysis. As the immediate vulnerability surface in neural networks mainly originates from the weight errors due to their repeated use in the inference operations and their consequent long-term impact on the model accuracy, our work puts particular emphasis on squeezing the error margins for model parameters.

9.4 Experimental Method

9.4.1 Experimental Setup

We implement the experimental setup in PyTorch [213]. The LeNet-5 model with the MNIST dataset [211], the VGG-16 [56] and the ResNet-18 [13] models with the CIFAR10 dataset [207], and the SqueezeNet model [215] with the GTSRB dataset [50] are used for the experiments. We utilize the following hyper-parameter configurations for training the target models: the LeNet-5 model is trained for 100 epochs with the Adam optimizer starting at a learning rate of 10^{-3}, and reducing the learning rate tenfold at epochs 33 and 66. The VGG-16 and the ResNet-18 models are trained for 200 epochs with the SGD (Stochastic Gradient Descent) optimizer (momentum=0.9) starting at a learning rate of 10^{-2} and reducing the learning rate tenfold at epoch 100. Finally, the SqueezeNet model is trained for 100 epochs with the SGD optimizer starting at a learning rate of 10^{-2}, and reducing the learning rate tenfold at epoch 50. The baseline and regularized models are trained with the same hyper-parameter configurations except for the usage of the regularization term. The value of proper regularization coefficients (α) depends on the target model and the regularization type. The utilized values are indicated in Table 9.1. The training outcome is not highly sensitive to minor changes in the regularization coefficients, and proper selection of these coefficients can be accomplished by starting with a relatively large value and reducing the value tenfold at every step until it is possible to train the model with a competitive accuracy. It should be noted that the utilized coefficient values for the proposed outlier regularization methods are noticeably larger than the

standard regularization terms (e.g., L1 or L2) in Table 9.1 since only outliers are penalized with the proposed regularization terms instead of all parameters.

We employ a 2's complement fixed-point representation and a symmetric quantization scheme in the experiments. We consider the widely-adopted 8-bit data types for the weights and activations in the context of the error injection experiments and the presented reliability analysis. More aggressive quantization schemes are employed in Section 9.5.4 to demonstrate the superior accuracy retention of the regularized models in the post-training quantization experiments.

We utilize two machines for model training and error injection experiments, each with 2× Intel Xeon E5-2630 v4 CPUs (central processing units), NVIDIA GeForce GTX 1080Ti GPU (graphics processing unit), and 16GB RAM.

9.4.2 Error Model

We design our in-house error injection framework to simulate the impact of bit errors on neural network variables. Bit errors are injected into weight and activation tensors prior to each convolutional and fully connected layer operation. Error injection is performed through a dedicated PyTorch function that receives the input tensor (weight or activation), the error probability, the data bit-width, and the maximum quantization magnitude as inputs. We indicate the portion of the erroneous bits in the weight/activation tensors with the term *bit error rate* to conform to the terminology used in previous studies [68]. The proposed framework is utilized to measure and report the classification accuracy at different bit error rates.

The bit error generation process is outlined in detail in Algorithm 9.1. The error injection function first generates the bit error mask through a Bernoulli distribution with the given error probability. A pseudo-quantization scheme allows us to determine the total numerical perturbation at each variable as a result of injected bit errors. Finally, the target tensor is perturbed by adding the generated numerical errors to simulate the bit error effects on the variables. The reader will note that the bit error rate in the perturbed tensors will approximately equal the provided bit error probability after error injection.

Algorithm 9.1: The methodology for error injection into DNN tensors

Input: Input tensor (I), error probability (p), data bit-width (b), maximum quantization magnitude (m)

Output: Output tensor (O)

1 Generate bit error locations through a Bernoulli distribution with probability p
2 Perform pseudo-quantization on the input tensor (I) by using b and m
3 Identify the original bit values in the quantized representation
4 Compute the numerical perturbation value for each bit error location
5 Compute overall numerical perturbations for each variable by summing numerical perturbations caused by individual bit errors
6 De-quantize overall numerical perturbation values that are computed for each variable
7 Add de-quantized perturbation values to the input tensor (I) to generate output tensor (O)

Our analysis and the resilience methods once again focus solely on the errors in the data path. We particularly focus on the bit errors on the parameters due to their permanent effect on the model behavior and the classification accuracy. Experimental analysis for the activation errors is also provided. Protection of the control circuitry can be delivered economically with conventional hardware fault-tolerance methods, as deep learning accelerators usually necessitate minimal control circuitry due to the inherent regularity of the micro-architecture and neural network computations. In addition, our analysis omits consideration of the errors in the quantization scale factors. As the adopted layer-wise quantization schemes incur a negligible number of scale factors when compared to the model parameters, i.e., at double the number of layers if distinct scale factors are utilized for the weights and activations at each layer, their resilience can be rendered by conventional hardware fault-tolerance methods at negligible overheads as well.

9.5 Experimental Results

We investigate the effectiveness of our approach by analyzing the impact of the proposed regularization terms through the measurement of full-precision accuracy values and observing the particularities of the parameter distributions at each layer. Moreover, bit error resilience characteristics are evaluated by measuring the classification accuracy at various bit error rates.

Table 9.1. The impact of regularization on the full-precision (non-quantized) model accuracy.

	Regularization Coeff. (α)	**Train Acc. (%)**	**Test Acc. (%)**
LeNet-5 Baseline	-	99.71	99.24
LeNet-5 Regularized (L1)	10^{-4}	99.86	99.27
LeNet-5 Regularized (L2)	10^{-4}	**100.00**	**99.41**
LeNet-5 Regularized (Max-Magnitude)	1	99.31	99.07
LeNet-5 Regularized (Max-Squared)	1	99.97	99.30
VGG-16 Baseline	-	99.71	91.55
VGG-16 Regularized (L1)	10^{-5}	99.65	**92.24**
VGG-16 Regularized (L2)	10^{-5}	99.80	91.93
VGG-16 Regularized (Max-Magnitude)	10^{-1}	99.76	92.21
VGG-16 Regularized (Max-Squared)	10^{-1}	**99.90**	91.89
ResNet-18 Baseline	-	99.37	92.22
ResNet-18 Regularized (L1)	10^{-5}	**99.86**	**93.57**
ResNet-18 Regularized (L2)	10^{-5}	99.82	92.85
ResNet-18 Regularized (Max-Magnitude)	10^{-1}	99.76	92.90
ResNet-18 Regularized (Max-Squared)	10^{-1}	**99.86**	92.97
SqueezeNet Baseline	-	**99.83**	94.36
SqueezeNet Regularized (L1)	5×10^{-4}	99.20	94.20
SqueezeNet Regularized (L2)	10^{-3}	**99.83**	93.94
SqueezeNet Regularized (Max-Magnitude)	5×10^{-1}	99.79	**94.47**
SqueezeNet Regularized (Max-Squared)	1	99.82	94.42

Finally, we perform post-training quantization experiments at lower bit widths to demonstrate the improvements in accuracy retention for the proposed regularization techniques as a result of the obtained parameter distribution characteristics. We compare the proposed regularization methods with the baseline case with no regularization and standard L1 and L2 regularization methods in the experiments.

9.5.1 Impact of Regularization Terms on Training and Full-Precision (Non-Quantized) Model Accuracy

The training efficacy and the model accuracy are essential considerations that need to be addressed for the proposed regularization techniques. Table 9.1 summarizes the regularization coefficients used for training (α) and the final accuracy values of the baseline and regularized

models. We observe that the target deep neural network models can be trained effectively with the outlier regularization techniques and outperform the baseline model accuracy values by a clear margin, particularly in the test/validation examples. We hypothesize that the described phenomenon in the test/validation examples occurs because of reduced over-fitting and improved generalization. Table 9.1 demonstrates that the obtained accuracy values through the outlier regularization methods are comparable and even sometimes superior to the standard L1/L2 weight regularizers that are widely used in practice to reduce over-fitting. The slight accuracy improvement in the training examples is another interesting phenomenon, necessitating a more comprehensive theoretical analysis for a more definitive answer.

9.5.2 Impact of Regularization Term on Parameter Distributions

The outlier regularization leads to a significant reduction in the dynamic range of the parameters at each layer and to parameter distributions that display a stronger clustering. Figure 9.3 and Figure 9.4 demonstrate the impact of the regularization term on the model parameter distributions at each layer for the LeNet-5 and ResNet-18 models. The colored portions of the violin plots demonstrate the distribution density at a particular weight value (i.e., wider regions indicate a higher density). The minimum and maximum values at each layer are indicated through the horizontal lines attached to both ends of the distribution curves.

The influence of the proposed regularization terms (i.e., *Max-Magnitude* and *Max-Squared*) on the parameter distributions is quite noticeable in both neural network models demonstrated in Figure 9.3 and Figure 9.4. We observe that the large magnitude outliers seen in the baseline models (*No Regularization*) disappear entirely as a result of the introduced regularization terms, as clearly exemplified in Figure 9.4. As a result, the dynamic range of parameters reduces significantly at each layer.[9] The boundaries of the clustered parameter distributions are learned automatically at each layer while attempting to squeeze the numerical range and minimize the cross-entropy loss at the same time. Furthermore, zero-centered and Gaussian-like

[9]Please note the scale difference in Figure 9.3 and Figure 9.4 for baseline and various regularized distributions.

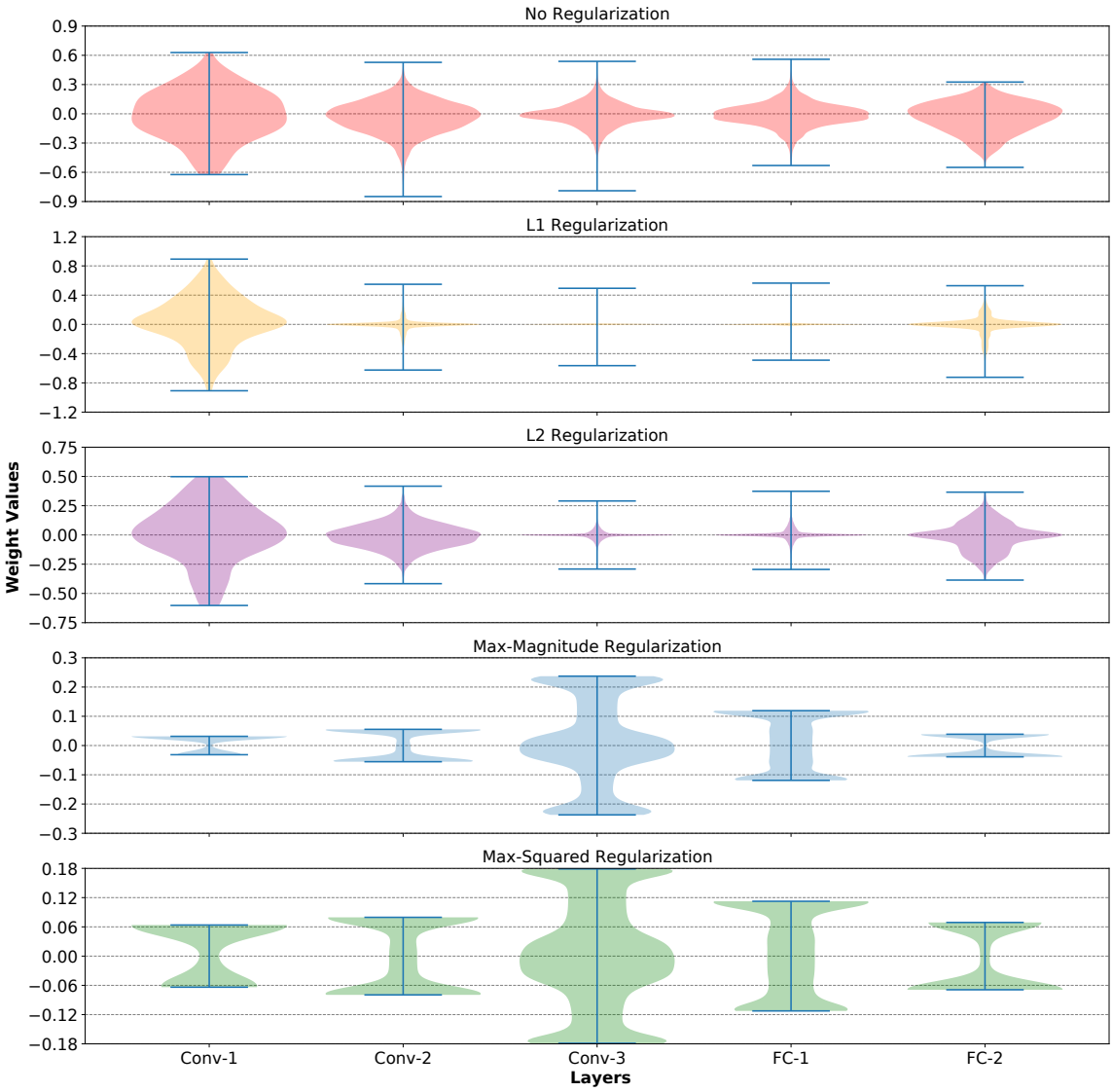

Figure 9.3. Regularization effect on the numerical range of the LeNet-5 parameters.

shapes of non-regularized layer parameter distributions transform into well-clustered uniform and even bimodal distributions that are concentrated around the limit values.

Our experimental results in Figure 9.3 and Figure 9.4 with the standard L1 and L2 regularizers indicate an interesting phenomenon. While L1 and L2 regularizers can reduce the numerical range, the obtained distributions are quite distinct from those obtained through the outlier regularization methods with the max-magnitude or max-squared terms. L1 and L2 terms penalize all parameters instead of only outliers; thus, the large magnitude discrepancies between

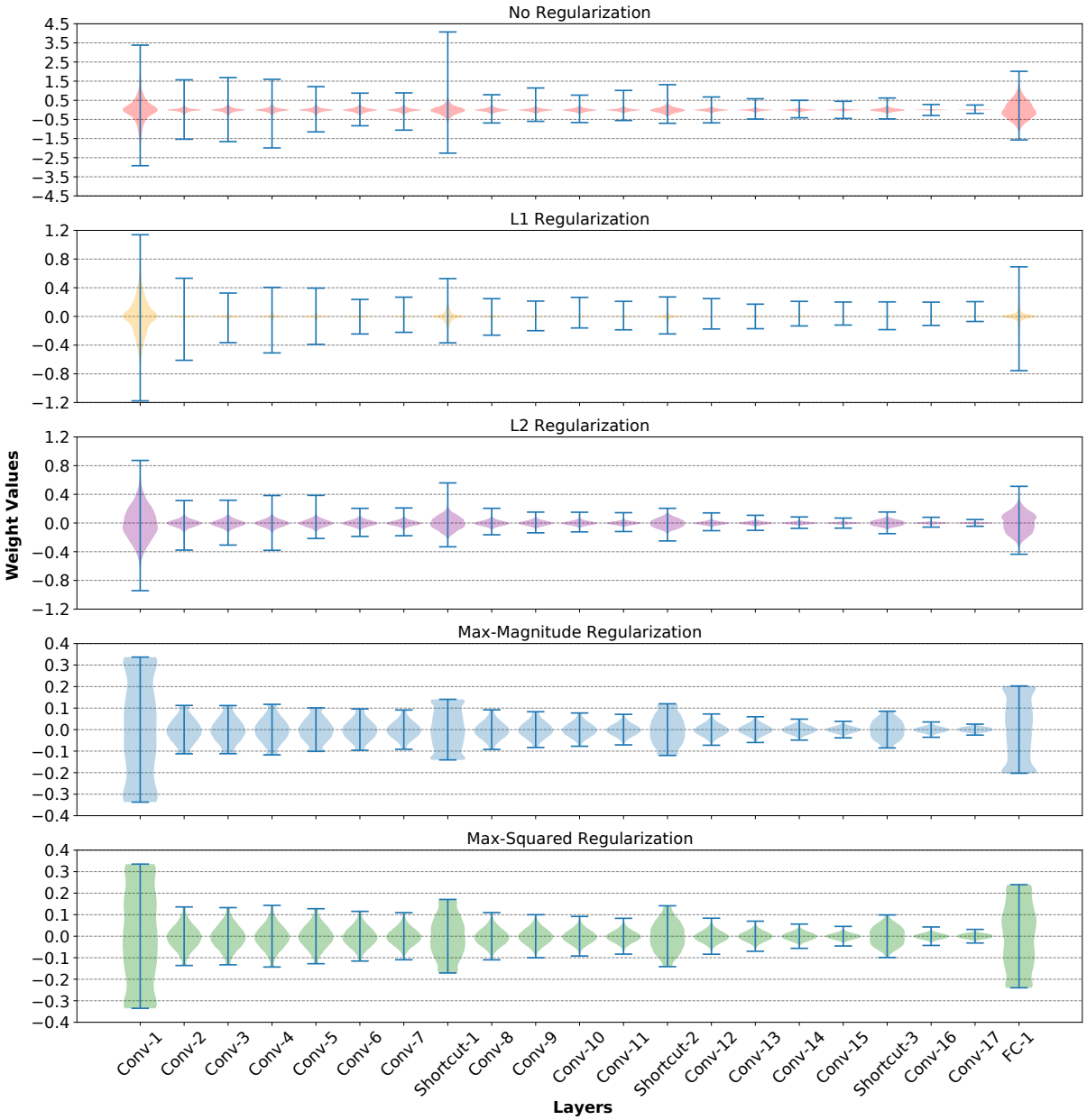

Figure 9.4. Regularization effect on the numerical range of the ResNet-18 parameters.

the layer parameters still remain in the final distributions. Moreover, L1 regularization can even amplify such discrepancies by penalizing the small variables and inducing sparsity in the model while failing to shrink outlier parameters that are significantly higher than the rest of the distribution. In other words, standard regularizers do not deliver the desired outcome, or worse, would result in the exact opposite of the desired effect in the layer parameter distributions.

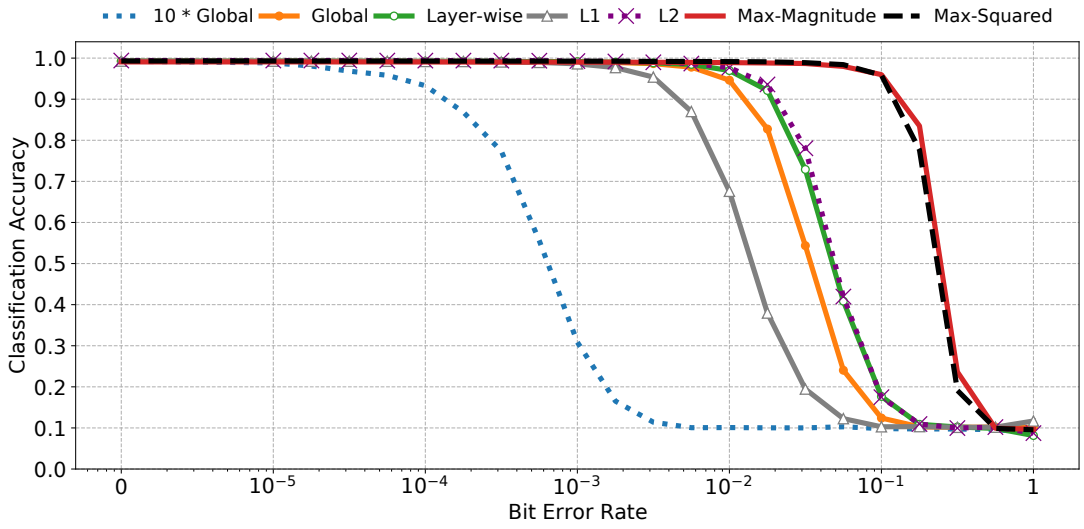

Figure 9.5. Weight error rate vs. LeNet-5 test set classification accuracy on MNIST.

The obtained parameter distributions attained through the proposed outlier regularization methods offer two fundamental advantages over the baseline parameter distributions. First, we expect the numerical impact of the bit errors to be relatively muted compared to an average parameter value due to the well-clustered and better-utilized quantization range. Second, the regularized models can be quantized more accurately because of the clustering behavior and the reduction in the quantization step size. As a result, improvements in both bit error resilience and quantization accuracy are anticipated in the experiments.

9.5.3 Bit Error Resilience Analysis

We investigate the bit error resilience of the target models in the quantized format by injecting errors into model parameters and measuring the test set classification accuracy at various bit error rates. The accuracy curves are shown in Figures 9.5, 9.6, 9.7, and 9.8.

We experiment with various numerical range allocation methods to demonstrate the effectiveness of the proper quantization schemes in delivering high error resilience in neural networks and show further resilience improvements obtained through the described regularization techniques. We first consider a worst-case scenario where the utilized numerical representation

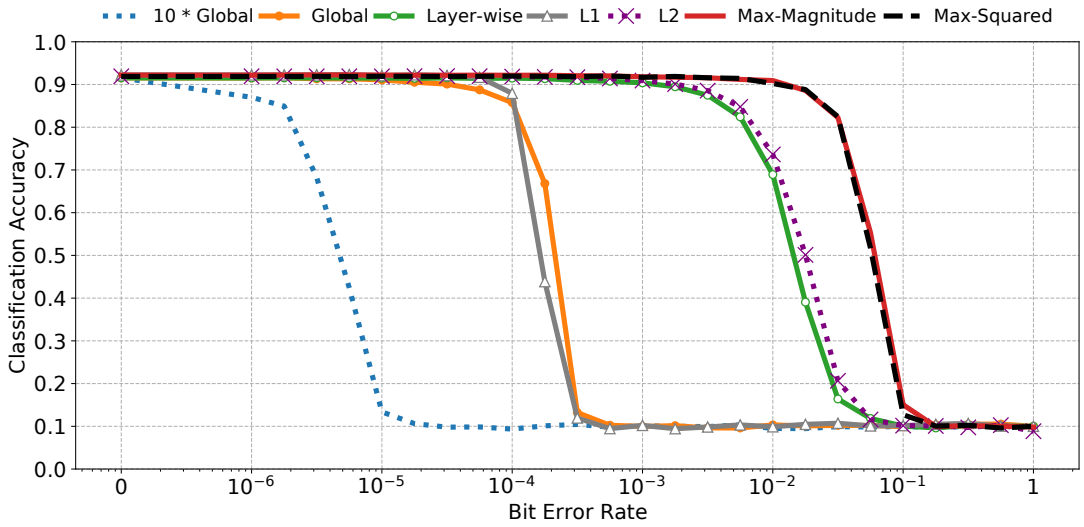

Figure 9.6. Weight error rate vs. VGG-16 test set classification accuracy on CIFAR10.

can cover numbers that are 10 times greater than the numerical bounds necessary to cover the maximum global magnitude of the model parameters (*10 * Global*). Such a scenario could occur in general-purpose programmable hardware (i.e., large width integer formats in commodity CPU architectures) where the offered range delivered through the number of bits and the number representation fails to be utilized by the quantized model parameters. An improved quantization scheme, *Global*, designates a numerical representation range just wide enough to cover the maximum global magnitude of the model parameters, with a single scale factor to quantize the neural network parameters. The *layer-wise* scheme employs unique scale factors at each layer to improve the quantization precision further. The parameter distribution at each layer is quantized individually in this scheme to obtain tighter quantization bounds. Finally, *L1*, *L2*, *Max-Magnitude*, and *Max-Squared* schemes perform layer-wise quantization on the regularized models that are obtained with the corresponding penalty terms.

We reach the following conclusions as a result of the weight error injection experiments performed on the target models. First, the non-utilized numerical range in the number representations constitutes a significant safety vulnerability in the presence of bit errors. For instance, the VGG-16 and the ResNet-18 models experience noticeable accuracy drops at bit error rates even

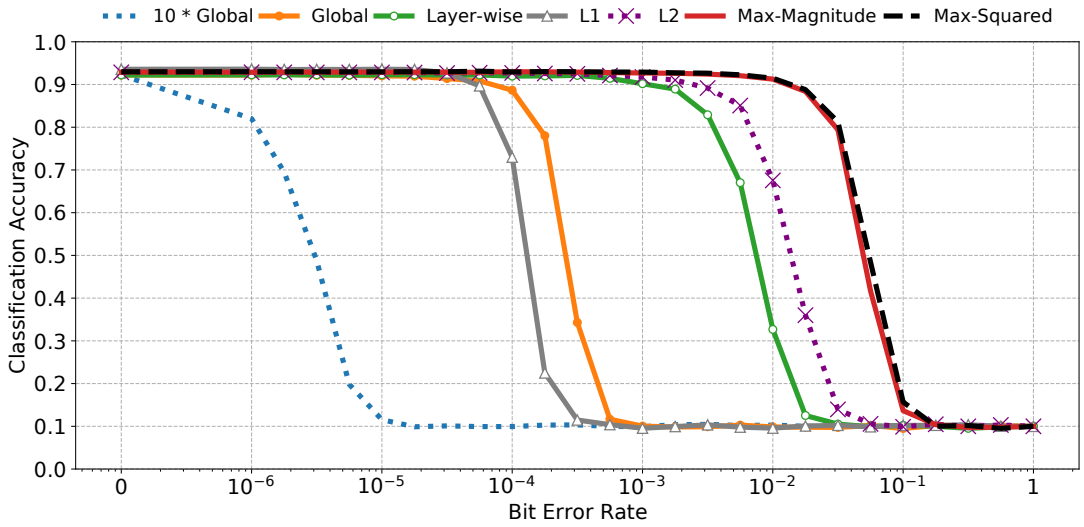

Figure 9.7. Weight error rate vs. ResNet-18 test set classification accuracy on CIFAR10.

smaller than 10^{-6} in the parameters in the *10 * Global* scenario. The described vulnerability can be alleviated with a single scaling factor change when the variables are quantized to cover solely the global parameter range (*Global*). Consequently, the bit error resilience of the models can be improved to tolerate bit error rates up to two orders of magnitude larger. Furthermore, if the dynamic range of parameters is different across the layers as in VGG-16 and ResNet-18, the resilience characteristics can be boosted further by up to 1-2 orders of magnitude through the *Layer-wise* quantization schemes and the use of unique scale factors at each layer. Finally, the regularized models offer resilience to bit error rates an order of magnitude higher when they are layer-wise quantized. The resilient deep neural network models obtained through the combined usage of layer-wise quantization and outlier regularization methods (*Max-Magnitude* and *Max-Squared*) can withstand extreme error rates, namely, bit error rates in the parameters as high as 5% for LeNet-5 and 1% for the VGG-16, ResNet-18, and SqueezeNet models while reining in the accuracy loss at around 1%. The LeNet-5 model can even tolerate parameter bit error rates as high as 10% with classification accuracy losses confined to around 3%.

We observe that standard L2 regularization has at best marginal and often no positive effect on error resilience over the non-regularized models when the parameter distributions are

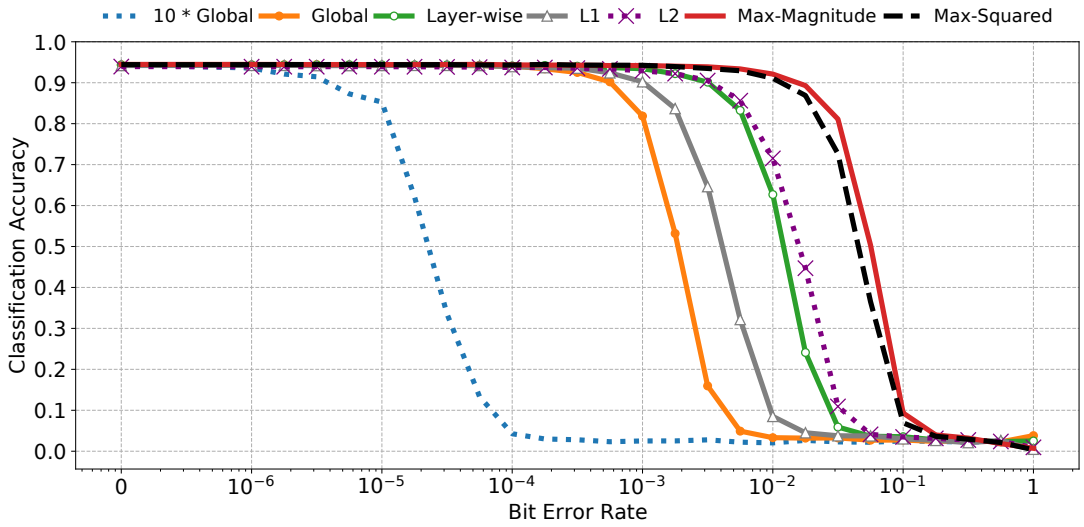

Figure 9.8. Weight error rate vs. SqueezeNet test set classification accuracy on GTSRB.

layer-wise quantized. Furthermore, L1 regularization degrades the bit error resilience of the models noticeably in the quantized form, in accord with the observed large numerical disparity across the layer parameters in Section 9.5.2. As a result, L1 regularized models with layer-wise quantization often exhibit weaker error resilience than the non-regularized models with global quantization. The experimental results clearly indicate that standard regularization techniques are inefficient for our purposes, and the proposed outlier regularization terms are essential for boosting parameter error resilience of deep neural networks.

We compare the parameter bit error resilience of the proposed regularized models (i.e., *Max-Magnitude* and *Max-Squared*) with the previously reported resilience values in prior literature in Table 9.2. Previous work includes neural network resilience methods such as FT-ClipAct [99], Median Feature Selection [5], and Anomaly Suppression [6]. We further compare our results with Binary Neural Networks [212], which are shown to be extremely resilient against bit errors in the parameters [71].

A tolerated weight bit error rate comparison at the same accuracy loss points demonstrates the noticeable superiority of our approach in enduring high bit error rates over the other active error resilience techniques [5, 6, 99]. For instance, our method delivers resilient deep learning

Table 9.2. Tolerated BER (bit error rate) for various deep neural network resilience methods.

		Tolerated Weight BER
LeNet-5 (MNIST)	Binary Neural Nets [71]	10^{-2}
	Median Feature Selection [5]	10^{-4}
	Anomaly Suppression [6]	10^{-3}
	This Work	$\mathbf{5 \times 10^{-2}}$
VGG-16 (CIFAR10)	Binary Neural Nets [71]	10^{-3}
	FT-ClipAct [99]	10^{-5}
	This Work	$\mathbf{10^{-2}}$
ResNet-18 (CIFAR10)	Median Feature Selection [5]	10^{-4}
	Anomaly Suppression [6]	10^{-4}
	This Work	$\mathbf{10^{-2}}$
SqueezeNet (GTSRB)	Median Feature Selection [5]	5×10^{-5}
	Anomaly Suppression [6]	10^{-4}
	This Work	$\mathbf{10^{-2}}$

models that can tolerate 50, 1000, 100, and 100 times larger weight bit error rates for the LeNet-5, VGG-16, ResNet-18, and SqueezeNet benchmarks, respectively, when compared to the highest resilience limits achieved by the others for accuracy loss restrictions of no higher than 1%.

The compared active error resilience methods [5, 6, 99] mainly operate on the activations and tackle the impact of parameter (weight) errors indirectly on the produced layer activations. The effectiveness of our approach against parameter errors stems from the fact that we attenuate the magnitude of the parameter bit errors directly by regularizing and restricting the parameter quantization range, thus largely preventing error diffusion into the activations in the first place. Regularization plays a particularly important role in muting the error effects through numerical range utilization improvements and boosts in the magnitude ratio of parameters to errors, in a manner analogous to improving the signal-to-noise ratio in communication systems. Moreover, it is to be noted that our approach necessitates neither additional operations nor any hardware extensions, unlike [5, 6]. As a result, our method can be deployed seamlessly in existing deep neural network accelerators with no additional overheads.

The obtained models are observed in Table 9.2 to be even more resilient than Binary

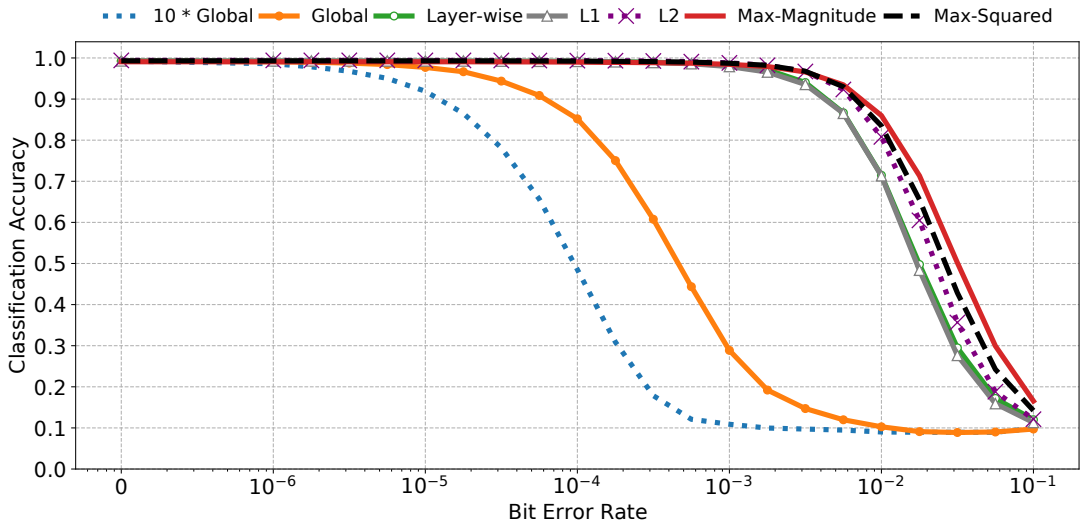

Figure 9.9. Activation error rate vs. LeNet-5 test set classification accuracy on MNIST.

Neural Networks [71], offering resilience for up to 5-10 times larger bit error rates in the parameters. We conjecture that while aggressive quantization schemes such as binarization deliver extensive error resilience by forestalling large error magnitudes (i.e., a parameter can only alternate between +1 and -1), nevertheless, each error transition in a compact binary representation changes the direction of information fully, thus exhibiting noticeably inferior resilience compared to our approach.

We perform similar error injection experiments on the activations by utilizing the same deep learning models. The results for activation error injection experiments are demonstrated in Figures 9.9, 9.10, 9.11, and 9.12. The *10 * Global*, *Global*, and *Layer-wise* activation quantization schemes are analogous to the descriptions provided for the weight quantization methods in the previous experiments. *L1*, *L2*, *Max-Magnitude*, and *Max-Squared* refer to the layer-wise activation quantization performed on the weight regularized models. We forgo the enforcement of any direct regularization on the activations due to the challenges outlined in Section 9.3.2.

A large numerical range in the activations can lead to safety vulnerabilities, similar to the neural network parameters. Squeezing the numerical range to the utilized global maximums

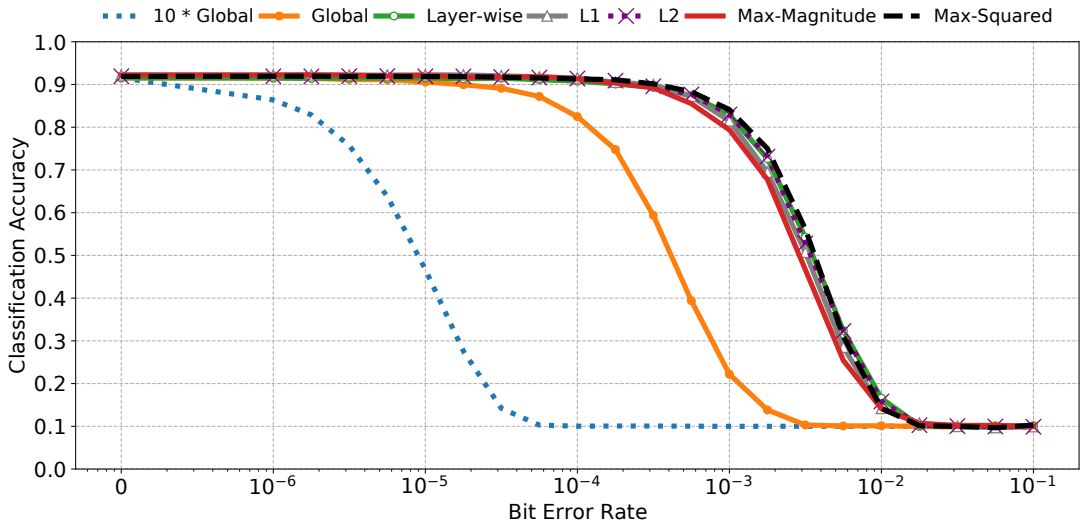

Figure 9.10. Activation error rate vs. VGG-16 test set classification accuracy on CIFAR10.

and further to the unique layer-wise range at each layer can improve resilience noticeably (i.e., resilience up to 1000 times error rates for *Layer-wise* quantization when compared to a loose numerical range in *10 * Global*). We observe that the regularization effect on the parameters can further improve the activation error resilience because regularized weights can indirectly eliminate the large outliers in the activations, reduce the activation quantization range, and consequently, mute the impact of large activation errors. The resilience to excessive activation error rates is often not as critical of a concern as parameter errors because the activations are generated from scratch at each forward inference pass with no long-term impact on model accuracy and thus less likely to exhibit an accumulation effect. However, the additional resilience properties obtained through proper quantization schemes are still desirable and provide additional safety against large impact errors in the activations.

9.5.4 Impact of Regularization Term on Quantization Accuracy

While we demonstrate significant improvements in bit error resilience and non-quantized model accuracy after training, the proposed outlier regularization techniques deliver additional benefits for post-training quantization methods as well. The variable distributions obtained

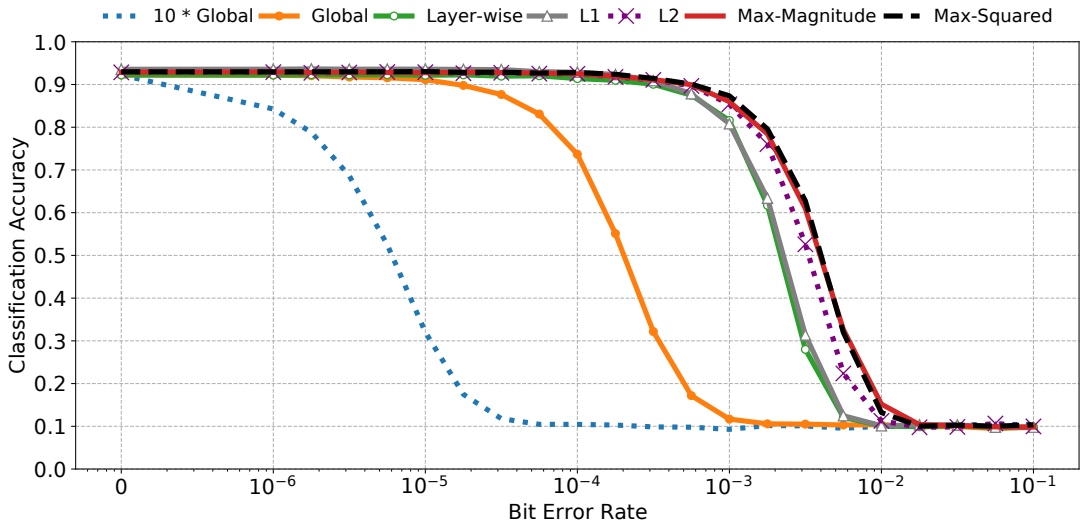

Figure 9.11. Activation error rate vs. ResNet-18 test set classification accuracy on CIFAR10.

through the proposed regularization methods are well clustered into a small dynamic range with no outliers; therefore, such distributions can be quantized to smaller bit widths while maintaining accuracy more gracefully than the non-regularized counterparts.

We conduct post-training quantization experiments on the model weights and activations to demonstrate the outlined advantage of the regularized models. We employ a layer-wise quantization scheme in these experiments by allocating a unique scaling factor for the weights and activations at each layer. The accuracy results are generated by measuring the classification accuracy of the post-training quantized models on the training and test datasets. We demonstrate the quantized model accuracy results in Table 9.3.

Experimental results indicate that LeNet-5 models with outlier regularization can be quantized into 2-bit weights and 4-bit activations directly, and accuracy losses confined to no more than 1% with neither fine-tuning nor compensation steps employed. On the other hand, the non-regularized LeNet-5 model results in around 60% accuracy loss when quantized directly after training. LeNet-5 models with standard regularization techniques experience a noticeable accuracy loss, which is even more severe than the non-regularized LeNet-5 model for the L1 regularization case. We observe similar trends in other neural network benchmarks. For instance,

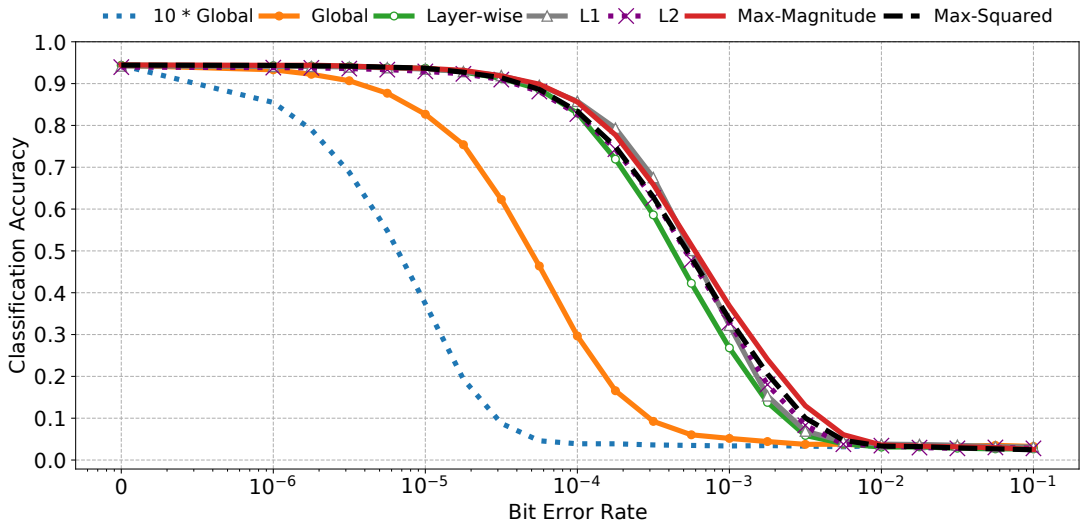

Figure 9.12. Activation error rate vs. SqueezeNet test set classification accuracy on GTSRB.

VGG-16 and ResNet-18 models with outlier regularization do not experience a noticeable loss, and the accuracy is often slightly improved when quantized to 4-bit weights and 8-bit activations. However, the non-regularized VGG-16 and ResNet-18 models incur 1.44% and 1.83% accuracy loss due to quantization, respectively. The accuracy loss values for the L1 and L2 regularized models are observed to be noticeably larger than the models with outlier regularization and even inferior to the non-regularized models in the case of L1 regularization.

While improving the quantization process is not the primary focus of our work, and various advanced techniques do exist to obtain accurate and compact quantized models [34], the outlined observation aims solely to showcase the positive impact of regularization on the quantized model accuracy under post-training scenarios with no fine-tuning.

9.5.5 Observed Differences Between Max-Magnitude and Max-Squared Regularization Terms

The proposed max-magnitude and max-squared regularization terms penalize the outlier parameters at each layer. We observe that both regularization terms can squeeze parameter distributions effectively and deliver significant error resilience improvements. Max-magnitude regularization usually incurs a slightly larger penalty value than the max-squared term at the

Table 9.3. The impact of regularization on the quantized model accuracy.

	Quantization Width (Weight, Activation)	Train Acc. (Change) (%)	Test Acc. (Change) (%)
LeNet-5 Baseline		39.38 (-60.33)	40.21 (-59.03)
LeNet-5 Regularized (L1)		24.21 (-75.65)	24.21 (-75.06)
LeNet-5 Regularized (L2)	(2, 4)	82.99 (-17.01)	82.17 (-17.24)
LeNet-5 Regularized (Max-Magnitude)		98.65 (**-0.66**)	98.41 (**-0.66**)
LeNet-5 Regularized (Max-Squared)		**99.16** (-0.81)	**98.59** (-0.71)
VGG-16 Baseline		98.63 (-1.08)	90.11 (-1.44)
VGG-16 Regularized (L1)		97.64 (-2.01)	89.86 (-2.38)
VGG-16 Regularized (L2)	(4, 8)	98.92 (-0.88)	90.42 (-1.51)
VGG-16 Regularized (Max-Magnitude)		99.71 (-0.05)	**92.02** (-0.19)
VGG-16 Regularized (Max-Squared)		**99.88** (**-0.02**)	91.93 (**+0.04**)
ResNet-18 Baseline		98.05 (-1.32)	90.39 (-1.83)
ResNet-18 Regularized (L1)		98.17 (-1.69)	91.36 (-2.21)
ResNet-18 Regularized (L2)	(4, 8)	99.48 (-0.34)	91.97 (-0.88)
ResNet-18 Regularized (Max-Magnitude)		99.77 (+0.01)	**92.89** (**-0.01**)
ResNet-18 Regularized (Max-Squared)		**99.90** (**+0.04**)	92.81 (-0.16)
SqueezeNet Baseline		99.18 (-0.65)	93.01 (-1.35)
SqueezeNet Regularized (L1)		96.39 (-2.81)	90.43 (-3.77)
SqueezeNet Regularized (L2)	(4, 8)	99.21 (-0.62)	92.20 (-1.74)
SqueezeNet Regularized (Max-Magnitude)		**99.55** (**-0.24**)	**93.84** (**-0.63**)
SqueezeNet Regularized (Max-Squared)		99.40 (-0.42)	93.57 (-0.85)

same regularization coefficient (α) since neural network parameters are often observed to be small values between [-1, 1], where the variable magnitude is numerically larger than its squared version. However, the difference between the obtained parameter distributions through these regularization terms is observed to be minimal in practice.

9.6 Discussion

The resilience of deep neural networks is commonly observed in previous studies. Neural networks can maintain accuracy gracefully, even under widespread noise and error conditions, as long as the captured information in the model variables is not overwhelmed by noise and error incidents. For instance, neural networks are known to maintain accuracy gracefully during

quantization, which is a process of widespread error introduction. From this perspective, error tolerance in neural networks bears significant similarities to the problem of improving the SNR (signal-to-noise ratio) in digital signal processing and telecommunication systems.

While the outlined observations draw a hopeful conclusion regarding the resilience of a neural network, bit error effects whose numerical impact stretches well beyond the described noise levels remain its Achilles' heel. The resilience properties of the neural networks can begin to shatter rapidly under large magnitude error incidents. The proper design choices that preclude such scenarios not only improve error resilience dramatically but may furthermore boost the accuracy of the system noticeably under error-free conditions. For instance, the proper utilization and the constriction of the numerical range in the quantization schemes improve both the quantization quality and tighten the potential error margins effectively.

We further demonstrate how the distribution of the model parameters can be formed in the desired manner through the proper training goals to obtain deep learning models that are highly error-resilient in the quantized form. Surprisingly, such restricted models can be even more accurate than their unrestricted counterparts in both full-precision and quantized formats.

In summary, we illustrate that deep learning hardware, when properly designed, can tolerate excessive bit errors in the datapath at levels even up to 10%, offering resilience shields unmatched by much costlier traditional fault-tolerant systems. Moreover, such resilient systems are constructed at no additional cost enjoying several auxiliary benefits such as improved model accuracy.

9.7 Chapter Summary

As the computational burden of deep neural networks challenges the delivery of functional safety in embedded deep learning applications through traditional fault tolerance techniques, we investigate an alternative path to embed unparalleled reliability characteristics into deep neural networks by utilizing their inherent resilience. Our results in this chapter indicate that

properly designed quantization schemes can reduce error margins and improve system resilience noticeably against bit errors in the neural network variables. We propose a novel regularization term to manipulate the parameter distributions at each layer and tighten error margins further. Neural network models obtained through the described regularization process are shown to be highly accurate after training, lend themselves to accurate quantization at low bit widths with minimal accuracy loss, and exhibit extreme levels of resilience to hardware bit errors in the quantized format. This chapter presents strong evidence that highly accurate and error-resilient deep learning systems can be built proactively through algorithmic modifications and proper design techniques while imposing essentially zero hardware, energy, and performance overheads.

9.8 Acknowledgements

Chapter 9 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021 ([7]). The dissertation author was the primary investigator and author of this paper.

Chapter 10

Boosting DNN Hardware Yields via Cost-Effective Defect Adaptation

The micro-architectural features of deep learning accelerators, when paired with the algorithmic characteristics of DNNs (deep neural networks), unlock novel and proactive opportunities to tackle semiconductor reliability problems in embedded deep learning devices. While the fine-grained bypassing of the faulty processing elements reins the computational impact of hardware defects, a one-time training of deep neural networks with Hardware-Aware Dropout/Dropconnect techniques proactively boosts model decentralization and facilitates accurate neural network inference in the degraded computational fabrics. Furthermore, on-device calibration methods can improve resilience even further without necessitating expensive defect compensation methods such as device-specific training. This chapter investigates the potential opportunities for improving the yield, reliability, and operational lifetime of embedded machine intelligence devices through a practical co-design of deep neural networks and configurable hardware architectures.

10.1 Introduction

The parallel computational resources and micro-architectural regularity of DNN accelerators spawn effective opportunities for tackling hardware defects. A spatial DNN accelerator is typically made up of hundreds, and often up to thousands, of small processing elements to

carry out a large number of multiply-accumulate operations [26, 35, 36, 37]. Their regular micro-architecture makes defective processing element isolation highly practical even at the finest granularities with a negligible loss in the underlying hardware capabilities.

Deep neural networks are known to tolerate certain error types, particularly when the numerical impact of the error is limited in the computations. The inherent resilience characteristics of deep neural networks impart an opportunity to compensate for the computational effects of defective hardware components through isolation, without necessitating hardware redundancy or incurring performance overheads. Furthermore, it is often possible to enhance the inherent resilience characteristics of deep neural networks with novel training techniques and maximize the potential benefits through the boosted resilience of deep neural networks.

As its primary contribution, this chapter advocates for a co-design methodology to tackle hardware defects in embedded deep learning accelerators by utilizing micro-architectural hardware features and algorithmic characteristics of deep neural networks. The innovative co-design aspect of this work stems from the fact that hardware and software enhancements complement each other in a synergistic manner to improve overall reliability, and both hardware and software parameters influence each other strongly in the design process.

As an initial step, we bypass the computational contribution of defective processing elements in a DNN accelerator through a mechanism that is similar to [79]. The resulting pruning effect is observed to be more palatable than leaving the hardware fault effects intact in neural network computations.

Despite the benefits of fault isolation through bypassing, an aggressive pruning effect on the variables due to bypassed hardware components could still lead to a noticeable accuracy loss particularly when the salient variables are impacted in a standard neural network. This accuracy deterioration limits the practical applicability of hardware bypassing, particularly at high defect rates. Fortuitously, deep neural networks could minimize the saliency of individual variables and cultivate a highly decentralized computational structure when they are motivated through proper training goals. The specific training methods that are traditionally adopted to tackle the over-

fitting problem, including Dropout [59] and Dropconnect [60], could embed such characteristics into deep neural networks, and their effectiveness against realistic hardware failure scenarios could be improved noticeably when the micro-architectural hardware parameters (e.g., dataflow type, systolic array dimensions) and the hardware mapping strategy (e.g., matrix tiling process) are taken into consideration in the training process. We demonstrate that deep neural networks obtained through such training could tolerate significantly higher rates of bypassed processing elements when compared to standard neural network models. Since neural network models with a decentralized structure maintain information content much more gracefully, the necessity for costly accuracy compensation techniques such as device-specific training [79, 80, 118] is obviated.

We further demonstrate that practical on-device calibration techniques can seamlessly complement the outlined design-time modifications, minimize accuracy loss due to fault effects in these systems, and extend neural network resilience to even higher hardware bypass rates. Namely, we propose two lightweight calibration techniques that can be carried out within the resources of the faulty embedded device. First, we ameliorate the accuracy loss caused by the mismatch of batch statistics in a faulty device through cost-effective calibration of the batch normalization layer statistics. The outlined calibration step is performed by running a small number of inference operations on the faulty device. Second, we observe that the order of neural network computations can be permuted by rearranging neurons/filters to yield computationally equivalent neural network structures, yet the pruning manifestation of the same bypassed hardware component differs widely in these neural networks. We propose a practical computational rearrangement mechanism that can be implemented on an embedded device at minimal cost to effect a rearrangement that mutes the inescapable impact of pruning on accuracy.

This chapter starts with the problem definition in Section 10.2. We describe the proposed approach in Section 10.3. The implementation details and the results of our experimental analysis are presented in Sections 10.4-10.5. We conclude our discussion in Section 10.6.

10.2 Problem Definition

Cutting-edge semiconductor technologies with ever-diminishing feature sizes have contributed to the overall efficiency of semiconductors, including hardware architectures for deep learning. While physical defects in the manufacturing process (e.g., open/short connections), timing failures caused by process variations, and aging effects [220] are more likely to manifest as permanent hardware faults in the advanced semiconductor technology nodes [40, 41, 42, 43, 44], the criticality of these issues has been exacerbated in DNN accelerators that consist of up to thousands of processing cores.

When a permanent fault impacts the high-order bits in a processing element, the numerical effect of a fault could be highly significant at the layer outputs. Furthermore, the computational impact of a permanent fault is often aggravated due to the reuse of the same hardware component in the subsequent stages of the computation. As a result, even a few faulty processing elements are shown to impair deep neural network accuracy critically in the prior studies [79, 80].

While the design of a systolic array that is capable of tolerating permanent defects is a decades-old interest in the research community [94, 221], conventional methods often discard the faulty units in a coarse-grained manner. As a result, these methods either suffer from performance overheads due to the reduced form of the systolic array, or require additional redundancy in the computational fabric to avoid the performance impact.

The computational plasticity of DNNs is harnessed in recent studies to overcome the rigid computational constraints of the defect tolerance problem in deep learning accelerators. Novel techniques such as [79, 80, 118] compensate for the computational effect of the bypassed processing elements by modeling the consequent pruning impact on the neural network, and performing fine-tuning on the pruned model to recover accuracy. While this approach can ameliorate accuracy loss and eliminate performance impact, its fundamental limitation stems from the fact that it requires a dedicated fine-tuning process for *each* faulty device with a unique defect pattern.

Even more concerning is that strict cost budgets play a determinative role in terms of industry adoption for low-cost embedded devices. Fault compensation techniques that require device-specific training [79, 80, 118] necessitate dedicated computing resources (i.e., on the cloud) to be allocated for each faulty device for a period of time and suffer the consequent highly undesirable scaling cost model for each manufactured faulty device, thus greatly diminishing their economic appeal. Furthermore, the outlined device-specific process needs to be repeated for each deep learning model if multiple neural network models are planned for deployment in the target device. Finally, device-specific training creates further practical challenges in the case of faults that surface after device deployment, such as aging effects.

The practical applicability of defect tolerance methods can be improved significantly for deep learning hardware if algorithmic innovations could help embed inherent resilience characteristics into DNNs through a *one-time* training process. Furthermore, device-specific aspects of the underlying defect patterns could be addressed within the capabilities of the faulty embedded device through on-device calibration techniques. As a result, the practical benefits of novel defect tolerance techniques can be assured in embedded deep learning hardware by incorporating minimal additional hardware configurability and utilizing unique algorithmic characteristics of DNNs.

10.3 Proposed Method

We outline a proactive co-design methodology for deploying deep neural networks in defective hardware accelerators. The proposed design flow comprises four fundamental steps:

- We avoid the unpredictable error deviations induced by permanent hardware defects through hardware configurability and bypassing of faulty hardware components.
- We train neural networks with *Hardware-Aware Dropout/Dropconnect* methods to boost decentralization and minimize information degradation due to bypassing.

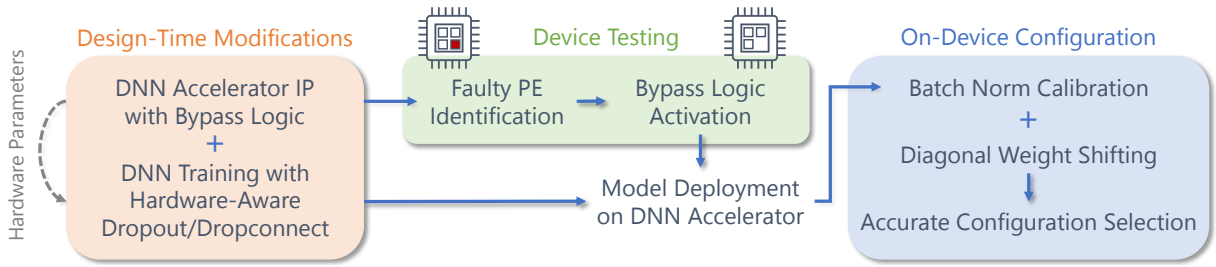

Figure 10.1. Overall summary of the proposed design flow.

- We further improve the accuracy of deep neural networks with batch normalization layers through the calibration of batch normalization statistics on the faulty device.
- We explore a slew of pruning manifestations for the bypassed hardware units through a lightweight hardware mapping re-adjustment and minimize accuracy loss by identifying the more benign pruning occurrences.

The outlined approach eliminates the need for hardware redundancy and costly fault compensation methods such as device-specific training, and it facilitates high accuracy even under extreme hardware defect rates in embedded deep learning accelerators. The proposed design flow is outlined Figure 10.1.

10.3.1 Isolating Faults Through Hardware Configurability

While permanent hardware faults could impact the accuracy of neural networks critically due to the possibility of large-magnitude error deviations, deep neural networks exhibit a more graceful accuracy degradation under particular error manifestations, such as when the numerical contribution of neural network variables is dropped from the computations by setting them to zero, as discussed in Section 8.4.1. The outlined resilience characteristics thus enable innovative techniques for handling permanent hardware defects in embedded DNN accelerators.

Spatial deep neural network accelerators such as systolic arrays consist of a highly-regular grid of small processing elements to provide parallelism in the underlying tensor operations. Moreover, their fine-grained and regular micro-architecture facilitates fault isolation opportunities

in the defective devices with a minimal loss in hardware capabilities.

Fault isolation techniques identify the processing elements with hardware faults through testing, and bypass the computational contribution of faulty processing elements. The outlined approach herein is agnostic to the particularities of the test method being employed, as long as the test process provides pass/fail information within the granularity of the processing elements in the systolic array. For instance, the proposed approach could be paired with manufacturing tests applied through scan chains to tackle manufacturing defects, or on-device testing techniques such as BIST (built-in self-test) for continuous monitoring against aging-related faults [220].

The action of bypassing relies on a limited amount of additional configurability in the hardware IP. We first summarize the bypass method introduced in [79] for the weight-stationary architecture, then propose bypass mechanisms for input-stationary and output-stationary dataflows.

Fault Isolation in Weight-Stationary Dataflow

The bypass mechanism for a weight-stationary architecture is demonstrated in Figure 10.2-A. The introduced multiplexing logic at each processing element allows a faulty unit to forward the received partial sum directly to the neighboring processing element. As a result of bypassing, all the weight values that map into the bypassed processing element are effectively pruned from neural network computations.

Fault Isolation in Input-Stationary Dataflow

The bypass logic for the input-stationary architecture resembles the one used in the weight-stationary design. However, the computational impact of bypassing differs since it results in the pruning of layer inputs that map into the bypassed processing element.

Fault Isolation in Output-Stationary Dataflow

The numerical impact of a hardware fault in the output-stationary architecture is confined to the partial sum values that are produced in the faulty processing element; therefore, layer outputs that are pinned into the faulty processing element should be masked by setting to zero.

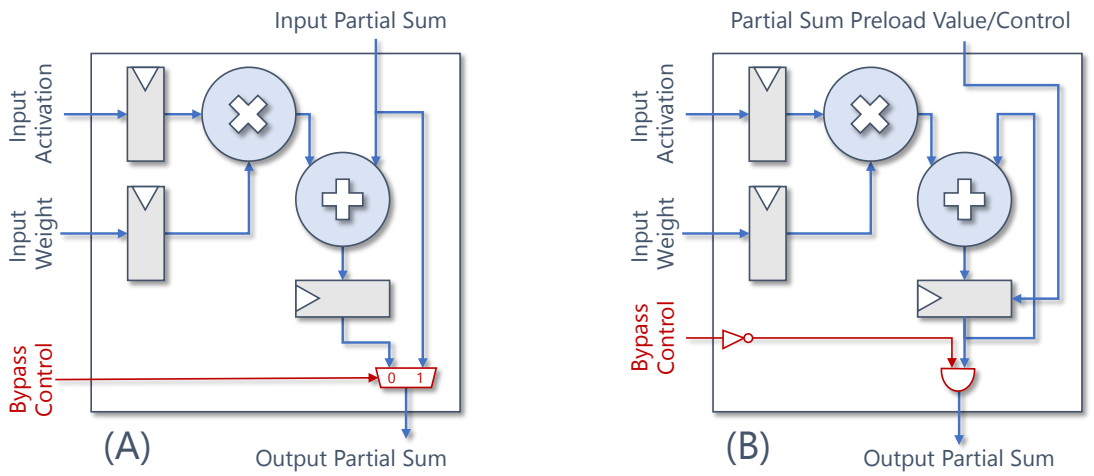

Figure 10.2. Bypass logic in (A) weight/input-stationary and (B) output-stationary processing elements.

The bypass mechanism in the output-stationary dataflow is demonstrated in Figure 10.2-B, and it can be implemented through a series of AND gates in lieu of the multiplexers used in the two alternative dataflows. The bypass mechanism prunes the layer outputs that map into the faulty processing elements in this architecture.

The proposed bypass mechanisms could isolate all permanent faults in a processing element except specific points in the bypassing circuitry. While the likelihood of a permanent defect occurring in the bypassing circuitry is small due to its minimal area compared to the processing element (as demonstrated in Table 10.1), costlier bypass mechanisms [118] can be utilized for mitigating faults in the additional logic as well.

10.3.2 Minimizing Information Loss with Decentralized DNNs

While the substitution of the unpredictable numerical impact of hardware faults through the bypassing of faulty hardware components results in a more tolerable pruning effect on the variables, it could still induce noticeable information loss in the standard neural networks even at moderate bypass rates.

Weights and activations are known to be associated with varying levels of saliency in neural network computations [85, 87]. While the pruning of a relatively critical weight or

activation is bound to be more noticeable on the overall accuracy, the saliency imbalance among the neural network variables can be alleviated through properly introduced training goals.

We propose a training approach with methods that are similar to Dropout or Dropconnect to encourage decentralization in the training process and reduce the saliency of individual neural network variables. In addition to alleviating co-adaptation and over-fitting problems, Dropout/Dropconnect training downsizes the significance of individual activations/weights and thus encourages decentralization in the applied neural network layers.[10] The deep neural network models obtained as a result of such training are expected to operate accurately through a subset of variables and tolerate the numerical effect of the dropped variables gracefully.

The benefits of the described training approach can be further boosted by reflecting the hardware characteristics accurately in this process. First, the impacted variable type (i.e., weight, layer input, layer output) is determined by the dataflow type of the DNN accelerator. Input-stationary and output-stationary dataflows necessitate a process resembling Dropout at the layer inputs and outputs, respectively. Meanwhile, a technique similar to Dropconnect is required for bypassing effect characterization in the weight-stationary designs.

Moreover, the systolic array shape and the utilized neural network mapping scheme influence the manifestation of hardware bypassing on deep neural network computations. We explore two unique versions of the training process with Dropout/Dropconnect. In *Random Dropout/Dropconnect*, the drop events for each weight, input, or output variable are controlled by unique independent random variables. While this approach resembles conventional Dropout/Dropconnect, it may not accurately reflect the hardware reuse effect on computations. The latter approach, *Hardware-Aware Dropout/Dropconnect*, takes the systolic array shape and the mapping scheme into consideration in the process of dropping variables. Hardware-Aware Dropout/Dropconnect mirrors the hardware bypassing behavior more accurately and improves training effectiveness in building strong algorithmic resilience against the bypassed components.

[10]The term "decentralization" defines a DNN characteristic where the individual variable significance for the correct output decision is moderated and the decision is generated through the collective contribution of variables.

Random Dropout/Dropconnect

This technique is applied at training time by dropping layer weights, inputs, or outputs, depending on the targeted dataflow type. We first generate a random mask tensor at each layer where the mask shape is equal to the shape of the target tensor to which the Dropout/Dropconnect is applied. We sample each mask entry from an independent Bernoulli distribution where the probabilities for values 1 and 0 are p and $1-p$, respectively. We element-wise multiply the target tensor (X) with the generated mask (M) and propagate the masked version of the tensor ($\hat{X}$) in the forward pass, as demonstrated in Equation (10.1). Similarly, gradients are back-propagated only to the variables that are not dropped in the forward pass and set to 0 for the dropped variables, as in Equation (10.2). We generate unique masks for each mini-batch to cover different variable subsets in the training process.

$$\hat{X} \quad \leftarrow \quad M \quad \odot \quad X \tag{10.1}$$

$$\partial Loss/\partial X \quad \leftarrow \quad M \quad \odot \quad \partial Loss/\partial \hat{X} \tag{10.2}$$

Hardware-Aware Dropout/Dropconnect

Random Dropout/Dropconnect reduces the saliency of individual variables, yet it often falls short in the accurate characterization of the hardware bypass behavior. DNN computations are performed by mapping variables in groups into the *same* computational fabric in a sequential manner. Therefore, the variables mapped into a processing element at different time steps are all affected by the bypassing of that unit. A more precise training approach thus requires modeling the bypass behavior through independent random variables generated for the hardware components and propagating them to the variables for accurate bypass effect characterization.

The mapping of the neural network computations into a systolic array is determined statically at compile-time, and thus knowledge of a few micro-architectural parameters and the

utilized mapping algorithm often suffices to pinpoint the variables impacted by the bypassing operations. First of all, the variable type that is pinned to the processing elements is pruned as a result of hardware bypassing. Furthermore, if the systolic array shape and the details of the mapping scheme are known beforehand, the variables that are pruned due to the bypassed processing element can be identified deterministically through modulo arithmetic and simple indexing operations.

To illustrate, let us assume a symmetric systolic array shape $N \times N$ where N indicates the number of columns and rows in the systolic array. In the scope of this work, we utilize simple mapping schemes for weight-stationary, input-stationary, and output-stationary dataflows that generalize well to both fully connected and convolutional layers. The weights of the fully connected layer can be represented with a tensor of shape $C_o \times C_i$, where C_o and C_i denote the output and input dimension sizes. A tensor of shape $C_o \times C_i \times S_w \times S_w$ is considered for the convolution layer weights, where C_o, C_i, and S_w are the output channel, input channel, and spatial dimension size of the weight tensors, respectively. In the weight-stationary dataflow, each neuron or convolution filter is mapped into a systolic array column by tiling the weight tensors in the output and input dimensions. As a result, the column and row indices (in the given order) of the mapped processing element (PE) for each weight could be identified through Equation (10.3) where c_o and c_i are the output and input dimension indices of the weight variable in the fully connected and convolutional layers:

$$PE(c_o, c_i) = (c_o \% N,\ c_i \% N) \tag{10.3}$$

The fully connected layer input is represented by a tensor of shape $B \times C_i$, where B is the batch size. Assuming the spatial dimension size of the input is S_i in a convolutional layer, the convolutional layer input is defined by a tensor of shape $B \times C_i \times S_i \times S_i$. In the input-stationary dataflow, each batch index can be mapped into a systolic array column by tiling the input tensor in the batch and input dimensions. Assuming b and c_i signify the batch and input dimension

indices of an input variable, the column and row indices of the mapped processing element can be determined through Equation (10.4) in the fully connected and convolutional layers:

$$PE(b, c_i) = (b\%N,\ c_i\%N) \tag{10.4}$$

A similar formulation can be considered for the output-stationary dataflow with the output tensors of shape $B \times C_o$ and $B \times C_o \times S_o \times S_o$ in the fully connected and convolutional layers. S_o signifies the spatial dimension size of the convolutional layer output tensor. The mapping process in the output-stationary dataflow can be performed by tiling the output tensor in the output and batch dimensions and mapping neuron/filter outputs into unique systolic array columns. If the output and batch dimension indices of an output variable are indicated by c_0 and b, the column and row indices of the mapped processing element can be identified through Equation (10.5) in the fully connected and convolutional layers:

$$PE(c_0, b) = (c_0\%N,\ b\%N) \tag{10.5}$$

The process is visualized for a weight-stationary dataflow in Figure 10.3. Hardware-Aware Dropout/Dropconnect first generates a hardware bypass mask of shape $N \times N$ consisting of random variables sampled from a Bernoulli distribution. The bypassed unit is indicated with the dashed lines in the example hardware mask in Figure 10.3-1. The hardware mask is then utilized for the mask generation for DNN variables.

The weights of a fully connected layer can be represented in the form of a matrix, where each column corresponds to the connections of a unique neuron. In Figure 10.3, we indicate each neuron and its corresponding weight matrix column with a unique color. Each weight entry is mapped to a single processing element in the weight-stationary dataflow, and when the dimensions of the weight matrix are larger than the systolic array, the layer is processed in multiple steps through the tiling of the weight matrix. For instance, the weight matrix in Figure 10.3 needs to be divided into four tiles and processed subsequently. We replicate the

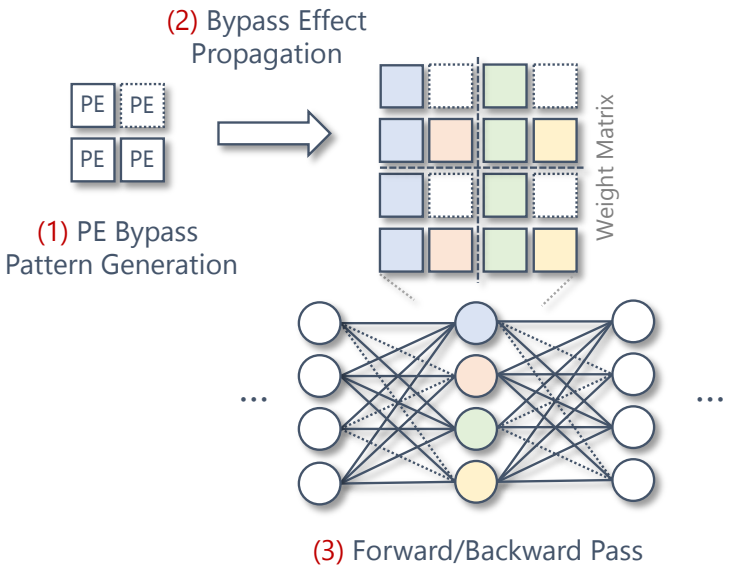

Figure 10.3. Hardware-Aware Dropconnect in weight-stationary dataflow.

hardware bypass mask to mimic the tiling behavior in the DNN mapping process and obtain a mask tensor with the same shape as the target DNN tensor. The mask tensor obtained through the replication process may need to be cropped if the tiled dimensions of the target neural network tensor are not an integer multiple of the corresponding systolic array dimension. As a result of bypass effect propagation (Figure 10.3-2), the weight entries that are mapped to the bypassed processing element at each tile are pruned. The pruned weight entries are indicated with the dashed lines both on the weight matrix and the neuron connection diagram. The outlined mask generation process results in perfectly correlated pruning behavior for neural network variables that are mapped to the same processing element, and thus aligns with the actual hardware behavior more accurately. After mask generation, the training process is carried out with the dropped connections through steps similar to Random Dropout/Dropconnect in terms of the forward and backward pass behavior.

10.3.3 Calibrating Statistical Properties of Faulty DNNs

Batch normalization is abundantly used in modern DNNs due to its benefit of accelerating training convergence. Furthermore, batch normalization is shown to be essential for training

deep architectures and particular DNN types such as BNNs (binary neural networks) [212].

Batch normalization improves the training process stability of deep neural networks against input distribution shifts; however, it utilizes pre-profiled mean and variance values at inference time instead of computing the batch statistics from the inference examples. As a result, deep neural networks with batch normalization layers are not immune to distribution shifts at inference time and are bound to suffer accuracy loss if the batch statistics differ from the pre-profiled values.

A distribution shift is likely to occur in the case of bypassed hardware components since dropping neural network variables will inevitably alter the layer output distribution. The consequent deviation from the pre-profiled batch statistics could thus incur a significant accuracy loss in the bypassed hardware fabrics if the pre-profiled statistics from training are used for normalization in the inference phase. The outlined problem is particularly exacerbated in the weight-stationary architectures since weight tensor alterations will impact all layer outputs in the inference batch and distort batch statistics more drastically.

This problem can be remedied to a great extent by calibrating batch normalization statistics on the embedded device with bypassed processing elements. The calibration process can be accomplished by running inference on a small set of examples on the faulty accelerator, extracting the mean and variance values for the outputs of each neuron or the convolution filter through a cumulative moving average (Equations 10.6-10.7), and replacing the old pre-profiled values with the newly obtained statistics. In Equations 10.6-10.7, μ_{global} and σ^2_{global} indicate the updated cumulative statistics in the calibration process, μ_B and σ^2_B signify the statistics of the current batch, and n corresponds to the current mini-batch index in the inference process.

$$\mu_{global} = \mu_B/(n+1) + (n \cdot \mu_{global})/(n+1) \tag{10.6}$$

$$\sigma^2_{global} = \sigma^2_B/(n+1) + (n \cdot \sigma^2_{global})/(n+1) \tag{10.7}$$

While using more examples in the calibration process provides higher confidence in the estimation of these statistics, we observe that a small number of examples (i.e., a few mini-batches that consist of a few hundred examples in total) often suffice in boosting accuracy to the highest possible level. The outlined process does not require costly device-specific training on the cloud and is performed on the embedded device without necessitating hardware extension.

10.3.4 Searching for Benign Pruning Patterns

The wide variation between the saliency of deep neural network variables can be alleviated through training with Dropout/Dropconnect, yet muted differences could still persist to a degree in the obtained deep neural network models. As a result, the final accuracy of a neural network could vary somewhat depending on which neural network variables are impacted as a result of the hardware bypassing operations.

The computational regularity and the parallelism of deep neural networks afford computational rearrangement through their permutational invariants. The order of neurons in a fully connected layer can be rearranged by permuting the output dimension of the weight matrix. This modification requires rearrangement in the input connections of the subsequent layer to ensure computational correctness, which can be achieved by applying the same rearrangement pattern to the input dimension of the weight matrix in the next layer. A similar modification can be applied in the convolutional layers by permuting the output channel dimension of convolutional layer weights and implementing the same rearrangement to the input channel dimension of the weights in the subsequent layer.

While the described neuron/filter permutations result in mathematically equivalent formulations and deliver consequently the same accuracy under a fault-free scenario, they are bound to yield diverse pruning effect manifestations and thus lead to divergent accuracy values on a systolic array with fixed bypassed processing element locations.

The outlined property allows us to boost accuracy by searching among neuron/filter permutations to pinpoint configurations where the effect of the systolic array bypass pattern is

less damaging to accuracy. On the other hand, such a permutation space is often quite large in practice, and measuring the accuracy of each candidate imposes a prohibitive cost. Alternatively, accurate candidates can be identified by optimizing a proxy cost function, such as minimizing the overall norm of the variables that map into the faulty processing elements [118], yet the optimization of even such proxy functions may not be a trivial task within the resource constraints of an embedded device. In addition, a flexible rearrangement mechanism that can instantiate all feasible permutations could be costly to implement on an embedded accelerator.

In light of these, we propose a lightweight approach to enable an embedded DNN accelerator to evaluate a small subset of such permutations, and boost system accuracy by circumventing particularly detrimental pruning manifestations. The proposed method involves shifting the weight tensors of each fully connected and convolutional layer diagonally in the input (channel) and output (channel) dimensions. For a systolic array with shape $N \times N$, this approach provides N distinct permutations that yield unique pruning effect manifestations before starting to repeat in modulo N for shift amounts larger than N. Figure 10.4 illustrates the diagonal shift operation with two different pruning manifestations for a 2×2 weight-stationary systolic array. The configuration with the highest accuracy could then be selected to improve overall accuracy.

The diagonal shift mechanism guarantees correct execution since the same permutation pattern is applied to both the input and the output dimensions of the weight matrices at every layer. Furthermore, this technique can be implemented through a shifted indexing mechanism while fetching weight tensors from memory; it neither requires dedicated hardware nor introduces irregularity in the memory accesses. Finally, shifting weight tensors diagonally leads to a similar amount of shift in the single dimension of the layer input and output tensors and thus produces a similar desired outcome in the input-stationary and output-stationary dataflows.

Overall, an embedded DNN accelerator can quickly test N shifted weight tensor permutations on a small set of labeled examples to identify the configuration with the highest accuracy. The input examples used for calibrating batch normalization layers could again be employed to measure the accuracy of each configuration without necessitating additional data.

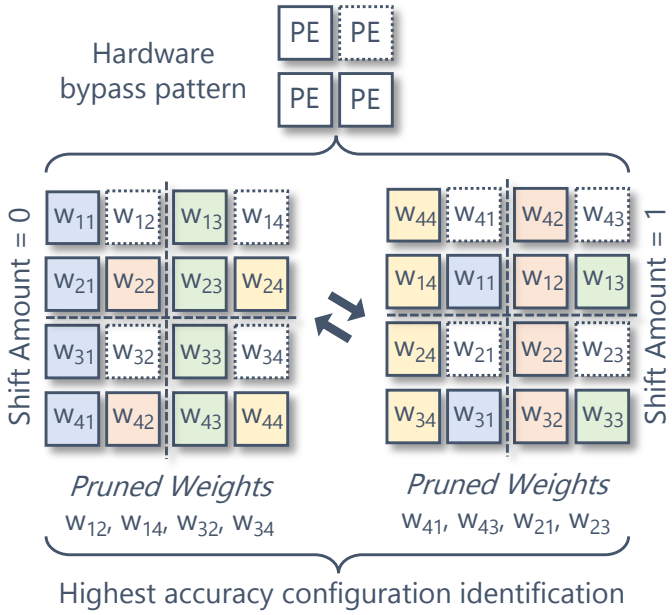

Figure 10.4. Diagonal shift operation in weight-stationary dataflow.

10.4 Experimental Method

Our analysis focuses on permanent faults in the processing elements. We model permanent fault effects on the module level by following the practice of [79, 80, 93, 118]. The term *PE Bypass Rate* indicates the percentage of processing elements with at least a single fault. We generate PE fault (i.e., bypass) patterns as a set of Bernoulli random variables of shape $N \times N$, where the fault probability is independent for each PE and controlled by the PE bypass rate. We assume that conventional manufacturing tests or built-in self-test capabilities are utilized to identify faulty processing elements, and that their impact is mitigated through the bypass mechanisms.

We utilize a variety of DNN models in our experimental analysis, including an MLP (multilayer perceptron) model with 3 hidden layers on the MNIST dataset [211] (layer sizes are set to 784-256-256-256-10 as in [79]), ResNet-32 model [13] on the CIFAR-10 dataset [207], and ResNet-44 model [13] on the CIFAR-100 dataset [207]. We train 7 different neural network versions for each listed benchmark: a baseline model with no Dropout/Dropconnect, a pair of

models for the weight-stationary dataflow with Random and Hardware-Aware Dropconnect, a pair of models for the input-stationary dataflow with Random and Hardware-Aware Dropout at the layer inputs, and finally a pair of models for the output-stationary dataflow with Random and Hardware-Aware Dropout at the layer outputs. We utilize the same number of training epochs for all neural network versions on the same benchmark, namely, 100 epochs for the MLP and 200 epochs for the ResNet models.

The value representing the probability of being present in Dropout/Dropconnect, p, is set to the same value for all 6 neural network versions trained with Dropout/Dropconnect on the same benchmark. Namely, p is set to 0.5 for the MLP (MNIST), while being set to 0.9 for both the ResNet-32 (CIFAR-10), and the ResNet-44 (CIFAR-100) benchmarks. We assume a typical systolic array size of 16×16 and utilize the mapping schemes described in Section 10.3.2 while modeling the Hardware-Aware Dropout/Dropconnect behavior.

We implement the previous state-of-the-art techniques, fault-aware pruning and fault-aware pruning + training [79], by pruning neural network variables from the baseline model to model the impact of hardware bypassing operations and fine-tuning the pruned model for each underlying fault pattern for 10% of the epochs that are used in the initial training process. The analysis in [79] is proposed only for the weight-stationary dataflow, yet the generalized implementation of these methods to input-stationary and output-stationary dataflows is relatively straightforward.

A small subset of 320 training images (i.e., 5 mini-batches of 64 examples) is utilized in all benchmarks for calibrating batch normalization statistics and searching for benign pruning patterns through the diagonal shifting of the weight tensors. The calibration of batch normalization statistics is not necessary for the MLP model due to the absence of batch normalization layers. In addition, we avoid batch normalization layer calibration in the input-stationary and output-stationary dataflows because neural networks can accommodate a certain amount of variation at the layer inputs and outputs gracefully; thus, the benefits of batch normalization layer calibration are observed to be minimal.

Finally, we provide a detailed hardware analysis by implementing the bypass logic in the processing elements with different input (8-bit fixed-point, 32-bit floating-point) and dataflow types. The hardware area, power, and delay impact are characterized through logic synthesis experiments at the 1 GHz target frequency and with the 15 nm technology libraries.

10.5 Experimental Results

10.5.1 Resilience to Processing Element Bypassing

We evaluate the resilience of trained DNNs by measuring their accuracy under different processing element bypassing rates. The results are presented in Figures 10.5-10.7 for the weight-stationary, input-stationary, and output-stationary dataflows.

FP (Fault-Aware Pruning) [79] indicates the case where bypassing affects a baseline model with standard training. *FP+T (Fault-Aware Pruning + Training)* [79] adds on a fine-tuning process to the previous approach by individually training each pruned model (10% of the initial training epochs) to recover accuracy. It bears noting that the training step in *FP+T* needs to be performed for *each* faulty embedded accelerator device with a unique hardware defect pattern. We present up to 4 distinct results to showcase the effectiveness of the proposed techniques. First, a deep neural network model with *RDT (Random Dropout/Dropconnect Training)* is subjected to the bypassing effect of the faulty processing elements. A similar experiment is also repeated with *HDT (Hardware-Aware Dropout/Dropconnect Training)*. In the case of DNNs with batch normalization in the weight-stationary dataflow, we improve the result of *HDT* by calibrating batch normalization layers and report the improved accuracy with the label *HDT+BNC (Batch Normalization Calibration)*. Finally, the results obtained both for *HDT+BNC*, and for *HDT* in the absence of batch normalization calibration, are improved further by testing 16 different diagonally shifted versions of the weight tensors (for 16×16 systolic array) and identifying the shift amount with the highest accuracy. The highest accuracy is reported with the label *HDT+(BNC)+DS (Diagonal Shift)*.

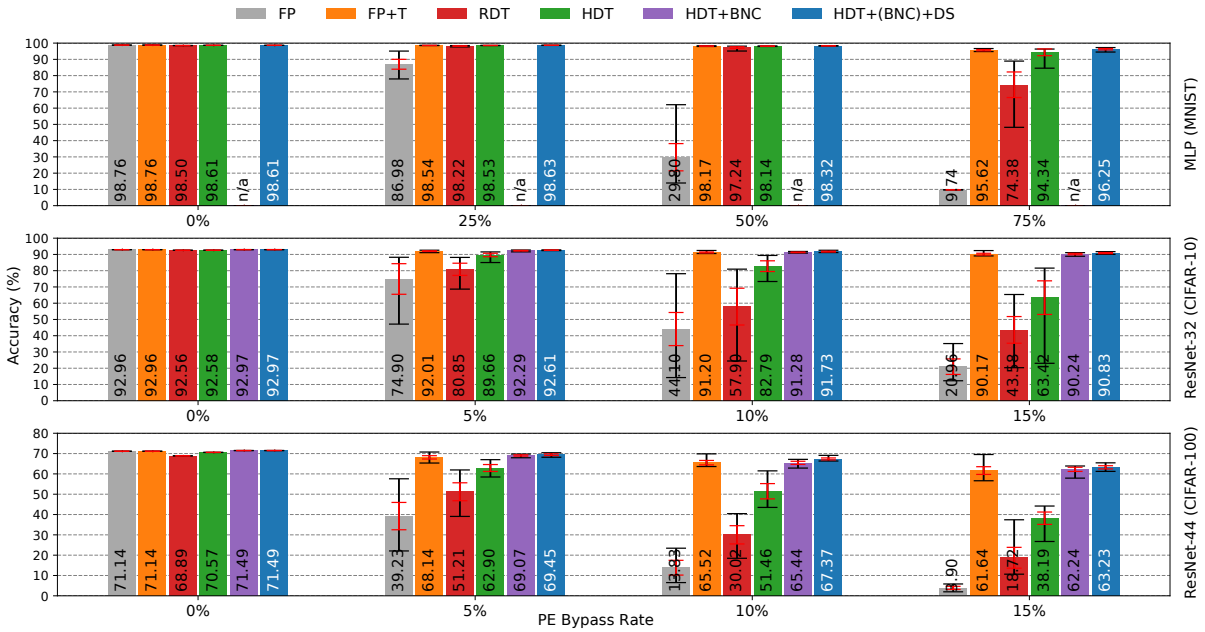

Figure 10.5. Resilience to bypass operations in weight-stationary dataflow.

The height of the bar charts and the annotation labels in Figures 10.5-10.7 indicate the mean accuracy values obtained through 10 independent trials with different hardware defect patterns. The observed minimum/maximum values in these trials and the 95% confidence interval for the mean are indicated with the black and red error bars, respectively.

Our results indicate that *FP (Fault-Aware Pruning)* implemented through bypassing could retain competitive accuracy only at low bypass rates. *FP* could result in significantly divergent accuracy values depending on the location of the impacted variables, particularly in the case of weight-stationary dataflow. The observed significant accuracy differences across the trials underscore the expected saliency variance among variables in the standard DNNs.

The training step in *FP+T (Fault-Aware Pruning + Training)* could be effective in restoring competitive accuracy levels after bypassing in weight-stationary architectures. However, the complete restoration of the original accuracy may not always be feasible since the pruning impact on the weights could exceed the levels that can be remedied entirely in a limited number of training iterations. The feasibility of this approach is further limited in practice since it requires a unique training process for *each* device with a unique defect pattern. For example, the

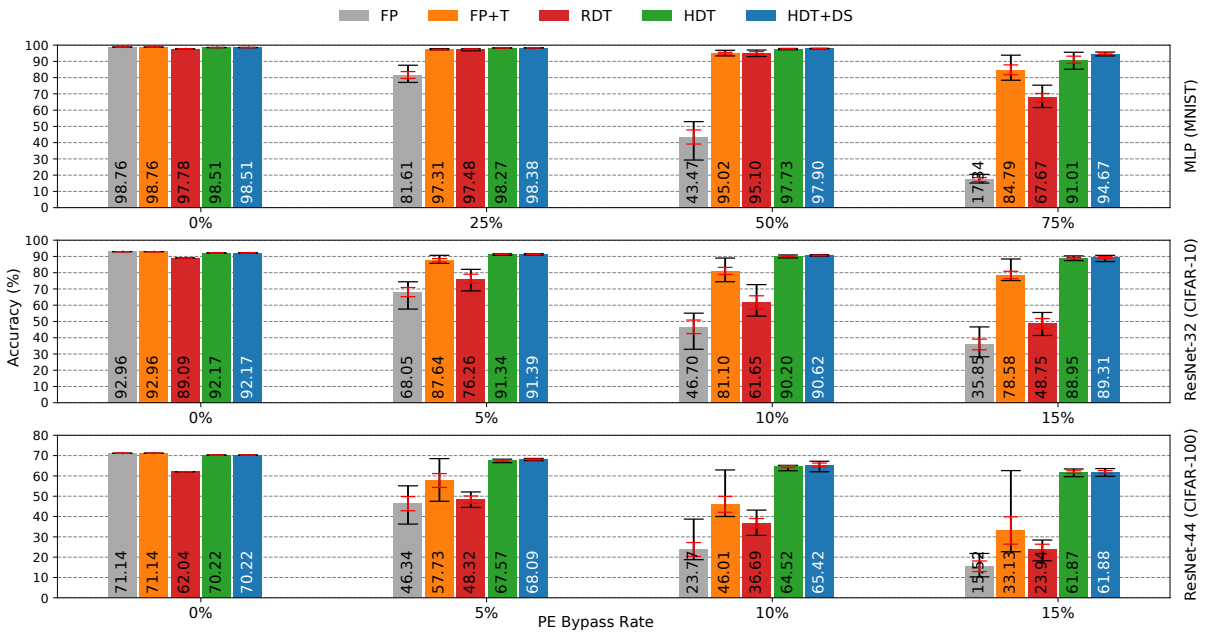

Figure 10.6. Resilience to bypass operations in input-stationary dataflow.

fine-tuning process with 10% of the initial training epochs for only 10 faulty accelerator devices will incur a computational cost that matches the initial training. As a result, the cost model of *FP+T* could quickly reach prohibitive levels in the case of cost-sensitive embedded devices.

The effectiveness of *FP+T* is limited in input-stationary and output-stationary architectures, resulting in accuracy being unable to be sufficiently restored with the fine-tuning budgets allocated for the weight-stationary designs. The difficulty of fine-tuning in input-stationary and output-stationary dataflows stems from the particularities of the hardware bypassing effect manifestation in these architectures. Training examples only with particular batch indices will be affected from a bypassed processing element (as detailed in Equations 10.4-10.5), and thus the number of effective fine-tuning epochs to compensate for the contribution of a bypassed unit diminishes by a factor that equals the batch size. The outlined problem can be alleviated by increasing the number of training epochs, yet this will exacerbate the training cost even further.

Training with *RDT (Random Dropout/Dropconnect Training)* improves model decentralization and delivers consistent and significant accuracy improvements (up to 67.44%) in the bypassed computational fabrics when compared to baseline models in *FP*. However, *RDT* could

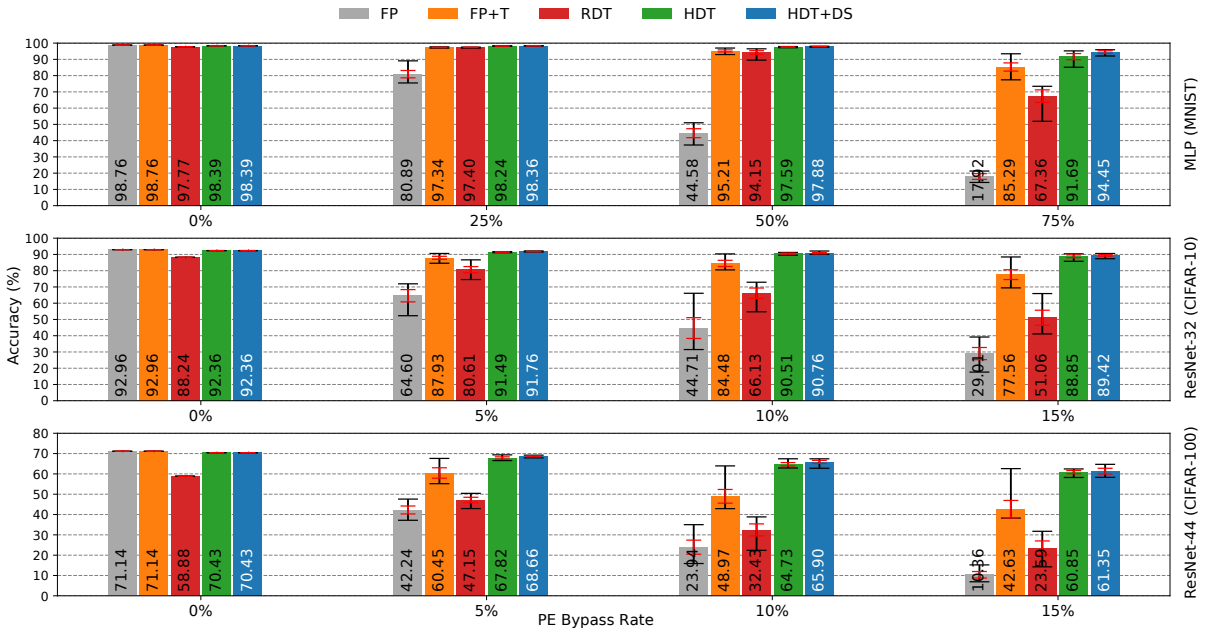

Figure 10.7. Resilience to bypass operations in output-stationary dataflow.

introduce a noticeable reduction in fault-free accuracy when the probability p of being present is not set sufficiently high in more challenging benchmarks such as ResNet-44 (CIFAR-100).

HDT (Hardware-Aware Dropout/Dropconnect Training) requires minimal additional information related to systolic array size and the utilized neural network mapping scheme, and it offers two fundamental advantages compared to *RDT*. First, by modeling the hardware bypassing effects more precisely in the training process, it provides a much more graceful accuracy degradation even when a large portion of the processing elements are bypassed due to hardware defects. DNNs trained with *HDT* suffice in delivering competitive accuracy values in the bypassed input-stationary and output-stationary fabrics without necessitating any additional measure. Second, the impact of *HDT* on error-free model accuracy is observed to be less noticeable than *RDT*. While dropping each variable independently in *RDT* minimizes co-adaptation [59], it could further negatively influence the error-free accuracy when the neural network capacity is limited in a given benchmark. The number of independent variables that control the drop patterns is limited in the case of *HDT* (i.e., as many as the number of processing elements), and the dropping behavior of the variables that map to the same processing element is perfectly

correlated. The group-wise control of the variables in the dropping process is observed to provide more expressiveness when compared to the case where the variables are individually controlled. As a result, the impact of *HDT* on error-free accuracy is observed to be often positive due to its regularization effect, and the accuracy loss does not exceed 0.92% even in the worst case. The minimal negative impact on the error-free accuracy loss could be eliminated by training a few neural networks with diverse values of p, and deploying the proper version of the model in accordance with the defect rate of the individual device to maximize accuracy.

An interesting phenomenon related to batch normalization layers is observed in the weight-stationary dataflow. In DNNs with batch normalization layers, the weight pruning as a result of hardware bypassing could alter layer output statistics and result in noticeable accuracy loss even for models with *HDT*. Nevertheless, the observed accuracy drop is often not caused by a permanent information loss and thus can be ameliorated through the cost-effective calibration of the batch normalization statistics. As a result, an accuracy improvement of up to 26.82% is observed in *HDT+BNC* results when compared to *HDT*. The outlined problem is primarily experienced for DNNs with batch normalization in the weight-stationary dataflow, and thus *HDT+BNC* results are reported only in these cases.

The accuracy values of *HDT*, or whenever applicable *HDT+BNC*, could be further improved (up to 3.66%) and the accuracy deviation between different fault manifestation patterns can be reduced in *HDT+(BNC)+DS* results by evaluating 16 diagonally shifted versions of the weight tensors and selecting the configuration with the highest accuracy.

Overall, a one-time training with Hardware-Aware Dropout/Dropconnect, coupled with on-device calibration techniques, could assure consistently high accuracy even at extreme bypass rates. The average accuracy loss does not exceed 0.61% in MLP (MNIST) at 50% bypass rate ($\sim$128 out of 256 PEs), 1.60% in ResNet-32 (CIFAR-10) at 10% bypass rate ($\sim 26/256$ PEs), and 2.13% in ResNet-44 (CIFAR-100) at 5% bypass rate ($\sim 13/256$ PEs). The proposed approach consistently outperforms the expensive *FP+T* by a significant margin without necessitating costly device-specific training.

Table 10.1. Area and power overheads of the bypass logic.

		Area Overhead (%)	Power Overhead (%)
Fixed-Point (8-bit)	**WS / IS**	7.087	5.357
	OS	4.003	2.273
Floating-Point (32-bit)	**WS / IS**	0.788	0.495
	OS	0.520	0.216

The resilience characteristics in Figures 10.5-10.7 could readily be translated into yield improvements by salvaging a larger number of defective DNN accelerators for commercial benefit, and when paired with periodic testing techniques, are expected to increase the operational lifetime of DNN hardware against aging effects by allowing higher defect accumulation counts.

10.5.2 Hardware Overhead Analysis

Table 10.1 outlines the area and power overheads of the bypass mechanisms. We present a single set of results for the input-stationary and weight-stationary architectures due to their identical processing element and bypass mechanism designs.

The largest area and power overheads hover around 7.09% and 5.36% in the case of 8-bit fixed-point weight/input-stationary processing elements. The bypassing overheads in output-stationary designs are up to 43.5% smaller in terms of area and 57.6% smaller in terms of power when compared to the weight/input-stationary dataflows as they can be implemented through AND gates instead of 2-to-1 multiplexers. The area and power overheads in the 32-bit floating-point processing elements are observed to be up to 9.0× and 10.8× smaller respectively when compared to the 8-bit fixed-point designs due to the larger footprint of floating-point units.

We further perform a timing analysis by measuring the longest path delay in the original and modified processing elements at 1 GHz target frequency. Our experimental results indicate that the bypass modifications have a negligible delay increase of up to 2.2% in the worst case.

The minimal timing impact is consistent with the expectations since the bypass logic introduces an additional 1-2 (2-input) gate delay due to the required 2-to-1 multiplexers in weight/input-stationary or (2-input) AND gates in output-stationary architectures. In summary, the hardware overhead results demonstrate that the bypass mechanisms can be embedded into the processing elements at minimal cost in practical designs.

10.6 Chapter Summary

The joint consideration of hardware micro-architecture and neural network characteristics unlocks novel opportunities to address reliability problems in deep learning accelerators. We tackle hardware defects and maintain neural network accuracy by circumventing the impact of faulty hardware components, boosting neural network decentralization proactively through one-time training with Hardware-Aware Dropout/Dropconnect, and complementing such design-time modifications with cost-effective on-device calibration and reconfiguration techniques. Overall, this chapter presents effective strategies and a practical design flow for boosting the reliability and operational lifetime of embedded deep learning accelerators in a proactive manner.

10.7 Acknowledgements

Chapter 10 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022 ([8]). The dissertation author was the primary investigator and author of this paper.

Chapter 11

Searching for Information Redundancy in DNNs

Taming the computational cost of DNNs (deep neural networks) has focused on first-order techniques, such as eliminating numerically insignificant neurons/filters through numerical contribution metric prioritizations, yielding passable improvements. Nevertheless, redundancy in DNNs extends well beyond the limits of numerical insignificance. Modern DNN layers exhibit a significant correlation among output activations; hence, the number of extracted orthogonal features at each layer rarely exceeds a small fraction of the layer size. The exploitation of this observation necessitates the quantification of information content at layer outputs. To this end, we employ practical data analysis techniques coupled with a novel feature elimination algorithm to identify a minimal set of computation units that capture the information content of the layer and squash the rest. Linear transformations on the subsequent layer ensure accuracy retention despite the removal of a significant portion of the computation units. The proposed approach in this chapter, in addition to delivering results overwhelmingly superior to hitherto promulgated heuristics, furthermore promises to spearhead the design of more compact deep learning models through an improved understanding of DNN redundancy.

11.1 Introduction

Structured pruning algorithms listed in Section 3.2.1 target all parameters in a neuron or convolution filter. The removal of an entire neuron or filter does not exacerbate irregular sparsity; therefore, the advantages can be reaped in any hardware platform with no demands on further memory compression or specialized hardware to skip computations. Nevertheless, structured pruning algorithms often lead to significant accuracy loss and necessitate onerous fine-tuning (re-training) steps to retrieve baseline accuracy levels. The sub-optimal decisions of a pruning algorithm can be minimized when it is applied iteratively with each timid pruning step immediately followed by re-training, yet this approach could incur significant costs in terms of time and computational resources. In contrast, a more comprehensive identification of DNN redundancy would enable precise redundancy removal methods, reduce the need for extensive fine-tuning steps, and thus facilitate the deployment of DNNs in resource-constrained applications at a minimal cost.

Structured pruning techniques in Section 3.2.1 rank the neurons or filters through various importance ranking heuristics and target the units with the smallest contribution to DNN decisions to minimize the impact on model accuracy. As conventional structured pruning algorithms analyze the importance of the computation units, their analysis is restricted to an individual unit and fails to consider relationships among the units. A computation unit might seem essential when it is independently evaluated (i.e., due to its output contribution), yet its role might prove redundant when assessed within a group of computation units.

Let us clarify this argument with a somewhat simplistic yet motivational illustration. A botanical researcher gathers various types of measurements (temperature, humidity, altitude, and vegetation density) to determine an ideal location for a plant species. After observing that the altitude data exhibits low variance, the researcher discards the altitude data since it hardly delivers any useful information in the comparison. A careful investigation of the data furthermore reveals that temperature readings are strongly correlated with vegetation density, with the possible

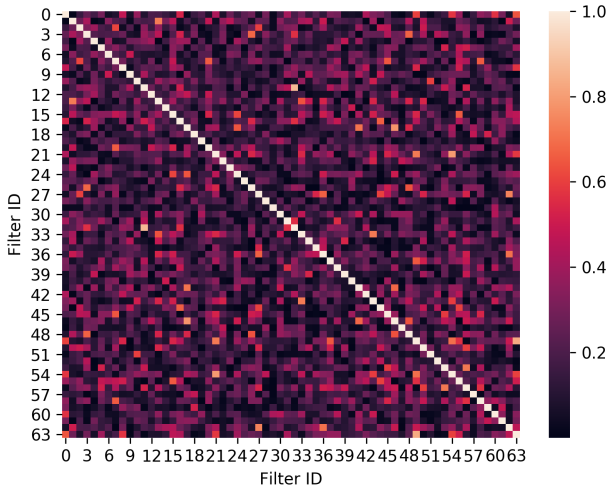

Figure 11.1. Pairwise output correlation magnitudes in a convolutional layer.

explanation of another non-measured variable, say the level of sunlight, affecting both. Although both variables are individually essential, the researcher ends up choosing only one of them since neither of the measurements provides significant additional information over the other. Analogously, if there are computation units with correlated outputs in a DNN layer, only a subset of the units should suffice to maintain the information content. The remaining units could be eliminated with minimal impact on accuracy if their magnitude contribution is properly expressed by the subset.

11.2 Information Redundancy in DNN Activations

While one could envision investigating the pairwise correlations among convolution filter outputs, as in Figure 11.1 for the first layer of VGG-16 [222] model trained on CIFAR-10 [207], relying solely on this approach fails to provide a complete picture of the redundancy since linear relationships can involve multiple computation units. We can characterize information redundancy more comprehensively by considering the computation unit output as a distinct vector for a set of predictions, and then identifying a minimal set of orthogonal base vectors that span the utilized layer output space.

Let us assume there are n_l computation units in layer l. We refer to the term *computation*

unit to signify a neuron in the fully connected layers or a filter in the convolutional layers. We consider the outputs of a fully connected layer (after the non-linear activation function) as a two-dimensional matrix $A_{n_l \times m}$ where m is the number of examples that the inference is performed on. We restrict our analysis to fully connected layer outputs for simplicity; however, a similar analysis also applies to convolutional layers if each output feature map channel is flattened along the temporal dimension, forming a two-dimensional matrix where each row is allocated to the outputs of a single channel. If there are linear dependencies between the rows of $A_{n_l \times m}$, the activation matrix could be reduced into a smaller matrix with n_l' orthogonal rows through linear transformations.

While n_l' can be easily determined through the *Gram-Schmidt orthogonalization* [223] process, a rigid elimination approach will deliver a minor reduction in the number of rows at best. The process will discard the row only if it exactly equals a linear combination of the other rows, yet such a condition is hard to satisfy for a matrix $A_{n_l \times m}$ when $m \gg n_l$ plagued by the presence of small numerical deviations.

A well-known data analysis tool, *PCA (Principal Component Analysis)* [206], seamlessly handles the effect of such inaccuracies. PCA expresses the input data with a smaller set of orthogonal variables that are known as principal components, which are ranked by the variance they explain in the data. As a preliminary analysis tool, we apply PCA to output activations and determine the minimum number of principal components that explain a certain amount of the output variance to assess the information redundancy in each layer.

Figure 11.2 presents the number of principal components that explain 95% of output variance in the layers of two DNNs: the LeNet-5 model trained on the MNIST dataset [211], and VGG-16 [222] trained on the CIFAR-10 dataset [207].[11] We observe that the rate of utilized output dimensions rarely exceeds 60% for LeNet-5 or 75% for VGG-16, excluding output layers. Moreover, certain convolutional and almost all fully connected layers are heavily under-

[11]The following convention is used to indicate layer types in Figure 11.2: C - convolutional layer, F - fully connected layer.

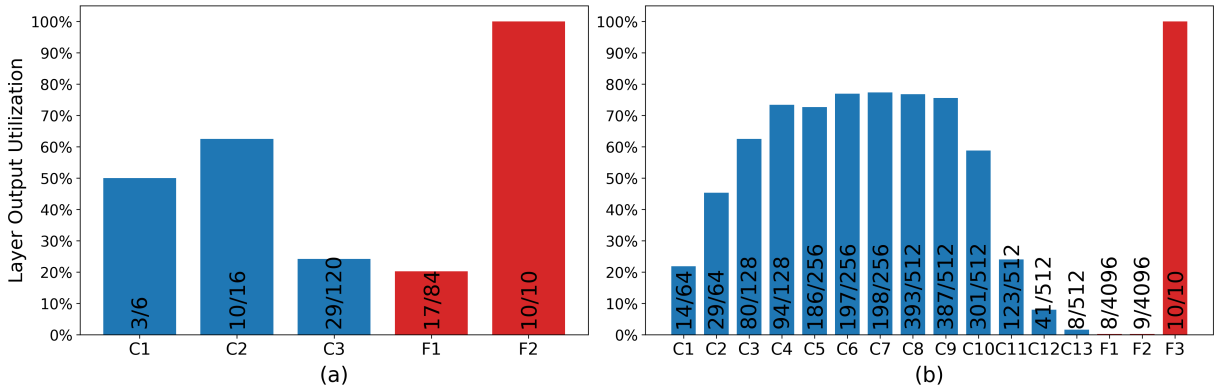

Figure 11.2. Layer output utilization in (a) LeNet-5 (on MNIST) and (b) VGG-16 (on CIFAR-10) architectures.

utilized. For instance, two hidden fully-connected layers in VGG-16 of 4096 neurons each end up having 95% of their output variance approximated by a rather minuscule number of principal components, namely 8 in one and 9 in the other. The utilization rates indicate an information redundancy; thus, the proper layer transformations could drastically reduce the model size and the inference costs with minimal impact on accuracy.

11.3 Squeezing DNN Correlations with Feature Elimination

11.3.1 Method Description

Principal component analysis extracts a set of useful features from a set of observed variables. Nevertheless, the generated principal components are not directly useful for the computation unit elimination problem since the principal components can not be directly affiliated with a single computation unit. We design a novel algorithm to address this issue. We first construct an approximated orthogonal base for the layer output space by following a procedure closely related to Gram-Schmidt orthogonalization. Second, we utilize the orthogonal base to determine a minimal subset of computation units that can span the utilized output space. We conclude with the elimination of the remaining units and form a transformation matrix that will properly update the weights of the subsequent layer so that the magnitude contribution of the eliminated units can be retained.

We have already designated the activations produced by layer l as a matrix $A_{n_l \times m}$. We denote a particular row of A as $\vec{A}_i$ where $1 \leq i \leq n_l$. Let us define a few more quantities before proceeding into the details of the algorithm. The variance of each row could be calculated as follows:

$$\sigma^2{}_{(\vec{A}_i)} := \frac{1}{m} \sum_{j=1}^{m} (A_{i,j} - \mu_{(\vec{A}_i)})^2 \tag{11.1}$$

In (11.1), σ^2 denotes the variance and μ represents the mean value of the row. The average row variance of $A_{n_l \times m}$ can be calculated through (11.2):

$$\sigma^2{}_{avg} := \frac{1}{n_l} \sum_{i=1}^{n_l} \sigma^2{}_{(\vec{A}_i)} \tag{11.2}$$

We aim to find an approximated orthogonal base $U_{n_l' \times m} := [\vec{U}_1, \vec{U}_2, ..., \vec{U}_{n_l'}]$ for $A_{n_l \times m} := [\vec{A}_1, \vec{A}_2, ..., \vec{A}_{n_l}]$. A modified version of the Gram-Schmidt orthogonalization can be utilized for this purpose as detailed subsequently.

As a first step, we check to see if the first row of the original activation matrix contains sufficient variance through the condition in (11.3) where δ is a non-negative experimental tuning coefficient that controls the approximation level in the Gram-Schmidt orthogonalization process:

$$\delta \cdot \sigma^2{}_{avg} \leq \sigma^2{}_{(\vec{A}_1)} \tag{11.3}$$

If (11.3) is satisfied, we set $\vec{A}_1$ as the first vector in the orthogonal base (11.4):

$$\vec{U}_1 := \vec{A}_1 \tag{11.4}$$

In the second step, we need to extract the sub-component of $\vec{A}_2$ that is orthogonal to $\vec{U}_1$. We first project $\vec{A}_2$ onto $\vec{U}_1$, then subtract the projection from $\vec{A}_2$ to obtain the orthogonal component $\vec{R}$:

$$\vec{R} := \vec{A}_2 - proj_{(\vec{U}_1)}(\vec{A}_2) = \vec{A}_2 - \frac{(\vec{A}_2 \cdot \vec{U}_1)}{(\vec{U}_1 \cdot \vec{U}_1)}\vec{U}_1 \tag{11.5}$$

If $\vec{R}$ contains sufficient variance (11.6), we include it in the orthogonal base (11.7):

$$\delta \cdot {\sigma^2}_{avg} \leq {\sigma^2}_{\vec{R}} \tag{11.6}$$

$$\vec{U}_2 := \vec{R} \tag{11.7}$$

Similar steps could be carried with the remaining rows of A by calculating the sub-component of the current row that is orthogonal to the base vectors obtained so far (11.8), and finally including the remaining vector $\vec{R}$ in the orthogonal base (11.9) if it still meets the constraints of sufficient variance expressed in Eqn. (11.6). In Eqn. (11.8), each projected component can be subtracted from $\vec{A}_i$ independently since the projected components are also orthogonal.

$$\vec{R} := \vec{A}_i - \sum_{k=1}^{size(U)} proj_{(\vec{U}_k)}(\vec{A}_i) = \vec{A}_i - \sum_{k=1}^{size(U)} \frac{(\vec{A}_i \cdot \vec{U}_k)}{(\vec{U}_k \cdot \vec{U}_k)}\vec{U}_k \tag{11.8}$$

$$\vec{U}_{size(U)+1} := \vec{R} \tag{11.9}$$

It should be noted that the proposed algorithm behaves identically to standard Gram-Schmidt orthogonalization when $\delta = 0$, and the level of approximation increases with the increasing values of δ.

Through the steps outlined, we construct an approximate orthogonal base $U_{n'_l \times m} := [\vec{U}_1, \vec{U}_2, ..., \vec{U}_{n'_l}]$ where $n'_l \leq n_l$. Since U is an approximate orthogonal base for A, a linear transformation can construct the approximated version of A from U as in (11.10):

$$A_{n_l \times m} \approx C_{n_l \times n'_l} \times U_{n'_l \times m} \tag{11.10}$$

We refer to matrix C as the *composition matrix*. The composition matrix can be calculated by multiplying both sides of (11.10) with the right inverse of $U_{n'_l \times m}$:

$$A_{n_l \times m} \times (U_{n'_l \times m})^{-1} \approx C_{n_l \times n'_l} \times U_{n'_l \times m} \times (U_{n'_l \times m})^{-1} \tag{11.11}$$

$$C_{n_l \times n'_l} :\approx A_{n_l \times m} \times (U_{n'_l \times m})^{-1} \tag{11.12}$$

Since the rows of $U_{n'_l \times m}$ are orthogonal, $U_{n'_l \times m}$ is easily right-invertible by merely taking the transpose of $U_{n'_l \times m}$ and normalizing each column with the square of its magnitude. $U_{n'_l \times m}$ and $C_{n_l \times n'_l}$ share similarities with the Q and R matrices in *QR factorization* [223], yet the process subtly differs in that the described factorization is approximate, and the base vectors in $U_{n'_l \times m}$ are not normalized.

As a next step, we define another matrix $\tilde{A}_{n'_l \times m}$ that consists of the subset of the rows of the original activation matrix $A_{n_l \times m}$ that deliver a distinct vector to $U_{n'_l \times m}$ after the projections (i.e., the rows in which (11.6) holds). The reduced form of (11.10) holds for $\tilde{A}_{n'_l \times m}$ if the row indices selected to construct $\tilde{A}_{n'_l \times m}$ are also chosen in the composition matrix to form the reduced composition matrix $\tilde{C}_{n'_l \times n'_l}$:

$$\tilde{A}_{n'_l \times m} \approx \tilde{C}_{n'_l \times n'_l} \times U_{n'_l \times m} \tag{11.13}$$

We extract $U_{n'_l \times m}$ by left-multiplying both sides with $(\tilde{C}_{n'_l \times n'_l})^{-1}$:

$$(\tilde{C}_{n'_l \times n'_l})^{-1} \times \tilde{A}_{n'_l \times m} \approx (\tilde{C}_{n'_l \times n'_l})^{-1} \times \tilde{C}_{n'_l \times n'_l} \times U_{n'_l \times m} \tag{11.14}$$

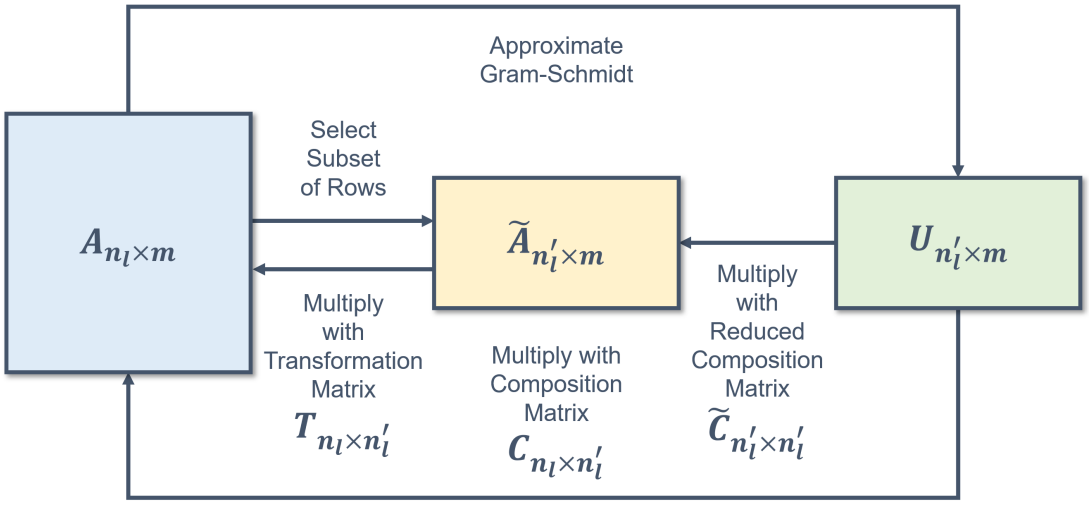

Figure 11.3. Relationships among the derived matrices.

$$U_{n_l' \times m} \approx (\tilde{C}_{n_l' \times n_l'})^{-1} \times \tilde{A}_{n_l' \times m} \tag{11.15}$$

Being a triangular matrix with no zero diagonal entries due to the orthogonalization process, $\tilde{C}_{n_l' \times n_l'}$ is always invertible. We embed $U_{n_l' \times m}$ in (11.10) to obtain:

$$A_{n_l \times m} \approx C_{n_l \times n_l'} \times (\tilde{C}_{n_l' \times n_l'})^{-1} \times \tilde{A}_{n_l' \times m} \tag{11.16}$$

We refer to $T_{n_l \times n_l'} := C_{n_l \times n_l'} \times (\tilde{C}_{n_l' \times n_l'})^{-1}$ as the *transformation matrix*, thus simplifying (11.16) to (11.17):

$$A_{n_l \times m} \approx T_{n_l \times n_l'} \times \tilde{A}_{n_l' \times m} \tag{11.17}$$

We have so far formulated multiple different matrices in our derivation. As a summary, we have visualized the relationships among the derived matrices in Figure 11.3.

$T_{n_l \times n_l'}$ captures an information essential to our algorithm. It enables us to produce $\tilde{A}_{n_l' \times m}$ instead of $A_{n_l \times m}$ in the current layer by eliminating $n_l - n_l'$ computation units. We can still approximately construct $A_{n_l \times m}$ by multiplying with the transformation matrix $T_{n_l \times n_l'}$. Although it

seems to create an additional computation stage at first glance, the matrix $T_{n_l \times n'_l}$ can be seamlessly merged with the subsequent layer thus further reducing the memory and computational costs of the subsequent layer since multiplication with the transformation matrix and the subsequent layer operations are essentially two subsequent matrix-multiplication steps. In detail, the next layer has a weight matrix $W_{n_{l+1} \times n_l}$ and biases $B_{n_{l+1} \times 1}$. It accumulates pre-activations through (11.18) where the biases are broadcasted into rows:

$$Z_{n_{l+1} \times m} := (W_{n_{l+1} \times n_l} \times A_{n_l \times m}) + B_{n_{l+1} \times 1} \tag{11.18}$$

We can embed (11.17) into (11.18) to obtain:

$$Z_{n_{l+1} \times m} \approx (W_{n_{l+1} \times n_l} \times T_{n_l \times n'_l} \times \tilde{A}_{n'_l \times m}) + B_{n_{l+1} \times 1} \tag{11.19}$$

As a final step, we multiply $W_{n_{l+1} \times n_l}$ and $T_{n_l \times n'_l}$ to obtain a reduced weight matrix $\tilde{W}_{n_{l+1} \times n'_l}$ that operates on the remaining computation unit outputs (11.20):

$$Z_{n_{l+1} \times m} \approx \tilde{W}_{n_{l+1} \times n'_l} \times \tilde{A}_{n'_l \times m} + B_{n_{l+1} \times 1} \tag{11.20}$$

As a result of these design-time modifications, the subsequent layer accumulates approximately close partial sum values even though the majority of the computation units have been eliminated in the current layer. It reduces the number of parameters and multiply-accumulate operations by a factor of n'_l / n_l in both layers and translates into remarkable memory footprint and performance improvements when this optimization is applied to all layers throughout the network. As a summary, we visualize the weight matrix update process in Figure 11.4.

The process could be naturally extended to convolutional layers as well. First, the spatial dimensions of the 4-dimensional output feature map ($A_{n_l \times s \times s \times m}$) need to be flattened along the temporal dimension to form a 2-dimensional input activation matrix ($A_{n_l \times s \cdot s \cdot m}$) where each row contains the outputs of a single channel, and s is the spatial dimension size of the feature

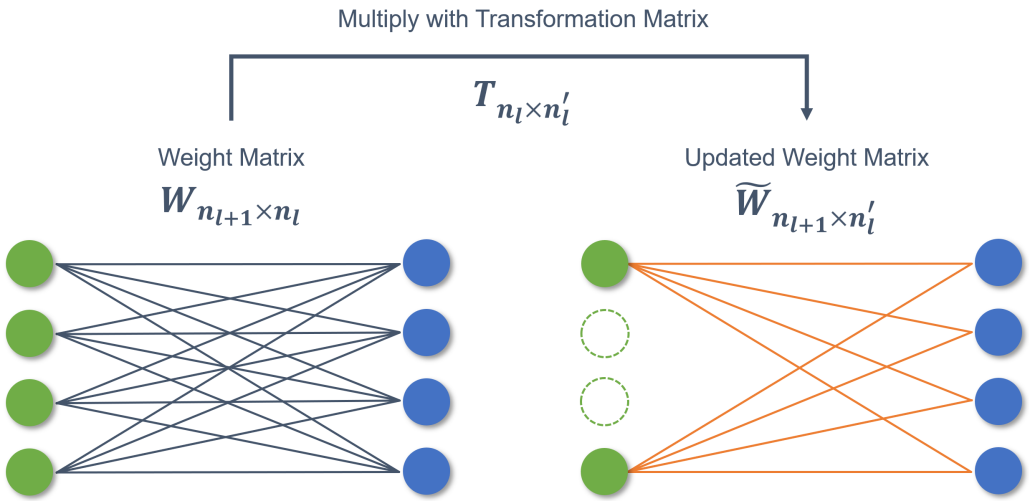

Figure 11.4. Weight matrix update process in the subsequent layer.

map. The described steps could be carried out similarly to extract the transformation matrix (T). Finally, the modifications on the subsequent convolutional layer could be performed by first flattening the 4-dimensional weight tensor into two dimensions (e.g., from $W_{n_{l+1} \times r \times r \times n_l}$ to $W_{n_{l+1} \cdot r \cdot r \times n_l}$ where r is the spatial dimension size of the filter) so that all dimensions except the input channel dimension are merged[12], multiplying with the transformation matrix, and then un-flattening the resulting weight matrix rows back into the original dimensions to form the updated weight tensor ($W_{n_{l+1} \times r \times r \times n_{l'}}$).

11.3.2 Practical Analysis of the Algorithmic Complexity

The initial stage of our algorithm consists of the Gram-Schmidt orthogonalization process, which scales linearly with m and quadratically with the layer size (n_l), thus incurring total complexity of $O(m \times {n_l}^2)$. The derivation of the composition matrix ($C_{n_l \times n'_l}$) through the iterative matrix multiplication algorithm also requires $O(m \times {n_l}^2)$ complexity since n'_l is bounded by n_l in the worst case. The derivation of the inverse-reduced composition matrix $(\tilde{C}_{n'_l \times n'_l})^{-1}$ (with *Gauss-Jordan elimination* [223]) and the derivation of the transformation matrix (with iterative matrix multiplication) both incur $O({n_l}^3)$ complexity. Finally, the update of the next layer's

[12]The entries of each $r \times r$ weight tensor slice are stored in the subsequent locations in the flattened format.

weight matrix through the iterative matrix multiplication requires $O(n_{l+1} \times {n_l}^2)$ complexity. In total, the run-time complexity scales in the order of $O((m \times {n_l}^2) + {n_l}^3 + (n_{l+1} \times {n_l}^2))$ in theory and frequently dominated by the first term in practice as m could be significantly large in certain scenarios. The number of columns in the profiled activation matrix (m) depends on both the number of predictions (linearly), as well as on the spatial dimensions of the output feature maps (quadratically) due to the flattening process. The feature map channels are flattened before the Gram-Schmidt process; therefore, an $s \times s$ feature map channel incurs s^2 columns for a single prediction in the profiled activation matrix ($A_{n_l \times m}$).

The memory requirements depend on the size of the intermediate matrices used in the calculations and thus scale in proportion with $O((m \times n_l) + {n_l}^2 + (n_{l+1} \times n_l))$. The memory complexity is largely dominated by the first term due to the large size of m in the case of convolutions.

11.3.3 Relationship with Low-Rank Tensor Decomposition

The fundamental operating principles of the algorithm share similarities with low-rank tensor decomposition techniques [161, 162, 163]. Let us explain this relationship on the fully connected layer whose pre-activations are accumulated through (11.21):[13]

$$Z_{n_l \times m} := W_{n_l \times n_{l-1}} \times A_{n_{l-1} \times m} + B_{n_l \times 1} \tag{11.21}$$

Tensor decomposition expresses the layer weight matrix as a product of multiple lower-rank matrices. Girshick [161] decomposes the fully connected layer operation into two sequential matrix multiplications through *SVD (singular value decomposition)* [223] as follows:[14]

$$Z_{n_l \times m} = U_{n_l \times n_{l''}} \times (S_{n_{l''} \times n_{l''}} (V_{n_{l-1} \times n_{l''}})^T \times A_{n_{l-1} \times m}) + B_{n_l \times 1} \tag{11.22}$$

[13] The formulation in (11.21) for the current layer resembles (11.18) that is constructed for the subsequent layer.
[14] S and V^T are statically multiplied to form weight matrix SV^T.

If the number of utilized singular values ($n_{l''}$) is relatively small, tensor decomposition reduces the parameters and multiply-accumulate operations by a factor of $(n_{l''} \times (n_l + n_{l-1}))/(n_l \times n_{l-1})$ in the target layer. Neurons in a fully connected layer extract features from the input data, yet the derived output features may overlap, resulting in feature correlation and consequent fair amount of redundant computations. Tensor decomposition allows a fully connected layer to extract only a small set of orthogonal features from the data (since SV^T is row-orthogonal). Then, each neuron forms its pre-activations by taking a linear mixture of the extracted features through the second matrix multiplication. Despite the benefits, the technique may introduce additional latency because a single matrix multiplication is converted into two sequential ones.

Although $Z_{n_l \times m}$ is constructed through a pre-determined linear combination of $n_{l''}$ orthogonal features, such information can not be directly utilized for elimination of the linearly dependent rows because $A_{n_l \times m}$ is generated by processing $Z_{n_l \times m}$ with a non-linear function in the following step. Non-linear activation functions (i.e., tanh or ReLU) interfere with the dependencies among pre-activation rows, yet certain linear properties are still preserved due to the local (e.g., tanh) or piece-wise (e.g., ReLU) linear nature of these functions.

Our approach differs from the low-rank tensor decomposition in the following aspects: First, we analyze the rows of $A_{n_l \times m}$ rather than the layer weights to account for the effect of the non-linearity. Second, we carry out a feature elimination procedure rather than feature extraction to be able to prune the computation units directly. As the magnitude contribution of the original layer still needs to be constructed, we avoid any depth expansion by embedding this operation into the next layer through weight updates. The procedure on the convolutional layers shares the outlined similarities and differences with the tensor decomposition method presented in [163].

11.4 Experimental Method

We utilize three different DNN models with three distinct datasets for the experiments: LeNet-5 on the MNIST dataset [211], VGG-16 [222] on the CIFAR-10 dataset [207], and

ResNet-50 [13] on the ImageNet dataset [224]. Keras [202] with the TensorFlow [203] backend is used to design the experiments. We train the LeNet-5 and VGG-16 base models from scratch and use the pre-trained weights for ResNet-50. LeNet-5 and VGG-16 training is performed with the SGD (Stochastic Gradient Descent) optimizer at a learning rate of 10^{-3} for 100 epochs. Model weights are initialized with the Glorot uniform initializer [225], and biases initialized to zeros. We utilize SCALE-Sim [62] to characterize the performance gains on a DNN accelerator model similar to Eyeriss [166] through cycle accurate simulations. We conduct our experiments on a moderate power desktop system with Intel i5-8600K (6 core) CPU (central processing unit), 32GB of memory, and NVIDIA GTX1060 (6GB) GPU.

The algorithm is tested with various algorithmic coefficients (δ) to measure the accuracy drop after one-shot elimination. The increasing values of δ result in more aggressive pruning levels. The algorithmic coefficient further allows us to match accuracy and footprint graphs in Section 11.5. The same pruning coefficient (δ) is used for all layers in LeNet-5 and VGG-16. ResNet-50 has four convolution layer groups where each layer contains the same number of filters in a group. Scaling the δ coefficients by a constant for each group (i.e., $\vec{k} = [1, 1.5, 2, 2.5]$) helps to eliminate more filters from the layers with a large number of filters and maintains the accuracy better for less redundant networks. In addition, we omit the pruning of the ResNet-50 layers that merge with the residual connections to avoid any dimension mismatch. We note that the number of profiled examples (m) could be kept much smaller than the training set size with no adverse effect on precision. For instance, while the entire training set is profiled in the LeNet-5 experiments, we have profiled only 1000 training examples for VGG-16 and 500 training examples for ResNet-50 to overcome computational complexity and memory bottlenecks. We have also tried to down-sample the feature maps prior to flattening, yet forwent its adoption due to a noticeable accuracy loss.

To ensure a fair comparison, we prune the same number of computation units at each layer by utilizing the previously suggested pruning heuristics in the literature, namely, ranking the neurons and convolution filters with L1 and L2 norms (Li *et al.* [135]), output statistics such as

APoZ[15] (Hu *et al.* [131]) and the standard deviation of the activations (described in Molchanov *et al.* [134]). We include the results of random pruning and the LeNet-5 and VGG-16 models trained from scratch (reduced network, initialized with [225]) for comparison.[16] We report the accuracy under various scenarios. First, we measure the accuracy after one-shot elimination without any fine-tuning. We believe this is the most transparent metric to analyze the precision of an elimination algorithm since the insufficiency of the heuristic can be easily disguised through extensive fine-tuning.[17] Second, we fine-tune the network with minimal training for 5 epochs for LeNet-5 and VGG-16, and 0.1 epoch for ResNet-50, and then measure the accuracy. Third, we train the network more comprehensively until there is no improvement in validation accuracy for LeNet-5 and VGG-16, and 1 epoch[18] for ResNet-50 before reporting the accuracy. We repeat the experiments 10 times for LeNet-5, 5 times for VGG-16, and 1 time for ResNet-50 due to run-time constraints, and report the corresponding average values. We reduce the learning rate down to 10^{-6} during fine-tuning if necessary.

Our method may require low learning rates in fine-tuning because of the effect of the weight transformations on the gradients. We report the number of parameters, inference MAC (multiply-accumulate) operations, and execution (clock) cycles spent on the accelerator.

11.5 Experimental Results

The experimental results validate that our approach can dramatically shrink the DNNs with minimal impact on accuracy. Linear layer transformations allow accuracy retention even under extreme pruning scenarios, resulting in an extreme accuracy gap between our method and the other compared heuristics, particularly before any fine-tuning. Figure 11.5 and Fig-

[15]Evaluated only on VGG-16 and ResNet-50 with ReLU activation function.

[16]We have also tried a gradient-based method outlined in [134]. Although the local approximation obtained through the gradients is helpful in iterative pruning, it fails to produce satisfactory results of note when a significant portion of the units is pruned in one shot. We have, therefore, excluded the results for the gradient-based approach.

[17]Unfortunately, this metric is rarely reported in prior literature.

[18]It requires $\sim$ 15 hours of computation time for ResNet-50.

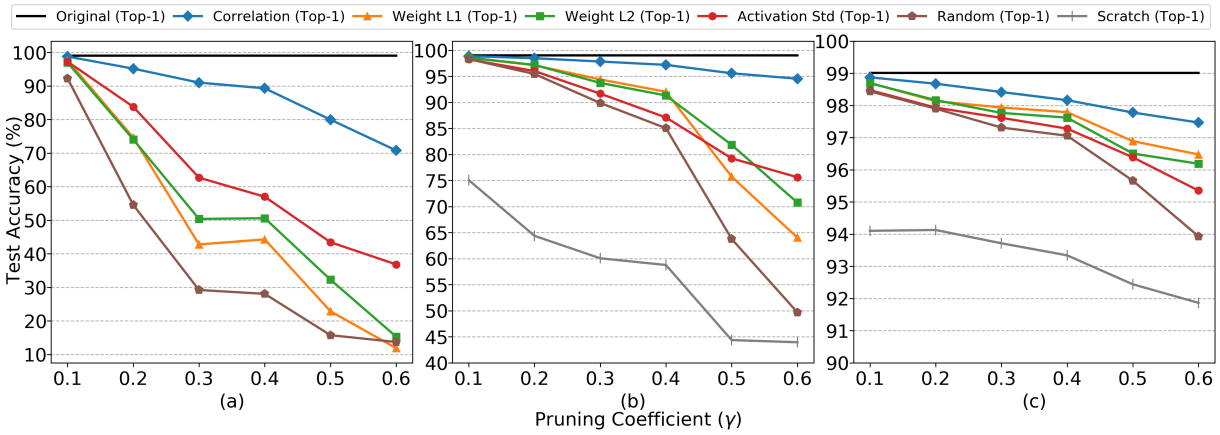

Figure 11.5. LeNet-5 accuracy drop (a) after one-shot pruning (b) after pruning + fine-tuning (5 epochs) (c) after pruning + fine-tuning (until no improvement).

ure 11.7(a)[19] demonstrate that we can eliminate 70.0% of the LeNet-5 parameters in one-shot, speed up the network by 2.2× on Eyeriss, and cause only a 3.9% accuracy loss without any fine-tuning. We further reduce the number of network parameters by 87.0% at around 9.7% accuracy loss with no fine-tuning while executing the network 3.7× faster. The best among the compared heuristics (*Activation Std.*) results in more than 42.0% accuracy loss at the same point. The highest accuracy difference between our method and the best of the heuristics even jumps to as large as 36.5% before any fine-tuning at certain pruning rates. The network pruned with our algorithm largely eradicates the accuracy loss after only a few epochs of fine-tuning (from 9.7% to 0.8%), and outperforms the competing heuristics in accuracy with only one fifth the amount of training when the networks are fine-tuned until accuracy improvement stops.

Figure 11.6 and Figure 11.7(b) demonstrate the performance of our method on VGG-16. We eliminate 79.7% of the parameters in one pruning step, delivering a 3.4× speedup with only a 1.8% accuracy loss and no fine-tuning. The comparison heuristics impair VGG-16 entirely even at lower pruning rates by causing the network to be stuck at a constant prediction while our accuracy loss is as small as 0.6% at the same pruning rate with no fine-tuning (79.4% accuracy gap). Fine-tuning delivers marginal benefits for all methods in VGG-16, and the networks pruned

[19]Figure 11.7 demonstrates remaining network parameters, required MAC operations to perform inference, and execution (clock) cycles spent on the accelerator.

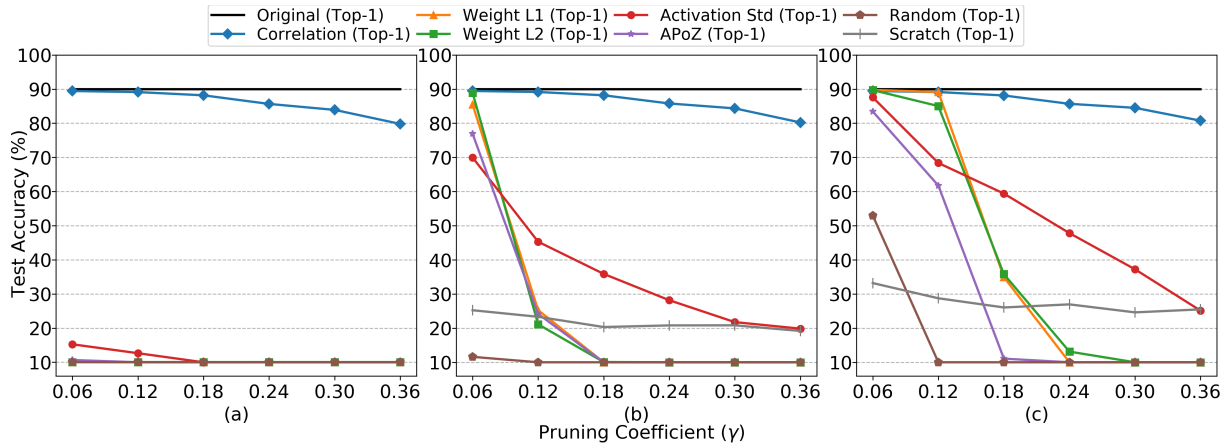

Figure 11.6. VGG-16 accuracy drop (a) after one-shot pruning (b) after pruning + fine-tuning (5 epochs) (c) after pruning + fine-tuning (until no improvement).

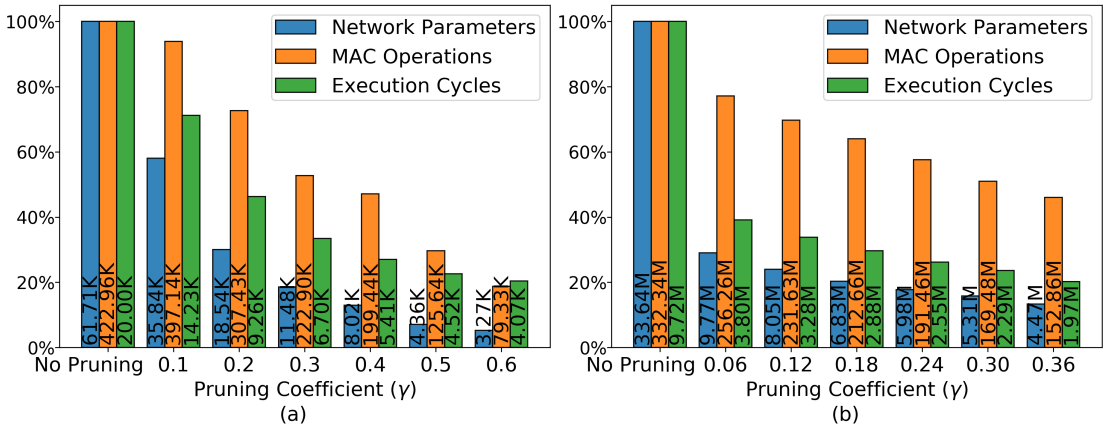

Figure 11.7. LeNet-5 (a) and VGG16 (b) hardware footprint after elimination.

with some of the other techniques can not even be re-trained after they have been completely destroyed in the pruning stage. We observe a 55.3% accuracy gap between our method and the closest heuristic even after the comprehensive re-training stage in the most aggressive pruning case, as we eliminate 86.7% of the parameters and make the network 4.9× faster.

Our method further performs remarkably well on ResNet-50, as seen on Figure 11.8 and Figure 11.9 even though model inherent redundancy is quite a bit lower relative to the previous examples. The relative absence of redundancy challenges the removal of computation units and accuracy retention with no fine-tuning. Yet we can still prune 21.3% of the parameters and reduce prediction times by 22.4% at the cost of only 2.9% top-5 accuracy loss with no

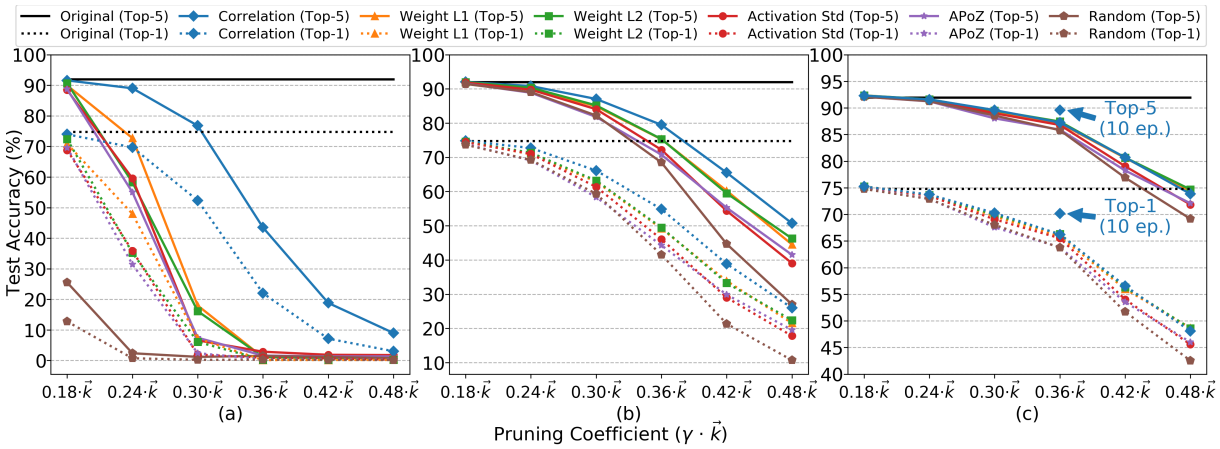

Figure 11.8. ResNet-50 accuracy drop (a) after one-shot pruning (b) after pruning + fine-tuning (0.1 epoch) (c) after pruning + fine-tuning (1 epoch).

fine-tuning. The closest method, with a top-5 accuracy comparable to the top-1 accuracy of the model pruned with our technique, diminishes top-5 accuracy by more than 19.2% at the same point. The top-5 accuracy difference between our technique and the closest heuristic surpasses 58.8% when no fine-tuning is applied, clearly demonstrating the superiority of our technique in accuracy retention. When complemented with fine-tuning our approach reduces the number of ResNet-50 parameters by 59.5% and multiply-accumulate operations by 52.7%; thus performance is boosted 2.14× with only 12.5%, 4.7%, and 2.3% top-5 accuracy drop after 0.1, 1, and 10 epoch(s) of fine-tuning, respectively. More fine-tuning epochs are expected to reduce the difference between the original and the pruned network further.

While fine-tuning does not significantly improve VGG-16 accuracy, reduced ResNet-50 models notably benefit from fine-tuning. It is relatively challenging to fine-tune the plain networks without residual connections (e.g., VGG-16) as pruning creates information bottlenecks in certain layers. Residual connections in ResNet-50 facilitate fine-tuning by providing an efficient flow for the gradients. As a result, despite their initial accuracy, the networks could reach similar accuracy values after sufficient fine-tuning. A good elimination heuristic still provides a head start and allows the network to converge to final accuracy values with fewer iterations.

The execution time of the proposed algorithm fades in practice. It needs to be applied

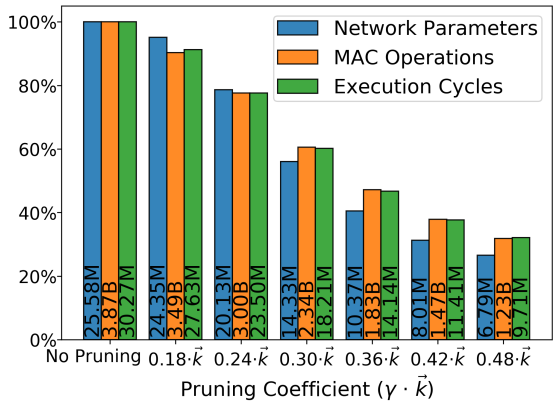

Figure 11.9. ResNet-50 hardware footprint after elimination.

once, and takes $\sim$ 20 minutes for ResNet-50 in the most challenging case, which is a small fraction of the time required by a single epoch of fine-tuning (i.e., $\sim$ 15 hours).

11.6 Chapter Summary

The number of extracted unique features rarely exceeds a tiny fraction of the layer size in modern DNN models; therefore, the information content of the layer could be represented by employing only a small number of computation units. This chapter presents a novel procedure to carry out feature elimination and perform successive modifications on the target model to compensate for the magnitude contribution of the eliminated units. We demonstrate the effectiveness of our approach as a powerful neuron/filter pruning technique that delivers results superior to the prior heuristics, yet requires only a minimal amount of fine-tuning. More importantly, we offer a principled way to understand and measure the redundancy in DNN computations, exposing an entirely new paradigm for the design of resource-efficient deep learning models for edge applications.

11.7 Acknowledgements

Chapter 11 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, “Squeezing Correlated Neurons for Resource-Efficient Deep Neural Networks,” in

Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD) - Part II, pp. 52–68, Springer, 2021 ([9]). The dissertation author was the primary investigator and author of this paper.

Chapter 12

Synergistic Co-design of Sparse DNNs and Hardware Accelerators

We have outlined a novel approach in Chapter 11 for comprehensive redundancy identification and structured neuron/filter elimination in deep neural networks. Alternatively, the redundancy extraction could be performed at a much finer-grained level in deep neural networks by pruning individual parameters to form sparsity patterns of unstructured nature. Unstructured sparsity patterns can deliver significantly higher model compression rates when compared to their structured counterparts, yet the hardware challenges posed by the irregular nature of unstructured sparsity are yet to be fully overcome in the existing hardware platforms. As algorithmic and hardware innovations individually deliver limited benefits, a synergistic approach is necessary to unleash the potential of sparse deep neural networks. This chapter presents a tightly-integrated design methodology for the sparsity patterns and hardware platforms to facilitate efficient hardware processing of fine-grained sparsity in deep neural networks. We demonstrate herein that novel complementary sparsity patterns can offer utmost expressiveness levels with inherent hardware exploitable regularity. Our novel dynamic training method converts the expressiveness of such sparsity configurations into highly accurate and compact sparse neural networks. Complementary sparsity is represented in a dense format, and when synergistically coupled with minimal yet strategic hardware engagements, can be processed in close concordance with the conventional dataflow of the dense matrix operations. This chapter thus demonstrates that there is ample room

for innovation beyond conventional techniques and immense practical potential for sparse neural networks through the synergistic design of sparsity patterns and hardware architectures.

12.1 Introduction

Unstructured sparsity patterns outlined in Section 3.2.1 yield significant reductions in the size and the computational complexity of deep neural networks; however, they fall short of delivering the anticipated computational benefits in the absence of specialized hardware support. Unstructured sparsity necessitates compressed storage formats to alleviate memory footprint and specialized operations to skip ineffectual computations involving zero-valued variables. Significant efforts thus have already been made in the recent past to construct hardware accelerators for sparse DNN inference as some examples are demonstrated in Section 3.2.2.

The deployment of sparse neural networks is still hindered by various fundamental challenges in practice. First, compressed storage formats such as run-length encoding or CSR representation [26] incur overheads that get further accentuated when the sparsity rate is limited. Second, sparse algorithms necessitate high sparsity rates (e.g., $> 95\%$) to deliver appreciable performance benefits over their dense alternatives [15, 130, 226, 227]. As a result, current sparse algorithms and hardware architectures may only be able to deliver tangible benefits compared to their highly-optimized dense counterparts if significantly elevated sparsity levels are reached.

These fundamental dilemmas regarding current sparse neural networks motivate us to explore potential avenues beyond the conventional design flow that consists of neural network pruning and sparse accelerator design. We argue that increasing the synergy between sparsity patterns and hardware could furnish unique opportunities for improving the performance and efficiency of sparse neural networks. Sparsity patterns can be formed under proper constraints to establish a *contract* between the sparse neural network and hardware architecture. Hardware architectures that are traditionally tailored for dense arithmetic can support such contracts through strategic dataflow enhancements without compromising their dataflow efficiency.

This chapter demonstrates the possibility and potential of architecting sparsity patterns that exhibit high inherent regularity yet with expressiveness comparable to unstructured sparsity. Furthermore, we present a novel training method to form highly accurate and compact sparse neural networks through the expressiveness and plasticity of such sparsity patterns during the training process. Finally, we propose strategic hardware enhancements on a systolic array to process the proposed sparsity patterns efficiently. While the enforced neural network constraints allow optimal compression and computational predictability, the strategic hardware enhancements enable sparse layer processing in the compressed format without suffering under-utilization or decompression overheads.

In summary, we list the contributions of this chapter as follows:

- We present three complementary sparsity schemes by forming single and two-dimensional groups in the layer parameters and enforcing exclusivity constraints at each group. The outlined sparsity types are architected to facilitate layer processing in the compressed format through operations that resemble the dense arithmetic dataflow.

- We construct a novel analytical model for assessing the expressiveness of various sparsity types. The analytical model is utilized to characterize the shortcomings of the existing sparsity patterns in the literature and showcase the outstanding expressiveness properties of the complementary sparsity patterns.

- We present a novel training approach to shape the proposed sparsity patterns under regularity constraints and utilize the plasticity nature of deep neural networks in the training process to achieve compression rates that are competitive with unstructured sparsity.

- We outline the systolic array enhancements to implement the proposed sparsity schemes, explore various hardware design points to outline hardware cost scaling characteristics, and identify cost-effective hardware configurations.

- Overall, this chapter advocates for a synergistic co-design paradigm for sparse neural networks and hardware architectures, and uncovers a promising path for innovation in the design of efficient deep learning processing systems.

The outlined approach shares conceptual similarities with prior work on the novel sparsity paradigms and packed representations in Section 3.2.1, yet it distinguishes itself by architecting sparsity patterns to minimize hardware complexity. The proposed sparsity patterns are evolved under group-wise exclusivity constraints through a novel training approach instead of sub-optimally pruning pre-trained models to enforce these constraints. As a result, state-of-the-art compression rates on par with unstructured pruning can be delivered with a competitive accuracy while the enforced sparsity constraints keep the complexity of strategic hardware enhancements at levels substantially lower than unstructured sparsity.

12.2 Designing Complementary Sparsity Patterns

This section introduces three distinct formats to construct sparse neural network layers. The first approach, *neuron/filter superposition*, attempts to pack multiple sparse neurons/filters into a single dense neuron/filter. The second approach maps a group of weights in a neuron/filter into a single entry, resulting in *shortened neurons/filters*. Finally, we demonstrate that these two methods can be seamlessly coupled to construct *two-dimensional complementary sparsity patterns* that offer further representation flexibility and hardware processing efficiency.

Let us start with the definition of a vector dot product as in Equation (12.1), which is a fundamental operation in DNN computations. As each neuron in fully connected layers performs a vector dot product, each step in the convolutional layers can be formulated as a dot product of the flattened filter and the feature map tensor.

$$z_m = \vec{I} \cdot \vec{W}_m \tag{12.1}$$

In Equation (12.1), z_m, $\vec{I}$, and $\vec{W}_m$ represent the computed pre-activation value, the input

activation vector, and the weight set of the m'th neuron or convolution filter.

For a highly sparse weight vector $\vec{W}_m$ of a neuron or convolution filter, the non-compressed version of $\vec{W}_m$ would incur unnecessary storage for zero entries with the majority of multiply-accumulate steps making no tangible contribution to the final result in a naive dot product implementation.

12.2.1 Packing Sparsity with Neuron/Filter Superposition

Neuron/filter superposition can be constructed through multiple weight sets. Let us introduce an additional vector $\vec{W}_n$ of the same size as $\vec{W}_m$, to denote the weights of another neuron or convolution filter in the same layer. Let us assume the non-zero positions in $\vec{W}_m$ and $\vec{W}_n$ avoid overlap as in Equation (12.2):

$$\vec{W}_m[i] = 0 \quad \vee \quad \vec{W}_n[i] = 0 \quad , \quad \forall i \tag{12.2}$$

As the non-zero entries in $\vec{W}_m$ and $\vec{W}_n$ are disjoint, the reader will note that $\vec{W}_m$ and $\vec{W}_n$ can be stored as a single vector in a compressed format as in Equation (12.3) with a single-bit extension in the weight variables sufficing to recover $\vec{W}_m$ and $\vec{W}_n$ from the compressed vector $\vec{W}_{\{m,n\}}$. In the generalized case, the weights of k disjoint neurons or filters in the same layer can be compressed into a single vector at the cost of $\lceil log_2(k) \rceil$ additional bits at each weight.

$$\vec{W}_{\{m,n\}} = \vec{W}_m + \vec{W}_n \tag{12.3}$$

While the compression benefits are rather obvious, the described representation delivers yet another fundamental advantage, one perhaps not so evident at first glance. It enables computation of the pre-activations of the two packed neurons/filters simultaneously through a single dot product as in Equation (12.4) by keeping track of the two partial sums and updating only one of them at each multiply-accumulate step as weight vectors are disjoint.

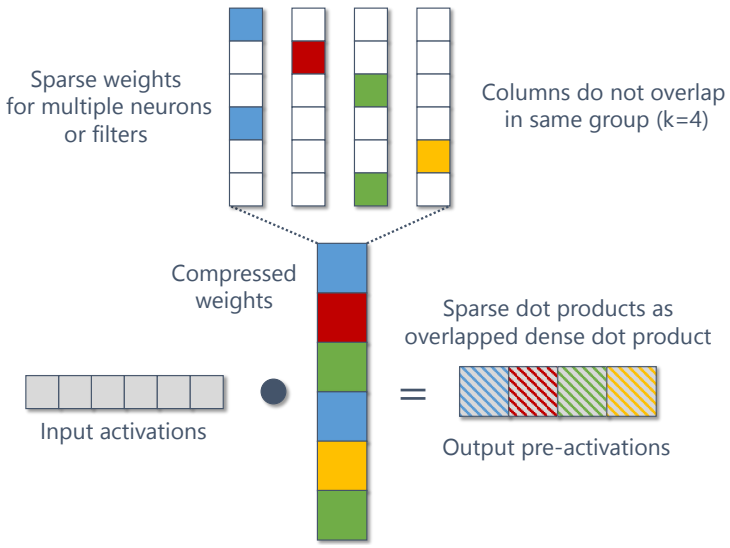

Figure 12.1. Neuron/filter superposition.

$$z_m, z_n = \vec{I} \cdot \vec{W}_{\{m,n\}} \tag{12.4}$$

In the general case, pre-activations for the k disjoint neurons or convolution filters can be generated through a single overlapped dot product by employing k partial sums and updating only one of them at each multiply-accumulate step.

In summary, the visualized scheme in Figure 12.1 requires a non-zero weight entry at each index position to be claimed only by a single neuron or filter in a group of k; thus, each weight group can be stored as a single vector and processed with a single dense dot product.

We implement filter superposition in the convolutional layers by enforcing groups in the output dimension of the weight tensor, which corresponds to distinct output channels.

12.2.2 Packing Sparsity with Shortened Neurons/Filters

An orthogonal scheme can be constructed within the boundaries of a single neuron or filter. This approach requires us to partition each weight vector into groups of l weight entries. Let us assume no more than one entry to be non-zero within each group as in Equation (12.5); then l weight entries in this group can be stored as a single entry.

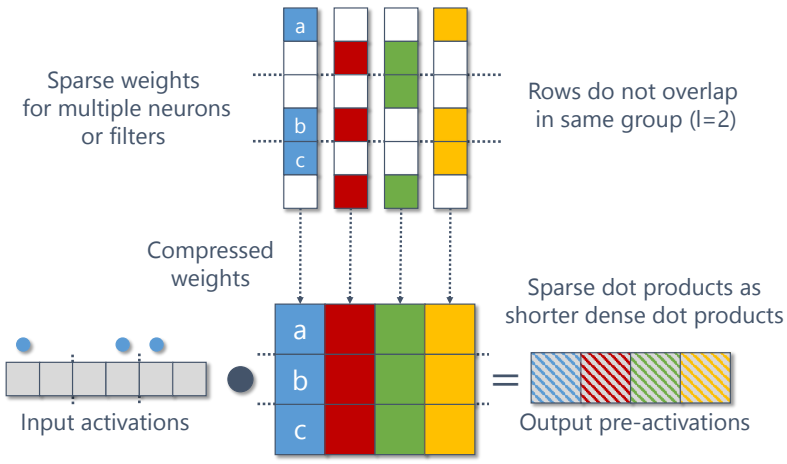

Figure 12.2. Shortened neurons/filters.

$$non\_zero\_count(\vec{W}[i \times l : (i+1) \times l]) \leq 1 \quad \forall i \tag{12.5}$$

The proposed packing scheme effectuates the compression of each weight vector size by l. Each weight entry needs to be extended by $\lceil log_2(l) \rceil$ bits to enable proper position restoration within the group.

The described sparsity pattern translates to yet further computational benefits. Any dot product that involves the compressed weight vectors can be performed efficiently in l times fewer multiply-accumulate steps. As only one weight entry will be non-zero at each group, the proper activation value from a group of activations should be selected at each multiply-accumulate step according to the position information stored in the weight entries.

The visualized dot product scheme in Figure 12.2 bears similarities to the conventional sparse dot product algorithms [168], yet our approach differs due to the usage of a local index value within each group, thus requiring smaller bit-widths for indexing and minimizing hardware communication overhead.

Shortened filters are constructed in the convolutional layers by enforcing exclusive groups in the input dimension of the weight tensor, which differentiates the input channels.

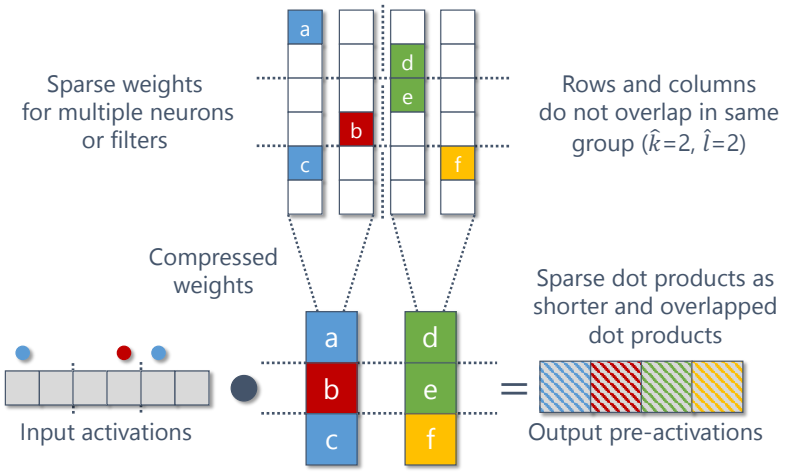

Figure 12.3. Complementary sparsity patterns in two dimensions.

12.2.3 Complementary Sparsity Patterns in Two Dimensions

We have so far constructed the sparsity groups in a single dimension of the weight matrices; yet, the complementary sparsity patterns can effectively generalize to two-dimensional regions in the weight matrices. Moreover, two-dimensional configurations could offer unique computational advantages compared to their single-dimensional counterparts, as they can construct groups more effectively by breaking them down into two dimensions. This section provides a conceptual discussion on the two-dimensional complementary sparsity patterns.

The neuron/filter superposition scheme constructs groups of shape $1 \times k$ in the weight matrices where only one entry is allowed to be non-zero within a group. Similarly, the shortened neurons/filters technique forms one-dimensional groups of shape $l \times 1$. As demonstrated in Figure 12.3, the natural extension to these two methods could be accomplished through two-dimensional regions of shape $\hat{l} \times \hat{k}$ in the weight matrices where only a single weight entry can assume a non-zero value within the confines of a rectangular region.

Two-dimensional sparsity patterns provide competitive compression benefits compared to their single-dimensional counterparts. An $\hat{l} \times \hat{k}$ group can be effectively compressed into a single weight to deliver reductions in both the input and the output dimensions of the weight matrices. The position of the non-zero weight can be recovered by storing position indices for

both dimensions at the cost of $\lceil log_2(\hat{l}) \rceil + \lceil log_2(\hat{k}) \rceil$ additional bits. The computational cost of a two-dimensional group is confined to a single multiply-accumulate operation that can be accomplished by selecting the proper activation and updating the correct partial sum at each step.

Two-dimensional sparsity patterns engender opportunities for hardware cost reduction by decomposing the one-dimensional selection operation from large groups into two selection steps with smaller group sizes (one at each dimension). Furthermore, the more balanced partial sum and activation requirements of the two-dimensional groups result in more sustainable memory bandwidth characteristics. A detailed discussion of these benefits can be found in Section 12.5.2.

Finally, higher dimensional sparsity groups can be constructed in weight tensors that incorporate more than two dimensions. For instance, the construction of groups with up to four dimensions is feasible in the convolutional layers. Nevertheless, the computational benefits for cases that extend beyond the two dimensions are often limited due to current hardware dataflow and memory layout constraints.

12.3 Evaluating Sparsity Type Expressiveness Through Analytical Models

We have emphasized the expressiveness of complementary sparsity; nevertheless, the source of such characteristics has so far remained unexplored. This section aims to provide a theoretical discussion and an analytical model to illustrate the potential of complementary sparsity patterns in representing unique sparse neural network architectures.

The expressiveness of a sparsity type correlates with its ability to represent unique sparsity configurations. While a fixed sparsity configuration is unlikely to deliver state-of-the-art accuracy upon training [196], it is possible to obtain highly-accurate sparse models if the underlying sparsity type offers the ability to express a large number of sparsity configurations.

Let us start our discussion by exploring the characteristics of unstructured sparsity. The most flexible sparsity format, unstructured sparsity, can represent all possible unique sparse

network configurations in a neural network architecture at the given non-zero parameter budget and thus constitutes the maximum theoretical limit. In a neural network architecture with p parameter positions, unstructured sparsity can express the following quantity of distinct non-zero parameter configurations when exactly r of such locations are non-zero:

$$\binom{p}{r} \tag{12.6}$$

Block-wise [152] and group-wise sparsity [153] allow non-zero configurations only in the granularity of groups. Due to the coarse granularity of the representation, the possible number of non-zero parameter configurations diminishes to the following quantity where g denotes the group size:

$$\binom{p/g}{r/g} \tag{12.7}$$

The reader will note that this quantity can be significantly smaller than that of the unstructured sparsity, even in the case of relatively small group sizes. This observation could explain the observed accuracy gap between the unstructured and group-wise sparsity patterns in practice since it is more likely to obtain a sparsity configuration with higher classification accuracy in a much larger pool of candidates.

Let us now derive the number of possible non-zero parameter configurations that complementary sparsity offers at the same non-zero parameter budget. Complementary sparsity allows one weight value to be non-zero at each group; thus, the sparsity structure has to consist of exactly r groups and the size of each group needs to be set to p/r. A non-zero entry can be located at any one of the p/r distinct positions within a group, and thus the complementary sparsity can represent the following amount of distinct configurations with r groups:

$$(p/r)^r \tag{12.8}$$

Table 12.1. Sparsity types vs. non-zero parameter configurations.

p	r	Sparsity Type	Configurations
10^4	10^2	Unstructured	$\sim 6.52 \times 10^{241}$
		Complementary	1.00×10^{200}
		Group ($g = 10$)	$\sim 2.63 \times 10^{23}$
		Group ($g = 100$)	1.00×10^{2}
	10^3	Unstructured	$\sim 8.73 \times 10^{1409}$
		Complementary	1.00×10^{1000}
		Group ($g = 10$)	$\sim 6.39 \times 10^{139}$
		Group ($g = 100$)	$\sim 1.73 \times 10^{13}$

Interestingly, the expression in Equation (12.8) is known to be a lower bound for the binomial coefficient in Equation (12.6) [228], and its accuracy tightens particularly when $r \ll p$ (i.e., at high sparsity rates). As a result, complementary sparsity is expected to deliver non-zero configuration counts that are oftentimes close to unstructured sparsity.

Table 12.1 demonstrates what the derived equations correspond to for a set of p, r, and g values. In the scope of this example, we assume $p = 10^4$ and investigate two different sparsity rates where $r = 10^2$ (99% sparsity) and $r = 10^3$ (90% sparsity). We utilize two group sizes ($g = 10$ and $g = 100$) to demonstrate the possible non-zero parameter configurations that can be represented with group-wise sparsity patterns. The results in Table 12.1 showcase the expressiveness of the complementary patterns clearly, where the number of configurations is often astronomically higher than the group-wise sparsity patterns and follows more closely the maximum theoretical limit that is set by unstructured sparsity. As a result, the complementary sparsity patterns are more likely to provide a sparsity configuration with highly competitive accuracy even at extreme sparsity rates.

In this section, we have provided a novel analytical approach to estimate the expressiveness of the sparsity types. We believe that the presented analytical method and the derived metrics are extremely valuable in evaluating sparsity types. The guidance provided by such

analytical models could effectively steer the design process to identify hardware-friendly sparsity types that offer competitive accuracy levels.

12.4 Evolving Sparsity Patterns in Training

We have introduced a variety of complementary sparsity schemes, outlined their benefits, and demonstrated their expressiveness in representing unique sparsity configurations. Searching for a competitive sparse neural network model within a large pool of candidates remains a formidable problem, nonetheless. Therefore, the formation and training of accurate sparse neural networks necessitate further deliberation.

Dense model training coupled with unstructured weight pruning is a widely employed approach to obtain highly-accurate sparse deep learning models [196]. However, this approach may not be suitable for producing complementary sparsity patterns since the global or layer-wise ranking heuristics need to be modified to accommodate group-wise exclusivity restrictions as no more than a single non-zero weight entry can be accommodated within each group. Even then, conducting group-wise pruning on a pre-trained model necessitates a significant amount of model restructuring through pruning steps, timorous by necessity, and consequent fine-tuning iterations.

We instead propose a novel integrated training approach to evolve sparsity patterns dynamically by enforcing group-wise exclusivity restrictions through soft pruning[20] at every training iteration. We utilize custom propagation and update rules to facilitate effective sparsity pattern re-arrangements in training and identify a competitive sparse architecture among the large set of candidates offered by complementary sparsity.

[20]The term soft pruning defines a process where the pruning decision is not permanent in training and, if necessary, revocable by the training algorithm.

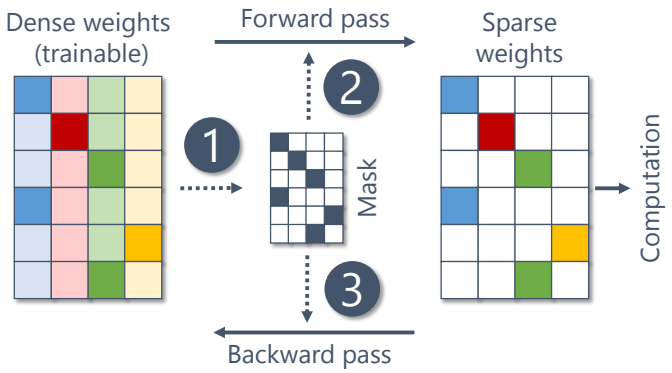

Figure 12.4. Sparsity generation and weight update in training.

12.4.1 Overview of the Sparse Training Process

We hold the layer weights in a dense format without any sparsity imposition. The unrestricted weight matrices are sparsified prior to each forward pass through masking, and the layer computations are carried out with the sparse version of the weight matrices. The forward pass of the fully connected layers can be formulated as in Equation (12.9):

$$\begin{aligned} \hat{W} &\leftarrow \Theta \odot W \\ \vec{z} &= \vec{I} \times \hat{W} \end{aligned} \tag{12.9}$$

The mask tensor (Θ) for each layer is generated in every forward pass by identifying the maximum magnitude weight entry at each group (Figure 12.4 - Step 1). The mask entries are set to 1 for the maximum magnitude weights at each group and 0 for the others. The sparse weights ($\hat{W}$) are generated through the element-wise multiplication ($\odot$) of the unrestricted weights (W) and the mask (Θ) (Figure 12.4 - Step 2). Finally, the output pre-activations ($\vec{z}$) are computed through the multiplication of the input activations $\vec{I}$ and the sparse weights $\hat{W}$.

The generated mask tensor in the forward pass plays an important role in the backward pass as well (Figure 12.4 - Step 3). The gradients for the non-masked weight entries are directly back-propagated from the sparse weights to dense trainable weights, and the gradient for the masked entries is multiplied by a scaling factor as in Equation (12.10):

$$\frac{\partial}{\partial W} = \Theta \odot \frac{\partial}{\partial \hat{W}} + (1 - \Theta) \odot \frac{\partial}{\partial \hat{W}} \times \beta \tag{12.10}$$

The training process is mildly sensitive to the described scaling factor (β); we observe that employing a small value (i.e., $\beta = 0.1$) often exhibits superior performance in practice compared to the full propagation of the gradients to masked entries ($\beta = 1$) or to the complete masking of their gradients ($\beta = 0$). While full gradient propagation might fuel increased competition and instability, complete masking frequently congeals to a locally optimal result. It should be noted that even when the scaling factor is set to zero, the masked entries still retain the possibility of being updated whenever they assume the mantle of leadership in the group due to a diminution in the magnitude of the group's current leader. While a proper selection of the scaling factor (i.e., $\beta = 0.1$) delivers a highly competitive accuracy on a variety of benchmarks, the accuracy could be improved marginally by repeating the training process with a few different scaling factors in a search for the optimal value for the given benchmark.

12.4.2 Layer-wise Group Size Selection Steps

An important design consideration prior to training is the selection of the unique group size, and therefore the sparsity ratio, at each layer. For instance, a group size of $k = 8$ implies that $1/k = 1/8 = 12.5\%$ of the weight entries will be non-zero in the enforced layer. To the best of our knowledge, the determination of the optimal sparsity rates for each layer remains an open problem [229]. Enforcing an identical sparsity rate at every layer usually leads to bottlenecks in the layers that originally held a smaller number of parameters. For instance, Frankle et al. [229] demonstrate that certain pruning heuristics [230, 231, 232] can outperform random pruning when applied at initialization, as such heuristics can deliver a better selection of the sparsity rates at each layer compared to random pruning, which induces the same sparsity rate at every layer.

While selecting the optimal sparsity rates at each layer is an open research problem whose precise solution lies beyond the scope of this work, we recognize a set of practical rules

through our experimental analysis and define a process for identifying a competitive set of group sizes with minimal design space exploration. We employ the number of remaining parameters in a sparse layer as a proxy for assessing the layers' expressive power and experimentally observe that balancing the number of parameters in the sparse layers can be an effective bottleneck avoidance strategy. We carry out four fundamental steps to identify layer-wise group size values:

Step 1: We compute the number of parameters in the dense neural network and determine a final parameter budget for the compressed model. The initial step is influenced by the resource limitations in the inference device and the tolerable accuracy loss vis-à-vis the dense model. As a result, distinct parameter budgets can be encountered, each defining a Pareto-optimal design point at varying accuracy levels.

Step 2: We distribute the identified parameter budget equally to each layer. Layers that had originally contained fewer parameters than the final allocated budget are kept in the dense format with the remaining part of their share equally distributed to other layers. This approach might prove insufficient in delivering reductions in the number of FLOP (floating-point operation) or MAC (multiply-accumulate) operations since the initial layers in modern networks (i.e., ResNets [13]) have a significantly higher FLOP/parameter ratio than the later layers. Thus, the initial layers need to be targeted manually after the initial allocation to increase sparsity and deliver further FLOP or MAC reductions.

Step 3: The group size is computed for each layer through the division of the initial layer parameter count by the final budget allocated to the layer and rounding the result to the nearest integer. The reader will note that the compression rates will slightly deviate from the target value due to rounding effects, and thus the exact parameter and FLOP compression rates could then be identified by using the computed group sizes at each layer.

Step 4: We carry out sparse model training with the identified group sizes, measure the final accuracy, and compute the accuracy loss compared to the dense model. If the target accuracy metric cannot be met within the given compressed parameter budget, we identify a less aggressive compression target and repeat the outlined process starting from the initial step. If the

accuracy goal is satisfied, it is still feasible to repeat the outlined steps with a more aggressive compression target that meets the desired accuracy goal at a smaller resource budget.

As the outlined process provides a certain amount of leeway depending on the compression target, Table 12.2 provides the selected group sizes to facilitate reproducibility.

12.5 DNN Inference with Complementary Sparsity

Proposed sparsity patterns offer tremendous advantages in DNN inference, including optimal memory compression and seamless scalability to a variety of sparsity rates because of their regularity. Moreover, layer parameters can be loaded into the computational fabric directly and used in the compressed format. The additional weight bits in the proposed sparse representations can be utilized to control the computational flow of non-sparse hardware architectures, enjoying the advantages of the fully utilized dense matrix operations on the high-throughput hardware platforms. We elaborate on the practical memory compression rates and present a case study on a systolic array accelerator to outline the computational benefits.

12.5.1 Packing Sparse Layers for Efficient Compression

The fully connected and the convolutional layer parameters can be represented as a matrix without loss of generality. The matrix shape is $\hat{W}_{c_i \times c_o}$ for the fully connected layers, where c_i and c_o indicate the number of input and output neurons. Convolutional layers can be represented as a flattened matrix $\hat{W}_{c_i s^2 \times c_o}$ where c_i, c_o, and s denote the input channel, output channel, and spatial dimensions in the original parameter tensor.

Neuron/filter superposition packs k columns into a single column, resulting in a k times reduction compared to the naive representation of the weight matrix $\hat{W}$. A group of k columns necessitates $\lceil log_2(k) \rceil$ additional bits at each weight value to capture position information. As a result, the weight size is extended by the ratio $\lceil log_2(k) \rceil / b$ where b is the original bit width (e.g., $b = 32$ for single-precision floating point, $b = 8$ for 8-bit fixed point). The practical memory compression rate is formulated in Equation (12.11) and depicted in Figure 12.5.

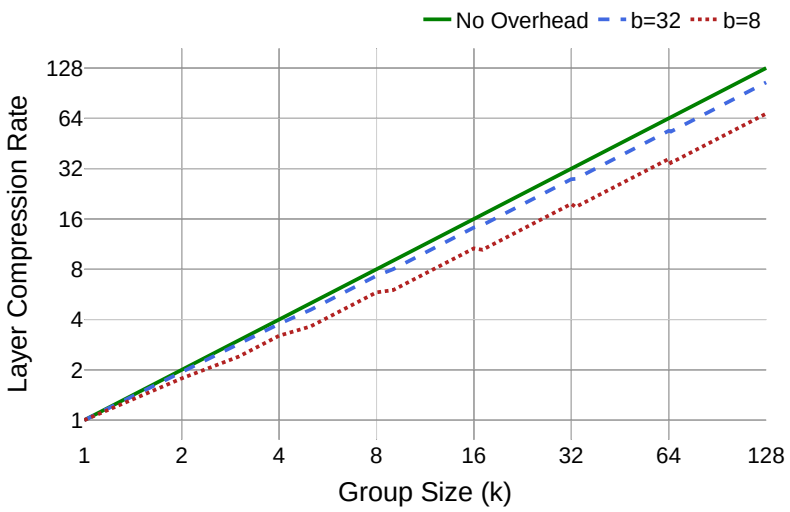

Figure 12.5. Packed group size vs. layer compression rate.

$$CR(k,b) = \frac{k}{1 + \frac{\lceil log_2(k) \rceil}{b}} \tag{12.11}$$

The compression rate can be formulated similarly for the remaining sparsity schemes, with the only difference being of compression delivered by merging the matrix rows in the case of shortened neurons/filters, and both matrix rows and columns for the two-dimensional patterns.

12.5.2 Processing Sparse Layers in the Dense Format

The commodity hardware platforms and DNN accelerators can deliver extensive throughput in dense matrix operations. However, the processing of sparse models through dense matrix operations results in an undesirable under-utilization, as the computations that involve zero-valued parameters do not have a material impact on the final result.

Sparse inference has been a widely researched topic whose focal areas include innovations in algorithms [233] and hardware boosts through accelerators [164, 167, 168, 169]. However, sparse algorithms and hardware accelerators offer advantages over dense counterparts only at high sparsity rates due to exacerbated control and communication overheads [15, 130, 226, 227].

Our hardware study involves highly regular, systolic array-based deep learning accelerators such as the architecture demonstrated in Section 2.3. The efficiency and high throughput

of systolic arrays in dense matrix operations stem from their rigid data flow, making them highly unappealing for sparse DNN inference. We undertake the challenge of accelerating sparse layers on the systolic array through minor hardware enhancements to boost hardware utilization and inference performance significantly. The presented principles generalize to the dense arithmetic operations in other platforms such as CPU (central processing unit) and GPU (graphics processing unit), and can be implemented through small enhancements to the existing control mechanisms that allow programmability.

The pre-activations of the superpositioned neurons/filters can be computed through a single dot product by keeping track of multiple partial sums and updating only one of them at each multiply-accumulate step. This operation can be achieved by mapping the superpositioned weight matrix column into a systolic array column directly and passing k partial sums to each systolic array column instead of one. We introduce multiplexing functionality at each MAC unit in Figure 12.6-(a) to select and update only one partial sum from a group of k. Other partial sums are forwarded to the next MAC unit directly. The correct partial sum is selected at each MAC unit by checking the extension bits in the stored weight value.

Enhancements in Figure 12.6-(b) allow us to support the shortened neurons/filters. The shortened neurons/filters scheme requires each weight entry to select exactly one activation input according to its original position in the sparse format. This behavior can be attained by feeding a group of l activation inputs into each MAC unit instead of one and introducing input multiplexing functionality to enable each MAC unit to select exactly one activation from the group of l by checking the extension bits in the stored weight value.

It is expected that the implemented flow control modifications in Figure 12.6-(b) to be consistently more hardware efficient than the one in Figure 12.6-(a) because only a single multiplexer is required for input selection in Figure 12.6-(b), and the multiplexer data widths at the inputs (e.g., 8 bits) are often noticeably smaller compared to the partial sums (e.g., 16+ bits) for the fixed-point MAC units.

Two-dimensional complementary sparsity patterns can be effectively supported by in-

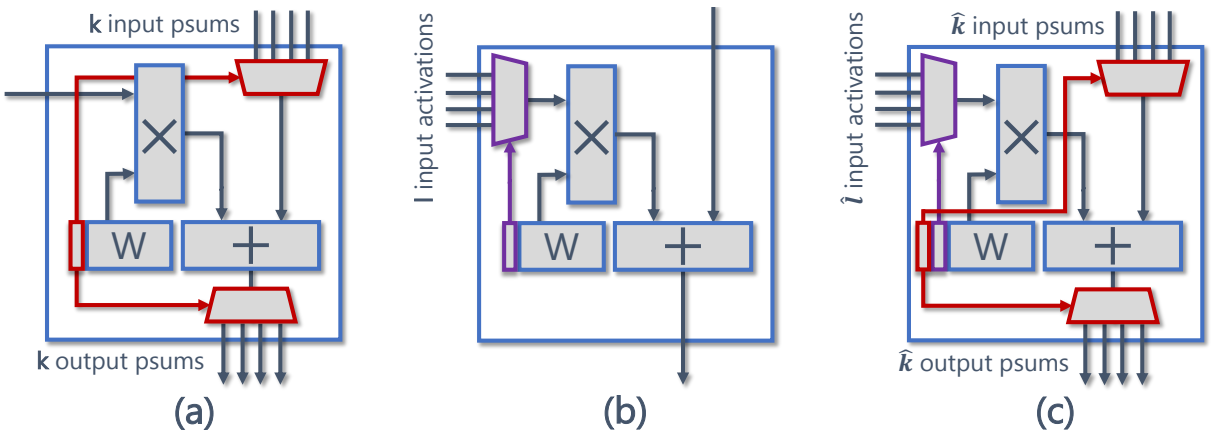

Figure 12.6. Flow-controlling multiply-accumulate units for sparse inference.

troducing multiplexing capability for both the partial sums and the input activations. The demonstrated architecture in Figure 12.6-(c) multiplies the pre-loaded weight with the selected value from a group of input activations and updates the selected partial sum with the multiplication result. The multiplexer configurations in this design can be controlled independently through the dedicated fields of the weight extension bits.

The design in Figure 12.6-(c) has certain inherent inefficiencies since it requires hardware for both input activation and partial sum selection, and it involves output multiplexing similar to Figure 12.6-(a), which is expected to be particularly costlier than input multiplexing in Figure 12.6-(b). On the other hand, the flow control mechanism in Figure 12.6-(c) can support large group sizes efficiently by decomposing the selection steps into two dimensions. As a consequence, while the hardware overhead of the multiplexers in the single-dimensional designs grows linearly with respect to the group size ($\propto k$ for the given group size k), the multiplexer hardware overhead in the two-dimensional configuration scales proportionally to the square root of the group size ($\propto \sqrt{k}$) when both dimensions are balanced. As a result, the flow control configuration in Figure 12.6-(c) may exhibit overheads especially more significant than the one in Figure 12.6-(b) for the small group sizes, yet it is expected to deliver more cost-effective designs in the case of large group sizes due to more graceful scalability characteristics.

Furthermore, the two-dimensional configuration in Figure 12.6-(c) leads to a more

balanced memory input/output bandwidth and further minimizes the overall memory bandwidth requirements. To illustrate, a conventional systolic array of size $s \times s$ consumes s activation values as input and produces s partial sums at each cycle, thus resulting in a bandwidth need that is proportional to $2s$. The neuron/filter superposition technique requires an output bandwidth increment by a factor of k that results in an overall memory bandwidth of $(1+k)s$. A similar increase is observed in the shortened neurons/filters scheme due to the heightened need for more input activations. In comparison, the two-dimensional configurations cause both the input and the output bandwidth to increase by a factor of $\sqrt{k}$ and result in an overall bandwidth of $2\sqrt{k}s$. In other words, two-dimensional configurations can cut down the linear bandwidth growth ($\propto k$) in the single-dimensional configurations to the square root of the group size ($\propto \sqrt{k}$), and thus alleviate the memory bandwidth problem for large group sizes effectively.

Different group sizes at each layer could be supported on the same hardware by designing sufficiently large multiplexing units and utilizing for smaller group sizes a subset of the multiplexer inputs. As the designs with the proposed flow-controlling multiply-accumulate units operate essentially identically to the standard systolic array in the dense layers by utilizing pre-determined multiplexer input positions, the sparse layer performance can be boosted dramatically for large groups (e.g., $64\times$ for $k, l = 64$) through the full use of all available multiplexer inputs. In addition, standard systolic array scheduling and matrix tiling techniques can be applied seamlessly in the enhanced designs due to the retained dataflow regularity.

The proposed designs enable inference with compressed weights. The dynamic nature of activation sparsity makes it challenging to construct regularity constraints within a group of activations and accommodate activation sparsity through the proposed enhancements in a systolic array. On the other hand, the proposed techniques can be seamlessly coupled with the existing memory bandwidth reduction techniques, such as specialized compressed representations for sparse activations in order to match the efficiencies delivered by the compressed weights in terms of memory bandwidth requirements.

12.6 Experimental Method

12.6.1 Model Compression Experiments

We investigate parameter compression rates, and remaining FLOP percentages that can be achieved through the proposed sparsity patterns on a variety of benchmarks, including LeNet-300-100 and LeNet-5-Caffe models [28] on the MNIST dataset [211], VGG-like [135], VGG-19 [159], and ResNet-56 [13] models on the CIFAR-10 dataset [207], and ResNet-50 [13] model on the ILSVRC'12 (ImageNet) [224] dataset. We utilize the reported results of the recent pruning techniques as a basis for comparison.

The models with complementary sparsity patterns are trained from scratch, as in the dense models. As a result, the lengthy and iterative fine-tuning steps of conventional model pruning are avoided. When more than one data point is reported for a benchmark, we differentiate them with unique letter suffixes (e.g., -A).

Table 12.2 indicates the group size enforced at each layer. The group sizes are initially selected to balance the number of parameters at each layer within the given parameter budget. As a second step, to reduce the computational complexity even further, we increase the group size (e.g., up to $2-4\times$) in the earlier layers with a large number of FLOPs. The maximum enforced group size at certain layers is limited by the original layer shape. Finally, we confine our analysis of two-dimensional patterns to square-shaped groups only.

Experimental models are implemented in PyTorch [213]. We have used 2 machines for model training, each with $4\times$ Intel Xeon E5-2630 v4 CPUs, $2\times$ NVIDIA GeForce GTX 1080Ti GPUs, and 32GB RAM.

12.6.2 Inference Performance Simulations

The inference performance of the obtained models is estimated through cycle-accurate simulations in SCALE-Sim [62]. SCALE-Sim delivers precise performance analysis of deep learning accelerators by considering SRAM (static random-access memory) / DRAM (dynamic

Table 12.2. Group sizes enforced at each layer.

LeNet-300-100 Superposition/Shortened-A
109-14-1
LeNet-300-100 Two-Dimensional-A
121-16-1
LeNet-300-100 Superposition/Shortened-B
235-30-1
LeNet-300-100 Two-Dimensional-B
361-49-1
LeNet-5-Caffe Superposition
4-50-500-10
LeNet-5-Caffe Shortened
1-20-500-10
LeNet-5-Caffe Two-Dimensional
1-64-784-16
VGG-like Superposition/Shortened-A
1-1-3-6-12-24-24-48-96-96-96-96-96-10-1
VGG-like Two-Dimensional-A
1-1-4-9-16-25-25-64-121-121-121-121-121-9-1
VGG-like Superposition/Shortened-B
1-6-6-12-24-48-48-96-128-128-128-128-128-32-1
VGG-like Two-Dimensional-B
1-4-9-16-25-49-49-100-196-196-196-196-196-25-1
VGG-19 Superposition/Shortened/Two-Dimensional
1-1-1-1-1-1-4-16-16-16-16-16-36-64-64-64-64-144-1
ResNet-56 Superposition/Shortened
1-2-2-2-2-2-2-2-2-2-2-2-2-2-2-2-2-2-2-2-4-7-7-7-7-7-7-7-7-7-7-7-7-7-7-7-7-7-7-14-29-29-29-29-29-29-29-29-29-29-29-29-29-29-29-29-1
Shortcuts: 1-1
ResNet-56 Two-Dimensional
1-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-16-25-25-25-25-25-25-25-25-25-25-25-25-25-25-25-25-1
Shortcuts: 1-1
ResNet-50 Superposition/Shortened
1-1-4-2-2-4-2-2-4-2-4-4-2-2-4-2-2-4-2-2-4-2-4-8-4-4-8-4-4-8-4-4-8-4-4-8-4-4-8-4-4-8-4-4-8-4-8-16-8-8-16-8-8-16-8-16 Shortcuts: 2-4-8-16
ResNet-50 Two-Dimensional
1-1-4-1-1-4-1-1-4-1-4-4-1-1-4-1-1-4-1-1-4-1-4-9-4-4-9-4-4-9-4-4-9-4-4-9-4-4-9-4-9-16-9-9-16-9-9-16-9-16 Shortcuts: 1-4-9-16

random-access memory) access and the computation latency. We utilize the default Eyeriss [166] accelerator configuration provided within SCALE-Sim, which is a weight-stationary architecture that consists of a 12×14 systolic array and 108KB SRAM buffers each for the input and output features, and the weight coefficients.

We scale the input (for shortened neurons/filters) or output (for neuron/filter superposition) channel dimensions of each layer through division by the assigned group size (k, l) to simulate the processing of the proposed sparse layers in the form of a smaller dense layer. In the case of two-dimensional patterns, each layer dimension is divided by the corresponding dimension size of the implemented sparsity group.

We utilize the compressed architectures of structured pruning methods in prior literature for further performance comparisons by measuring their inference latency on a systolic array of identical size. An accurate performance characterization is feasible only for the studies that report layer-wise compression information (i.e., the remaining number of neurons or filters at each layer or layer-wise group sizes in the case of [159]). The complexity of the sparse accelerator designs [26] and the negligible performance benefit of the unstructured sparsity patterns on a systolic array preclude direct performance comparisons with the unstructured pruning techniques.

12.6.3 Hardware Measurements for Flow-Controlling MAC Units

We construct fixed-point and floating-point MAC units and implement the proposed flow control enhancements in Verilog for overhead characterization. Estimated area and power overhead values are measured through hardware synthesis experiments with the Synopsys Design Compiler. Synthesis experiments are performed with Silvaco Open-Cell (15 nm) and Synopsys DesignWare libraries at 1GHz target frequency. We adjust MAC unit input switching activities during power analysis to model a weight stationary architecture [26]. We employ the measured overhead values to estimate the efficiency of the proposed hardware architectures through metrics such as the number of effective MAC operations per unit area and power.

Table 12.3. Classification error, parameter compression rate, and remaining FLOPs percentage for LeNet-300-100 on MNIST.

	Type (S/U)	Base Error (%)	Final Error (%)	CR (×)	FLOPs (%)
Louizos et al. [137]	S	1.60	1.80	9.29	11
Louizos et al. [140]	S		1.40	3.84	26
	S		1.80	9.99	10
Xiao et al. [29]	S	1.60	1.82	10.98	9
Han et al. [28]	U	1.64	1.59	12.14	8
Guo et al. [125]	U	2.28	1.99	56	
Dong et al. [126]	U	1.76	2.43	66.7	
Ullrich et al. [127]	U	1.89	1.94	64	
Molchanov et al. [128]	U	1.64	1.92	68	
Li et al. [129]	U	2.28	2.18	114	
Xiao et al. [29]	U	1.72	1.78	80	
Shortened-A	S*	1.75	2.11	50.22	1.99
Superposition-A	S*	1.75	1.90	50.22	1.99
Two-Dimensional-A	S*	1.75	1.80	55.24	1.81
Shortened-B	S*	1.75	2.40	88.71	1.13
Superposition-B	S*	1.75	2.08	88.71	1.13
Two-Dimensional-B	S*	1.75	2.22	**117.59**	**0.85**

12.7 Experimental Results

12.7.1 Model Compression Results

Proposed sparsity patterns consistently achieve excellent parameter compression rates that are competitive and even frequently surpass unstructured weight pruning on various benchmarks, as demonstrated in Tables 12.3-12.8.[21] Our results outperform structured pruning methods by a large margin as their effectiveness is limited by the coarse granularity of the elimination process. For instance, the $\sim$118$\times$ and $\sim$251$\times$ compression rates for LeNet-300-100 and LeNet-5-Caffe models exhibit remarkable superiority over structured pruning and preponderantly elevated rates

[21] Unreported data points are left blank in Tables 12.3-12.8. We use the following indicators to mark the pruning/sparsity types in the tables: S - structured sparsity, U - unstructured sparsity, S* - proposed sparsity patterns with structured nature and seamless hardware regularity.

Table 12.4. Classification error, parameter compression rate, and remaining FLOPs percentage for LeNet-5-Caffe on MNIST.

	Type (S/U)	**Base Error (%)**	**Final Error (%)**	**CR (×)**	**FLOPs (%)**
Wen et al. [130]	S		1.00	1.06	22
Neklyudov et al. [136]	S	0.80	0.86	5	9
Louizos et al. [137]	S	0.90	1.00	156	7
Louizos et al. [140]	S		0.90	11	49
	S		1.00	70	17
Xiao et al. [29]	S	0.78	0.80	43	7
Han et al. [28]	U	0.80	0.77	11.97	16
Guo et al. [125]	U	0.91	0.91	108	
Dong et al. [126]	U	1.27	2.04	111	
Ullrich et al. [127]	U	0.88	0.97	162	
Molchanov et al. [128]	U	0.80	0.75	280	
Li et al. [129]	U	0.91	0.91	298	
Xiao et al. [29]	U	0.78	0.80	260	
	U	0.91	0.91	**310**	
Shortened	S*	0.86	0.93	141.15	16.10
Superposition	S*	0.86	0.98	223.63	**4.59**
Two-Dimensional	S*	0.86	0.85	251.27	13.68

over the unstructured pruning methods. Similarly, the $\sim$99$\times$ reduction in the VGG-like model noticeably surpasses all compared pruning techniques. We report in Tables 12.7-12.8 $\sim$12$\times$ and $\sim$8$\times$ parameter reductions in the challenging ResNet-56 and ResNet-50 benchmarks, whose prior highest compression rates with an acceptable accuracy loss stand at $\sim$3.4$\times$ and 2.2$\times$. Despite the regularity of the complementary sparsity patterns, the proposed dynamic training approach can frequently deliver higher sparsity rates than the prior unstructured pruning methods since these techniques usually perform pruning post-training and complement it with fine-tuning.

The remaining FLOPs percentage can be used as a significant indicator of inference complexity. Proposed sparse models induce FLOP percentages that are a small fraction of the dense models. For instance, LeNet-300-100, LeNet-5-Caffe, and the VGG-like models require only $\sim 0.9 - 2.0\%$, $\sim 4.6 - 16.1\%$, and $\sim 5.7 - 18.6\%$ of the FLOPs respectively compared to

Table 12.5. Classification error, parameter compression rate, and remaining FLOPs percentage for VGG-like model on CIFAR-10.

	Type (S/U)	Base Error (%)	Final Error (%)	CR (×)	FLOPs (%)
Li et al. [135]	S	6.75	6.60	2.77	66
Neklyudov et al. [136]	S	7.20	7.50		43
	S	7.20	9.00		32
Zhuang et al. [141]	S	6.01	5.43	15.58	34.90
Zhu et al. [142]	S	6.42	6.69	8.50	31.25
Xiao et al. [29]	S	7.60	8.50		23
Molchanov et al. [128]	U	7.30	7.30	65	
Xiao et al. [29]	U	7.60	7.82	75	
Shortened-A	S*	7.15	7.29	43.98	18.62
Superposition-A	S*	7.15	7.31	43.98	18.62
Two-Dimensional-A	S*	7.15	7.70	51.75	17.17
Shortened-B	S*	7.15	8.45	80.01	**5.65**
Superposition-B	S*	7.15	7.81	80.01	**5.65**
Two-Dimensional-B	S*	7.15	7.87	**98.58**	5.96

Table 12.6. Classification error, parameter compression rate, and remaining FLOPs percentage for VGG-19 model on CIFAR-10.

	Base Error (%)	Final Error (%)	CR (×)	FLOPs (%)
Kung et al. [159]		5.30	6.25	
Shortened	5.10	5.09	**29.53**	**13.28**
Superposition	5.10	5.04	**29.53**	**13.28**
Two-Dimensional	5.10	5.06	**29.53**	**13.28**

their dense counterparts. We obtain ResNet-56 and ResNet-50 models with $\sim$18.6% and $\sim$25.9% of the original FLOPs. Such a reduction to roughly one quarter of the original FLOPs for the ResNet models is at least 2$\times$ smaller than the bulk of the competition.

All proposed methods are capable of delivering Pareto-optimal design points in different benchmarks, and they often result in similar classification accuracy values for identical layer-wise

Table 12.7. Classification error, parameter compression rate, and remaining FLOPs percentage for ResNet-56 on CIFAR-10.

	Type (S/U)	**Base Error (%)**	**Final Error (%)**	**CR (×)**	**FLOPs (%)**
Li et al. [135]	S	6.96	6.94	1.16	72.40
He et al. [139]	S	7.20	8.20		~ 50
Zhuang et al. [141]	S	6.20	6.19	3.37	52.91
Yu et al. [144]	S	6.96	6.99	1.74	56.39
He et al. [145]	S	6.41	6.11		85.30
	S	6.41	6.53		71.60
	S	6.41	6.22		58.90
	S	6.41	6.65		47.40
He et al. [30]	S	6.41	6.51		47.40
Kang & Han [148]	S	6.31	6.77	1.94	48.50
Wang et al. [149]	S	6.62	6.25		46.20
Shortened	S*	6.41	6.91	**12.02**	23.40
Superposition	S*	6.41	6.93	**12.02**	23.40
Two-Dimensional	S*	6.41	6.92	10.57	**18.55**

group sizes. As a result, we conclude that group size is an important factor that impacts final accuracy, with the role of group shape and orientation being relatively less consequential.

The structure of the shortened neurons/filters scheme shares similarities to the sparsity patterns obtained through column combining in Kung et al. [159].[22] Despite their structural similarity, our work differs from [159] fundamentally since we evolve the sparsity patterns through a novel dynamic training approach instead of sub-optimal pruning and manual column combining steps. Our effective training approach thus enables significantly higher compression rates than [159] without compromising accuracy.

We demonstrate the superiority of our approach by implementing the VGG-19 model on CIFAR-10 through the architectural description in [159], applying the proposed sparsity schemes, and comparing our final accuracy and the compression rates with the reported results in [159] in

[22]The shortened neurons/filters scheme is implemented by combining weight matrix rows in our study. A similar scheme is accomplished through column combining in [159] because of the construction of the matrix multiplication operation in the transposed format.

Table 12.8. Classification accuracy, parameter compression rate, and remaining FLOPs percentage for ResNet-50 on ILSVRC'12.

	Type (S/U)	Top-1 Base Accuracy (%)	Top-1 Final Accuracy (%)	Top-5 Base Accuracy (%)	Top-5 Final Accuracy (%)	CR (×)	FLOPs (%)
Luo et al. [138]	S	72.88	72.04 (-0.84)	91.14	90.67 (-0.47)	1.51	63.21
	S	72.88	71.01 (-1.87)	91.14	90.02 (-1.12)	2.06	44.17
	S	72.88	68.42 (-4.46)	91.14	88.30 (-2.84)	2.95	28.5
He et al. [139]	S			92.20	90.80 (-1.40)		∼50
Zhuang et al. [141]	S	76.01	74.95 (-1.06)	92.93	92.32 (-0.61)	2.06	44.44
Lin et al. [143]	S	75.13	72.61 (-2.52)	92.30	91.05 (-1.25)		58.03
	S	75.13	71.89 (-3.24)	92.30	90.71 (-1.59)		48.70
	S	75.13	70.93 (-4.20)	92.30	90.14 (-2.16)		40.67
Yu et al. [144]	S	72.88	72.67 (-0.21)			1.37	72.69
	S	72.88	71.99 (-0.89)			1.78	55.99
He et al. [145]	S	76.15	74.61 (-1.54)	92.87	92.06 (-0.81)		58.20
He et al. [30]	S	76.15	74.83 (-1.32)	92.87	92.32 (-0.55)		46.50
Xiao et al. [29]	S	74.90	74.50 (-0.40)			2.20	
Kang & Han [148]	S	75.89	75.27 (-0.62)	92.98	92.30 (-0.68)		45.7
Wang et al. [149]	S	76.13	75.76 (-0.37)	92.86	92.67 (-0.19)		55.9
	S	76.13	75.11 (-1.02)	92.86	92.35 (-0.51)		44.9
Shortened	S*	75.09	73.32 (-1.77)	92.23	91.48 (-0.75)	**7.98**	**25.87**
Superposition	S*	75.09	73.34 (-1.75)	92.23	91.22 (-1.01)	**7.98**	**25.87**
Two-Dimensional	S*	75.09	73.17 (-1.92)	92.23	91.52 (-0.71)	7.64	33.65

Table 12.6.[23] The higher parameter compression rates ($\sim$29.5$\times$ in comparison with $\sim$6.3$\times$ in [159]) and the superior accuracy values clearly demonstrate the significance and the novelty of the proposed training mechanism in forming highly accurate sparse models.

12.7.2 Inference Performance Results

The compact model design is a multi-parametric optimization problem where the accuracy and footprint can be balanced according to application needs. Previous structured pruning studies

[23]The final accuracy and pruning ratio values for the VGG-19 model (with 3×3 convolution) are reported as 94.7% and 84.0% in [159], respectively. The reader will note that the accuracy value of 94.7% is equivalent to $100.00\% - 94.70\% = 5.30\%$ final classification error in Table 12.6. Similarly, the compressed model contains $100.0\% - 84.0\% = 16.0\%$ of the original parameters, which corresponds to the $100.0\% \ / \ 16.0\% = 6.25\times$ compression rate shown in Table 12.6.

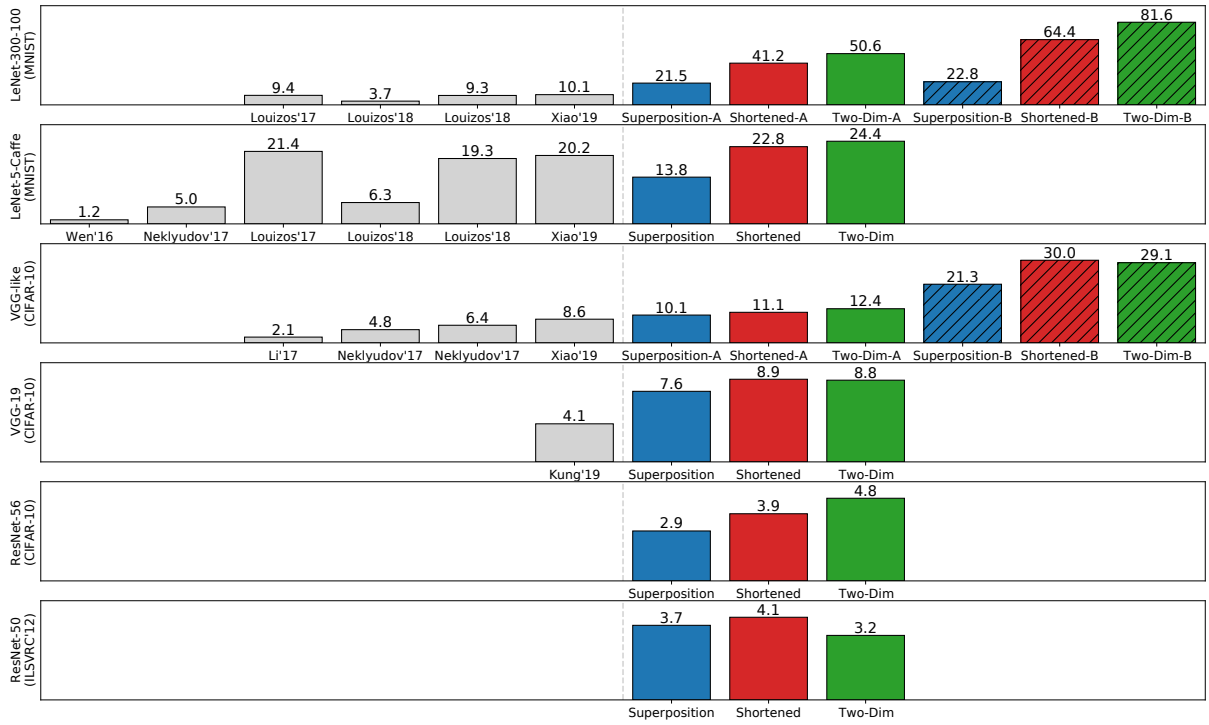

Figure 12.7. Inference speed-up ($\times$) comparison for the complementary sparsity patterns.

have emphasized accuracy, delivering no more than moderate compression rates and FLOP reductions. While we prioritize a competitive accuracy, we aim at substantial parameter and FLOP reductions as well, thus delivering outstanding performance improvements shown in Figure 12.7. Inference performance can be boosted up to 81.6$\times$ for the fully connected and 30.0$\times$ for the convolutional neural networks, and even the challenging ResNet-50 benchmark can be accelerated 4.1$\times$ when compared to the dense model.

The obtained practical speedup depends not only on the final FLOP count but also on the compressed layer shapes. We often observe better hardware utilization and higher speedups in the case of weight matrices with balanced dimensions. For instance, four-dimensional convolutional layer weights are often mapped into the systolic array by first flattening into a two-dimensional matrix. The input dimension may grow significantly larger than the output dimension in the flattened matrix since the input dimension is constructed by merging two spatial dimensions and the input channel dimension of the convolutional layer weights. The shortened neurons/filters

scheme is observed to be consistently more effective than neuron/filter superposition in reducing latency as it delivers compression in the large input dimension of the weight matrix and improves hardware utilization when more balanced compressed weight matrices are tiled and mapped into the systolic array. To illustrate, an interesting data point can be observed in the case of the LeNet-5-Caffe benchmark, where the compressed model obtained through the shortened neurons/filters contains $\times 3.51$ more FLOPs, yet it exhibits $\times 1.65$ higher performance compared to neuron/filter superposition due to better hardware utilization.

It is to be noted that a reduction in a single dimension may not deliver balanced matrices for the more aggressive compression cases or if the original weight matrix is already balanced (e.g., fully connected layers). Under these circumstances, reducing both dimensions through two-dimensional techniques results in balanced weight matrices and improves hardware utilization, as exemplified by the highest practical speedups across various benchmarks in Figure 12.7.

The performance benefits of the proposed sparsity patterns are observed to be significantly higher than the compressed models obtained in prior structured pruning studies consistent with our superior parameter and FLOP compression rates. To illustrate, we deliver up to $\sim 8.1\times$ and $\sim 3.5\times$ higher performance for the LeNet-300-100 and VGG-like models when compared to the best previous result. Furthermore, the superior compression rate that is fueled by our novel training approach leads to a $\sim 2.2\times$ better performance in VGG-19 for the shortened neurons/filters when compared to [159] despite the structural similarity of these sparsity patterns.

12.7.3 Overhead Results for Flow-Controlling MAC Units

Table 12.9 demonstrates the area and power overhead percentage of the flow control enhancements (Figure 12.6) with group sizes of 8, 16, 64, and 256 for 8-bit fixed-point and 32-bit floating-point MAC units. We implement two-dimensional MAC unit configurations for group sizes of 16, 64, and 256 by balancing the input and output dimension sizes.

The experimental results demonstrate overhead percentages that are muted for 32-bit floating-point MAC units since the area and power consumptions of 32-bit floating-point units

Table 12.9. Area and power overheads (%) of the flow control enhancements.

	8-bit Fixed Point		32-bit Floating Point	
	Area	**Power**	**Area**	**Power**
Shortened-8	14.63	14.06	3.38	2.89
Superposition-8	111.68	95.85	10.89	5.42
Shortened-16	32.04	19.59	7.35	4.67
Superposition-16	229.68	192.63	21.80	10.61
Two-Dim-4x4	63.73	65.67	6.86	6.53
Shortened-64	125.36	36.87	28.81	8.68
Superposition-64	932.12	609.68	87.57	31.90
Two-Dim-8x8	126.30	111.75	14.27	9.42
Shortened-256	487.00	91.71	114.57	16.17
Superposition-256	3628.66	2694.93	349.60	133.90
Two-Dim-16x16	261.72	213.36	29.15	16.47

significantly exceed those of their 8-bit fixed-point counterparts. We observe that among the single-dimensional configurations the shortened neurons/filters scheme is consistently more hardware efficient than neuron/filter superposition because of the outlined reasons in Section 12.5.2. A group size of 8 in the shortened neurons/filters scheme can support sparsity rates up to 87.5%, while the multiplexing area and power overheads are contained to 14.63% and 14.06% for the fixed-point, and to the rather minute levels of 3.38% and 2.89% for the floating-point MAC units. The area and power overheads become noticeable for single-dimensional schemes as the group size grows, particularly for the fixed-point case, yet some single-dimensional configurations could still be preferable when throughput improvement is desired.

The power footprint often scales better than area for large group sizes since multiplexer configurations that are controlled by additional weight bits become highly stable in a weight-stationary architecture; thus, the additional circuitry triggers minimal switching activity while transferring the correct operands to MAC units.

The neuron/filter superposition technique may suffer from additional storage overheads due to the required accumulation registers between neighboring MAC units. On the other hand,

the additional input register overheads in the shortened neurons/filters scheme can be amortized for each systolic array row as activations can be broadcast to the entire row without requiring an individual activation register at each MAC unit.

Two-dimensional MAC unit configurations require both input and relatively more expensive output multiplexing circuitry. However, the utilization of both dimensions delivers improved scalability characteristics to large group sizes, consistent with the theoretical expectations outlined in Section 12.5.2. While the area overhead grows linearly with the group size in the single-dimensional designs, the area overhead scaling in the two-dimensional configurations is observed to be proportional to the square root of the group size increase. The two-dimensional configurations are more expensive than the shortened neurons/filters in terms of power in the listed group sizes due to the inefficiencies of output multiplexing. However, power overhead growth is observed to be smaller than the single-dimensional designs as group size is increased.

We have evaluated the area and power overheads of the flow-controlling MAC units in Table 12.9. The flow-controlling MAC unit can provide a throughput increase that matches the supported group size (e.g., 16× throughput boost for the group size of 16). It is therefore helpful to demonstrate the efficiency improvements of the flow-controlling MAC units as in Table 12.10 by evaluating the number of effective MAC operations per unit area and power when compared to the baseline MAC unit in a dense DNN accelerator. These metrics are obtained by normalizing the throughput improvement with the final area and power of the flow-controlling MAC unit.

As a dense neural network accelerator suffers from an inherent inefficiency due to its inability to exploit sparsity, the computational efficiency of the flow-controlling MAC units improves monotonically with increasing group size and sparsity rates. For instance, a floating-point MAC unit for the shortened neurons/filters scheme with a group size of 16 misses the 16× ideal efficiency improvement goal by a tad, undershooting slightly at 14.90× for ops/area and 15.29× for ops/power due to the overheads of the multiplexing logic. In other words, the throughput boost in these designs is accompanied by a relatively smaller hardware overhead growth, thus enabling us to squeeze a larger amount of effective computations into a unit of

Table 12.10. Efficiency improvements ($\times$) for the flow-controlling MAC units.

	8-bit Fixed Point		32-bit Floating Point	
	Ops/Area	**Ops/Power**	**Ops/Area**	**Ops/Power**
Shortened-8	6.98	7.01	7.74	7.78
Superposition-8	3.78	4.08	7.21	7.59
Shortened-16	12.12	13.38	14.90	15.29
Superposition-16	4.85	5.47	13.14	14.47
Two-Dim-4x4	9.77	9.66	14.97	15.02
Shortened-64	28.40	46.76	49.68	58.89
Superposition-64	6.20	9.02	34.12	48.52
Two-Dim-8x8	28.28	30.22	56.01	58.49
Shortened-256	43.61	133.54	119.31	220.36
Superposition-256	6.87	9.16	56.94	109.45
Two-Dim-16x16	70.77	81.69	198.22	219.80

hardware area or power, and improve accelerator efficiency in the sparse computations.

Simultaneous improvements in both performance and efficiency are feasible since the proposed hardware enhancements enable the utilization of computational sparsity that the dense accelerators cannot exploit. One could prefer to constrain the accelerator design by a power budget; in that case, the designs with these enhancements would exhibit higher performance than a dense accelerator in sparse computations. Alternatively, one could aim for a target performance metric; then, the power consumption of the proposed designs will be much lower than their dense counterparts due to their improved efficiency.

Finally, the trends between different sparsity schemes in Table 12.9 generalize to the efficiency results in Table 12.10 as well, such as the superiority of the shortened neurons/filters when compared to the neuron/filter superposition and the relatively more graceful scaling characteristics of the two-dimensional configurations compared to the single-dimensional designs.

12.8 Chapter Summary

Sparsity is regarded as a valuable tool to restrain the ever-escalating computational complexity of deep neural networks, yet in practice, the delivered benefits often fall short of expectations due to the hardware challenges induced by the irregular nature of sparsity. We illustrate that the plasticity of neural networks facilitates the attainment of state-of-the-art accuracy levels when expressiveness is carefully architected into the fabrics of regular sparsity patterns and molded through novel training techniques. The regularity of such sparsity patterns opens up avenues for synergistic hardware support through minor enhancements on the existing architectures that specialize in dense matrix operations. The synergistic design paradigms offer an inspiring direction towards enabling sparsity at a low cost in deep learning hardware.

12.9 Acknowledgements

Chapter 12 is a re-organized reprint of the material as it appears in Elbruz Ozen and Alex Orailoglu, "Unleashing the Potential of Sparse DNNs Through Synergistic Hardware-Sparsity Co-Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 4, pp. 1147–1160, 2023 ([10]). The initial conference version of this manuscript is published in Elbruz Ozen and Alex Orailoglu, "Evolving Complementary Sparsity Patterns for Hardware-Friendly Inference of Sparse DNNs," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE/ACM, 2021 ([11]). The dissertation author was the primary investigator and author of both papers.

Chapter 13

Discussion

13.1 Summary of Technical Chapters

The technical chapters of this dissertation have focused on various practical applications for the advocated algorithm-centric and synergistic design paradigms:

- We have outlined several innovative techniques for hardware error detection and localization in deep neural network accelerators. Instead of precise yet expensive conventional error detection methods, we have explored approaches to identify and locate critical hardware errors with high precision via computational invariants embedded into deep neural networks. Invariant embedding is performed during the architectural definition phase or while guiding deep neural networks in the training process.

- We have presented novel error rectification methods through dropping, clipping, or filtering deep neural network variables that are contaminated with error effects, and experimentally demonstrate the effectiveness of these solutions in providing a graceful accuracy degradation curve under hardware errors. The proposed techniques contain the error impact effectively by harnessing the inherent resilience of deep neural networks to limited magnitude perturbations and complementing their graceful toleration to sparsity.

- We have seamlessly coupled the outlined algorithmic techniques with minimal yet synergistic hardware enhancements in deep neural network accelerators to carry out the required

error localization and rectification tasks in an efficient manner.

- Our experimental observations have opened up future possibilities for novel algorithmic design techniques capable of providing strong hardware error resilience characteristics in deep neural networks. We have demonstrated that the resilience of deep neural networks can be boosted significantly and at no additional cost through the proper containment of the hardware range and strategic manipulations in the numerical distribution of deep neural network variables.

- We have described a practical hardware/software co-design methodology for manufacturing yield improvements in deep neural network accelerators by building upon a low-cost hardware bypass mechanism powered through the micro-architectural regularity of deep learning accelerators and boosting the decentralization of deep neural networks through innovative training techniques. Moreover, we calibrate deep neural networks on the deployed device through computational rearrangements and the adjustment of the batch normalization layers to boost accuracy even further.

- We have investigated various data analysis measures to reveal functional unit correlations and proposed a methodology to eliminate correlation-based redundancy in deep neural networks. As a result, we have obtained compact deep neural network models that can be run efficiently on commodity hardware platforms.

- We have demonstrated that the regularity constraints in the sparsity patterns coupled with minimal micro-architectural enhancements to deep neural network accelerators can establish an effective contract between hardware and software. As a result, we have further increased the synergy between sparse deep neural networks and hardware platforms, eliminated the hardware challenges of unstructured sparsity, and thus delivered practical performance and efficiency boosts in sparse deep neural network inference.

13.2 Significance of Dissertation Research

This dissertation advocates a holistic perspective that considers the characteristics of deep learning algorithms and hardware architectures at the same time. The insights obtained from this holistic approach are utilized for addressing the hardware challenges of machine intelligence platforms. The algorithmic characteristics of deep neural networks, such as plasticity, resiliency, and redundancy, and the hardware platform properties, such as micro-architectural regularity, are translated into significant improvements in hardware safety, reliability, and resource efficiency through a variety of novel design strategies.

One unique perspective brought to the table by this dissertation is the proactive design strategies to deliver an increased level of immunity against hardware error effects. This perspective reduces the need for costly safety and reliability measures in practical hardware systems and encourages the construction of innovative design principles.

Conventional measures necessitate excessive information redundancy and overheads in hardware systems for identifying anomalous behavior. We construct relationships and invariants across deep neural network variables and utilize them to detect hardware error effects in deep neural network computations. The outlined approach not only reduces the overheads required for error detection but also presents a unique perspective and further avenues in the research domain.

Similarly, addressing hardware errors through perfect restoration requires an even more significant amount of information redundancy. Instead, we present a cost-effective solution to minimize the numerical impact of hardware errors on overall functional accuracy. The success of the proposed approach stems from the effective use of deep neural network resiliency to small perturbations while addressing the numerical effect of hardware errors.

As another significant contribution, we employ embedded relationships for constructing regularity within sparse neural networks and boost performance and hardware efficiency by forming an effective contract between deep neural networks and hardware micro-architectures. The result of our study demonstrates the potential of reshaping deep learning algorithms with

alternative training objectives and motivates the synergistic design of deep learning processing systems.

Embedding auxiliary objectives and relationships in deep neural networks incurs additional costs. An essential consideration in this dissertation involves the minimization of such costs through the practical use of deep neural network characteristics, especially their redundancy and plasticity in the training process, while forming computational structures associated with auxiliary safety, reliability, and efficiency objectives.

The proposed methodology in this dissertation promotes strong safety and reliability for deep learning hardware in the face of not only the rare occurrence of hardware errors but even the highly elevated levels that are assumed to be unmanageable in conventional computing systems. As a result, strong functional safety characteristics could be attained in mission-critical domains without compromising the overall functional accuracy while minimizing the associated costs and overheads through the relaxation of structural hardware correctness requirements.

The outlined research has the potential to improve the resource efficiency of deep learning processing systems through effective redundancy extraction and synergistic hardware-software co-design. As more comprehensive redundancy extraction techniques can obtain smaller yet accurate deep neural networks, the sparsity patterns can be engineered with specific constraints to exhibit both high expressiveness and seamless hardware predictability. Efficient deep neural networks obtained through these techniques could position more powerful deep learning techniques in resource-constrained application domains and consequently boost the pace of digital transformation by powering novel machine intelligence applications.

The immediate outcomes of the dissertation will promote the deployment of deep learning processing systems in mission-critical and resource-constrained application domains, including but not limited to autonomous driving, healthcare devices, defense, and industrial systems. The potential benefits could contribute to the lives of millions through safe, reliable, and efficient artificial intelligence hardware systems.

13.3 Open Questions and Future Directions

The technical chapters of the dissertation demonstrate tremendous potential for innovation in challenging artificial intelligence hardware problems. Several research questions and practical considerations remain unanswered, and we believe that the research domain could benefit from further investigation in these areas to ensure the applicability of these concepts in practical hardware systems.

13.3.1 Comprehensive Characterization of DNN Redundancy

A plethora of model compression techniques have been proposed in the recent literature to eliminate redundancy in deep neural networks. While these studies target different types of redundancy and exhibit varying levels of success in the elimination process, the current research literature still lacks a comprehensive understanding of the functionally essential components and the extent of the redundancy in deep neural networks.

Comprehensive methodologies for understanding the extent of redundancy in deep neural networks have immense importance in constructing efficient deep neural network architectures that can satisfy functional requirements at the cost of minimal resources. Furthermore, the proposed approach of reshaping deep neural networks for hardware safety and reliability objectives necessitates additional resources within the model that are not necessarily associated with the primary functional goal. The advocated methodology of reshaping deep neural networks thus can greatly benefit from novel approaches for evaluating the extent of the required additional resources.

13.3.2 Addressing Training Challenges of Proposed DNN Constraints

Section 5.4.2 has previously discussed various training challenges for deep neural networks such as discontinuities and imperfectly aligned objectives when trained with additional constraints. Constructing balanced training objectives, resolving relationship conflicts across

variables, and ensuring efficient gradient propagation are observed to be essential to ensure the viability of effective deep neural network training. We have proposed strategies for enhancing the conventional training flows, such as custom gradient estimation techniques, to address these challenges and obtain competitive functional accuracy in a variety of deep learning benchmarks.

On the other hand, standard training techniques that solely focus on loss function minimization through parameter updates in the direction of gradients could result in locally optimal steps in the presence of discontinuities and imperfectly aligned objectives across deep neural network variables. As a result, further innovative training techniques that are cognizant of discontinuities and relationships in the optimization process could greatly help the invariant embedding process for the goals of safety, reliability, and efficiency, and thus ensure that competitive functional accuracy is not sacrificed in challenging deep learning benchmarks due to the outlined training complexities when deep neural networks are subject to additional constraints.

13.3.3 Exploring Interactions Between Proposed Techniques

We have presented various technical studies throughout this dissertation to address hardware safety, reliability, and resource-efficiency challenges. Meanwhile, these techniques make use of the same characteristics of deep neural networks, and they could result in further interactions when they are deployed simultaneously in the same deep learning model.

To illustrate, the proposed safety and reliability measures could interfere with resource-efficiency techniques, and the impact of such interaction could vary depending on the scenario. For example, the introduced numerical order relationships across the activation variables in Chapter 8 could incur dependencies and lead to difficulties for the structured neuron/filter pruning in Chapter 11, yet we expect weight sparsity patterns in Chapter 12 not to be significantly affected by the constructed relationships across the activations.

As the outcome of the interaction could differ depending on the involved techniques, further research efforts are necessary to ensure the viability of the simultaneous application of methods in practical deep learning processing systems.

13.3.4 Harnessing DNN Resilience for Further Efficiency Improvements

The outline approach in this dissertation enables deep neural networks to endure severe error conditions at a minimal cost. Such resilience characteristics are invaluable not only for hardware safety and reliability but also open up a slew of opportunities for the next generation of embedded designs in deep neural network processing. The relaxation in the rigidity of the individual correctness requirements in the hardware can be exploited to reduce the resource footprint through design optimizations or utilizing alternative manufacturing technologies that suffer from inherent variability and may be prone to high defect rates.

The aggressive hardware optimizations can deliver remarkable energy efficiency improvements in CMOS (complementary metal-oxide-semiconductor) devices when the consequently elevated hardware error rates can be tolerated through the enhanced resilience characteristics of deep neural networks. Such aggressive hardware optimizations include supply voltage scaling in the on-chip SRAM (static random-access memory) buffers [103], computational fabric [78], or the interconnect circuitry [172].

Furthermore, the error resilience of deep neural networks can facilitate novel computation paradigms [175], and novel communication fabrics through photonics, wireless, or 3D interconnects [234] when the inherent inaccuracy and manufacturing issues for such emerging technologies can be endured through the algorithmic resilience of deep neural networks.

More comprehensive hardware studies, as well as the implementation of automated design tools, are necessary for the precise characterization of computational mediums with high fault rates so that utmost performance and energy efficiency goals can be attained with guaranteed operational accuracy.

13.3.5 Effective and Comprehensive Evaluation of DNN Reliability

The benefits of conventional fault tolerance methods have been relatively straightforward to characterize and quantify due to their adherence to the strict structural correctness requirements.

To illustrate, error correction codes such as Hamming codes [52, 89] offer guaranteed correction and detection of errors within a given limit. The reliability of modular redundant systems [53, 90] can be characterized analytically, as an example is provided in Section 8.8.1.

On the other hand, comprehensive evaluation of DNN reliability is a more involved pursuit due to the inherent resiliency and the statistical nature of deep neural networks. A variety of techniques in Section 3.1.1 have attempted to close this gap, often through empirical methods such as fault simulation, yet the accuracy and scalability challenges of such tools remain unresolved. The reader will note the similarity of the deep neural network hardware error resilience problem to certain phenomena investigated in the digital signal processing domain, as discussed in Chapter 9. The theoretical background of the digital signal processing domain could be beneficial, yet significant research efforts are likely needed to bridge the gap and construct a sound theoretical infrastructure for reliability evaluation in deep neural networks.

More effective and comprehensive reliability evaluation techniques are critical in the near term to assure the practical applicability of the proposed hardware safety and reliability techniques into mission-critical domains such as automotive electronics with strict safety requirements [51, 235]. The reliability evaluation techniques will not only help the system designers to identify potential problems but also guide the researchers to explore more effective avenues for boosting the safety and reliability of deep learning hardware.

13.3.6 Applicability to Future Algorithms and Hardware Technologies

An important consideration is the applicability of the proposed techniques to the evolving algorithms in the fast-evolving domain of artificial intelligence. Furthermore, the actively researched device technologies may drastically alter the landscape of machine intelligence hardware in the next decade. It is therefore critical to question whether the proposed framework in this dissertation could withstand the pressure of time, and if so, we further need to examine the impact of the additional constraints that may emerge in the future.

Throughout the dissertation, we make use of the unique algorithmic characteristics of

deep neural networks, including inherent redundancy, plasticity in the design process, and resiliency to small numerical perturbations. On the other hand, the evolving landscape of deep learning algorithms could impact the wide-scale availability of these characteristics. The continuous evaluation of these characteristics is thus needed for the state-of-the-art deep neural network architectures to be able to make the necessary revisions and reveal further research opportunities.

The novel device technologies, such as in-memory or analog domain processing, may lead to fundamental shifts in how we approach deep learning hardware problems. For instance, the safety and reliability techniques could need to adapt to unique hardware fault models in novel device technologies. The additional inaccuracy challenges may prove to be challenging to address with conventional measures and thus accentuate the need for algorithm-centric and synergistic design paradigms even further. Overall, the alternative strategy of considering deep learning hardware from a statistical and functional perspective can furnish innovative answers to existing hardware challenges and facilitate the seamless adaptation of future artificial intelligence hardware technologies.

13.3.7 Promoting Synergistic Avenues in DNN Processing System Design

Defining clear abstraction levels between hardware and software has remained a highly effective strategy in developing computing systems for over half a century. Conventional processor micro-architectures have abstracted themselves from software development through well-defined ISAs (instruction set architectures), as the necessary performance and efficiency improvements are usually delivered by the advancements in semiconductor technology, with the help of well-known phenomena such as Moore's Law [39]. The stagnation in semiconductor technology scaling has led to the inception of hardware accelerators and custom silicon as an alternative solution in the past decade, capitalizing on the innate inefficiencies of the general-purpose architectures to deliver performance and efficiency improvements over conventional processor micro-architectures. However, the long-term effectiveness of this approach is yet

in question due to the excessive scaling trends of large deep neural networks. The described techniques fall short of considering the synergy that can arise when hardware and software are simultaneously optimized, resulting in missed opportunities to maximize benefits.

Foreseeing the next technology wave that can enable the efficient and scalable processing of deep neural networks is undoubtedly a challenging pursuit. The co-design techniques that synergistically optimize software and hardware can play an essential role in the design of the next generation of machine intelligence systems and offer feasible avenues for improving the hardware safety, reliability, and resource efficiency of deep learning processing systems. Despite the strong support for the synergistic co-design techniques in the scope of the dissertation, the adoption of synergistic design principles is still in its infancy; further research, development, and promotion efforts are necessary both in academia and industry to unlock the full potential of synergistic hardware-software co-design techniques for deep learning hardware.

13.4 Acknowledgements

Chapter 13 partially contains material from Elbruz Ozen and Alex Orailoglu, "SNR: Squeezing Numerical Range Defuses Bit Error Vulnerability Surface in Deep Neural Networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021 ([7]).

Chapter 14

Conclusion

The past decade has witnessed remarkable progress in artificial intelligence. The recent advancements have led to numerous practical use cases in various critical domains such as autonomous driving, healthcare, and industrial automation. As the role of artificial intelligence keeps expanding rapidly, our critical cyber infrastructure will heavily rely on intelligent systems in the near future.

Deep learning hardware systems are hindered by various challenges in practice, including the elevated computational cost of deep neural networks and the increasing concerns about hardware safety and reliability in the mission-critical application domains. The computational cost of deep neural networks has been traditionally addressed through isolated hardware and software optimizations in the recent past. Meanwhile, conventional hardware safety and reliability solutions developed for general-purpose electronics often lead to prohibitive area, power, or performance overheads.

This dissertation advocates for algorithm-centric and synergistic co-design techniques and it explores outside-of-the-box solutions to the demanding hardware challenges of deep learning processing systems. The innovative approach of this dissertation stems from the distinctive landscape of deep learning processing systems when compared to general-purpose computing platforms. We investigate unique computational characteristics of deep neural networks, including their redundancy, algorithmic plasticity, and resiliency to small numerical

perturbations. At the hardware level, the unique micro-architectural characteristics of deep learning accelerators, such as regularity, are considered.

The proposed approach in this dissertation relies on four fundamental foundations: reshaping deep neural networks in the training process for the desired goals, enhancing hardware platforms minimally yet strategically at the cost of minimal resources, prioritizing the functional correctness and the overall goal rather than the structural correctness of individual variables and operations, and finally harnessing the statistical nature of deep learning algorithms to relax the strict preciseness requirements and innovate cost-effective solutions.

We demonstrate various practical applications for the outlined principles throughout this dissertation. On the hardware safety and reliability side, we first propose low-cost methods for detecting hardware errors in deep neural network accelerators through the use of embedded algorithmic invariants. Novel error rectification methodologies are presented for curbing the impact of critical errors on deep neural networks. We further promote accurate deep neural network operation even under extreme hardware error rates through proactive design and training strategies. Finally, we tackle permanent hardware defects in deep neural network accelerators through cost-effective adaptivity techniques. To address the hardware resource efficiency challenges, we demonstrate and systematically eliminate unique redundancy types that stem from functional correlations in deep neural networks. Furthermore, we significantly boost the inference performance of sparse deep neural networks by defining regularity constraints in sparsity patterns and strategically enhancing hardware platforms.

Overall, the outlined effective strategies share the common attribute of the holistic consideration of hardware and software, which yields insights that translate into innovative solutions. A variety of challenging hardware problems of deep learning processing systems can thus be addressed effectively through the use of the proposed innovative design principles. The outlined strategies in this dissertation open up promising avenues for building safe, reliable, and resource-efficient hardware systems for machine intelligence.

Bibliography

[1] E. Ozen and A. Orailoglu, "Low-cost error detection in deep neural network accelerators with linear algorithmic checksums," *Journal of Electronic Testing (JETTA)*, vol. 36, no. 6, pp. 703–718, 2020.

[2] E. Ozen and A. Orailoglu, "Sanity-Check: Boosting the reliability of safety-critical deep neural network applications," in *Proceedings of the 28th Asian Test Symposium (ATS)*, pp. 7–12, IEEE, 2019.

[3] E. Ozen and A. Orailoglu, "Concurrent monitoring of operational health in neural networks through balanced output partitions," in *Proceedings of the 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 169–174, IEEE, 2020.

[4] E. Ozen and A. Orailoglu, "Shaping resilient AI hardware through DNN computational feature exploitation," *IEEE Design & Test (D&T)*, vol. 40, no. 2, pp. 59–66, 2023.

[5] E. Ozen and A. Orailoglu, "Boosting bit-error resilience of DNN accelerators through median feature selection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 39, no. 11, pp. 3250–3262, 2020.

[6] E. Ozen and A. Orailoglu, "Just Say Zero: Containing critical bit-error propagation in deep neural networks with anomalous feature suppression," in *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, IEEE/ACM, 2020.

[7] E. Ozen and A. Orailoglu, "SNR: Squeezing numerical range defuses bit error vulnerability surface in deep neural networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5s, pp. 1–25, 2021.

[8] E. Ozen and A. Orailoglu, "Architecting decentralization and customizability in DNN accelerators for hardware defect adaptation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 11, pp. 3934–3945, 2022.

[9] E. Ozen and A. Orailoglu, "Squeezing correlated neurons for resource-efficient deep neural networks," in *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML-PKDD) - Part II*, pp. 52–68, Springer, 2021.

[10] E. Ozen and A. Orailoglu, "Unleashing the potential of sparse DNNs through synergistic hardware-sparsity co-design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 4, pp. 1147–1160, 2023.

[11] E. Ozen and A. Orailoglu, "Evolving complementary sparsity patterns for hardware-friendly inference of sparse DNNs," in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8, IEEE/ACM, 2021.

[12] "Software 2.0." https://karpathy.medium.com/software-2-0-a64152b37c35. Accessed: Apr-4-2023.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[14] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proceedings of the European Conference on Computer Vision*, pp. 21–37, Springer, 2016.

[15] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[16] Z. Zhang, J. Geiger, J. Pohjalainen, A. E.-D. Mousa, W. Jin, and B. Schuller, "Deep learning for environmentally robust speech recognition: An overview of recent developments," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 9, no. 5, pp. 1–28, 2018.

[17] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang, Q. Liang, D. Bhatia, Y. Shangguan, B. Li, G. Pundak, K. C. Sim, T. Bagby, S.-y. Chang, K. Rao, and A. Gruenstein, "Streaming end-to-end speech recognition for mobile devices," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6381–6385, IEEE, 2019.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[19] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *arXiv preprint arXiv:2005.14165*, 2020.

[20] "ChatGPT: Optimizing language models for dialogue." https://openai.com/blog/chatgpt/. Accessed: Apr-4-2023.

[21] "DALL·E 2." https://openai.com/dall-e-2. Accessed: Apr-4-2023.

[22] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "DeepDriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 2722–2730, 2015.

[23] Y. Huang and Y. Chen, "Autonomous driving with deep learning: A survey of state-of-art technologies," *arXiv preprint arXiv:2006.06091*, 2020.

[24] A. Esteva, A. Robicquet, B. Ramsundar, V. Kuleshov, M. DePristo, K. Chou, C. Cui, G. Corrado, S. Thrun, and J. Dean, "A guide to deep learning in healthcare," *Nature Medicine*, vol. 25, no. 1, pp. 24–29, 2019.

[25] P. Villalobos, J. Sevilla, T. Besiroglu, L. Heim, A. Ho, and M. Hobbhahn, "Machine learning model sizes and the parameter gap," *arXiv preprint arXiv:2207.02852*, 2022.

[26] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks," *Synthesis Lectures on Computer Architecture*, vol. 15, no. 2, pp. 1–341, 2020.

[27] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning*, pp. 6105–6114, PMLR, 2019.

[28] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural networks," *Advances in Neural Information Processing Systems*, vol. 28, 2015.

[29] X. Xiao, Z. Wang, and S. Rajasekaran, "AutoPrune: Automatic network pruning by regularizing auxiliary parameters," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[30] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4340–4349, 2019.

[31] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[32] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, "PACT: Parameterized clipping activation for quantized neural networks," *arXiv preprint arXiv:1805.06085*, 2018.

[33] J. Choi, S. Venkataramani, V. V. Srinivasan, K. Gopalakrishnan, Z. Wang, and P. Chuang, "Accurate and efficient 2-bit quantized neural networks," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 348–359, 2019.

[34] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *arXiv preprint arXiv:2103.13630*, 2021.

[35] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[36] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *58th Design Automation Conference (DAC)*, pp. 769–774, ACM/IEEE, 2021.

[37] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to FPGAs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE/ACM, 2016.

[38] G. Pang, "The AI chip race," *IEEE Intelligent Systems*, vol. 37, no. 2, pp. 111–112, 2022.

[39] G. E. Moore, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, no. 1, pp. 82–85, 1998.

[40] "Tiny chips, big headaches: Chip errors are becoming more common and harder to track down." https://www.nytimes.com/2022/02/07/technology/computer-chips-errors.html. Accessed: Apr-4-2023.

[41] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, pp. 9–16, 2021.

[42] "Aging problems at 5nm and below." https://semiengineering.com/aging-problems-at-5nm-and-below. Accessed: Apr-4-2023.

[43] "International Roadmap for Devices and Systems 2021 Edition: More Moore." https://irds.ieee.org/editions/2021. Accessed: Apr-4-2023.

[44] A. J. Strojwas, K. Doong, and D. Ciplickas, "Yield and reliability challenges at 7nm and below," in *Electron Devices Technology and Manufacturing Conference (EDTM)*, pp. 179–181, IEEE, 2019.

[45] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 303–314, ACM, 2018.

[46] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 372–387, 2016.

[47] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 5, pp. 828–841, 2019.

[48] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Towards proving the adversarial robustness of deep neural networks," *arXiv preprint arXiv:1709.02802*, 2017.

[49] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proceedings of the Advances in Neural Information Processing Systems*, pp. 1097–1105, 2012.

[50] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural Networks*, vol. 32, pp. 323–332, 2012.

[51] "ISO 26262-1:2018 road vehicles – functional safety." https://www.iso.org/standard/68383.html, 2018.

[52] H. Shaheen, G. Boschi, G. Harutyunyan, and Y. Zorian, "Advanced ECC solution for automotive SoCs," in *Proceedings of the 23rd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 71–73, IEEE, 2017.

[53] M. Baleani, A. Ferrari, L. Mangeruca, A. Sangiovanni-Vincentelli, M. Peri, and S. Pezzini, "Fault-tolerant platforms for automotive safety-critical applications," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 170–177, ACM, 2003.

[54] I. Lee, M. Basoglu, M. Sullivan, D. H. Yoon, L. Kaplan, and M. Erez, "Survey of error and fault detection mechanisms," tech. rep., University of Texas at Austin, 2012.

[55] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann, 2020.

[56] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[57] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[58] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *Proceedings of the 32nd International Conference on Machine Learning*, pp. 448–456, 2015.

[59] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[60] L. Wan, M. Zeiler, S. Zhang, Y. Le Cun, and R. Fergus, "Regularization of neural networks using DropConnect," in *International Conference on Machine Learning*, pp. 1058–1066, PMLR, 2013.

[61] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[62] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic CNN accelerator simulator," *arXiv:1811.02883*, 2018.

[63] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of DNN accelerators using SCALE-Sim," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 58–68, IEEE, 2020.

[64] M. Shafique, M. Naseer, T. Theocharides, C. Kyrkou, O. Mutlu, L. Orosa, and J. Choi, "Robust machine learning systems: Challenges, current trends, perspectives, and the road ahead," *IEEE Design & Test*, vol. 37, no. 2, pp. 30–57, 2020.

[65] M. Lee, K. Hwang, and W. Sung, "Fault tolerance analysis of digital feed-forward deep neural networks," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5031–5035, IEEE, 2014.

[66] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.

[67] X. Jiao, M. Luo, J.-H. Lin, and R. K. Gupta, "An assessment of vulnerability of hardware neural networks to dynamic voltage and temperature variations," in *International Conference on Computer-Aided Design (ICCAD)*, pp. 945–950, IEEE/ACM, 2017.

[68] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proceedings of the 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[69] M. A. Neggaz, I. Alouani, P. R. Lorenzo, and S. Niar, "A reliability study on CNNs for critical embedded systems," in *Proceedings of the International Conference on Computer Design (ICCD)*, pp. 476–479, IEEE, 2018.

[70] M. A. Neggaz, I. Alouani, S. Niar, and F. Kurdahi, "Are CNNs reliable enough for critical applications? An exploratory study," *IEEE Design & Test*, vol. 37, no. 2, pp. 76–83, 2019.

[71] M. Sabbagh, C. Gongye, Y. Fei, and Y. Wang, "Evaluating fault resiliency of compressed deep neural networks," in *International Conference on Embedded Software and Systems (ICESS)*, pp. 1–7, IEEE, 2019.

[72] Z. Yan, Y. Shi, W. Liao, M. Hashimoto, X. Zhou, and C. Zhuo, "When single event upset meets deep neural networks: Observations, explorations, and remedies," in *25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 163–168, IEEE, 2020.

[73] A. Mahmoud, S. K. S. Hari, C. W. Fletcher, S. V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing selective protection for CNN resilience," in *IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 127–138, IEEE, 2021.

[74] N. Narayanan, Z. Chen, B. Fang, G. Li, K. Pattabiraman, and N. DeBardeleben, "Fault injection for TensorFlow applications," *IEEE Transactions on Dependable and Secure Computing*, 2022. (Early Access).

[75] G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A configurable fault injector for TensorFlow applications," in *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 313–320, 2018.

[76] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "BinFI: An efficient fault injector for safety-critical machine learning systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–23, 2019.

[77] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "PyTorchFI: A runtime perturbation tool for DNNs," in *Proceedings of the Workshop on Dependable and Secure Machine Learning (DSML)*, 2020.

[78] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, "Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators," in *Proceedings of the 55th Annual Design Automation Conference*, pp. 19:1–19:6, ACM, 2018.

[79] J. J. Zhang, T. Gu, K. Basu, and S. Garg, "Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator," in *Proceedings of the IEEE 36th VLSI Test Symposium (VTS)*, pp. 1–6, 2018.

[80] J. J. Zhang, K. Basu, and S. Garg, "Fault-tolerant systolic array based accelerators for deep neural network execution," *IEEE Design & Test*, vol. 36, no. 5, pp. 44–53, 2019.

[81] A. Ruospo, A. Balaara, A. Bosio, and E. Sanchez, "A pipelined multi-level fault injector for deep neural networks," in *International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, IEEE, 2020.

[82] A. Chaudhuri, J. Talukdar, F. Su, and K. Chakrabarty, "Functional criticality classification of structural faults in AI accelerators," in *International Test Conference (ITC)*, pp. 1–5, IEEE, 2020.

[83] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on GPUs," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2018.

[84] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang, "Resiliency of automotive object detection networks on GPU architectures," in *International Test Conference (ITC)*, pp. 1–9, IEEE, 2019.

[85] W. Choi, D. Shin, J. Park, and S. Ghosh, "Sensitivity based error resilient techniques for energy efficient deep neural network accelerators," in *56th Annual Design Automation Conference*, ACM, 2019.

[86] G. Montavon, S. Lapuschkin, A. Binder, W. Samek, and K.-R. Müller, "Explaining nonlinear classification decisions with deep Taylor decomposition," *Pattern Recognition*, vol. 65, pp. 211–222, 2017.

[87] C. Schorn, A. Guntoro, and G. Ascheid, "Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 979–984, IEEE, 2018.

[88] A. Mahmoud, S. K. S. Hari, C. W. Fletcher, S. V. Adve, C. Sakr, N. Shanbhag, P. Molchanov, M. B. Sullivan, T. Tsai, and S. W. Keckler, "HarDNN: Feature map vulnerability evaluation in CNNs," *arXiv preprint arXiv:2002.09786*, 2020.

[89] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[90] P. Bannon, G. Venkataramanan, D. D. Sarma, and E. Talpes, "Computer and redundancy solution for the full self-driving computer," in *IEEE 31st Hot Chips Symposium (HCS)*, 2019.

[91] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: Circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[92] C. Schorn, A. Guntoro, and G. Ascheid, "Efficient on-line error detection and mitigation for deep neural network accelerators," in *Proceedings of the International Conference on Computer Safety, Reliability, and Security*, pp. 205–219, Springer, 2018.

[93] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, 1984.

[94] J. A. Abraham, P. Banerjee, C.-Y. Chen, W. K. Fuchs, S.-Y. Kuo, and A. L. N. Reddy, "Fault tolerance techniques for systolic arrays," *Computer*, vol. 20, no. 7, pp. 65–75, 1987.

[95] Z. Xu and J. Abraham, "Safety design of a convolutional neural network accelerator with error localization and correction," in *International Test Conference (ITC)*, pp. 1–10, IEEE, 2019.

[96] S. K. S. Hari, M. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[97] S. Pandey, S. Banerjee, and A. Chatterjee, "Error resilient neuromorphic networks using checker neurons," in *24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pp. 135–138, IEEE, 2018.

[98] S. Pandey, S. Banerjee, and A. Chatterjee, "ReiNN: Efficient error resilience in artificial neural networks using encoded consistency checks," in *23rd European Test Symposium (ETS)*, pp. 1–2, IEEE, 2018.

[99] L.-H. Hoang, M. A. Hanif, and M. Shafique, "FT-ClipAct: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1241–1246, IEEE, 2020.

[100] Z. Chen, G. Li, and K. Pattabiraman, "A low-cost fault corrector for deep neural networks through range restriction," in *International Conference on Dependable Systems and Networks (DSN)*, IEEE/IFIP, 2021.

[101] J. Zhan, R. Sun, W. Jiang, Y. Jiang, X. Yin, and C. Zhuo, "Improving fault tolerance for reliable DNN using boundary-aware activation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 10, pp. 3414–3425, 2021.

[102] N. Cavagnero, F. Dos Santos, M. Ciccone, G. Averta, T. Tommasi, and P. Rech, "Transient-fault-aware design and training to enhance DNNs reliability with zero-overhead," in *IEEE 28th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 1–7, IEEE, 2022.

[103] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, 2016.

[104] S. Burel, A. Evans, and L. Anghel, "Zero-overhead protection for CNN weights," in *IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, IEEE, 2021.

[105] T. Hirtzlin, M. Bocquet, J.-O. Klein, E. Nowak, E. Vianello, J.-M. Portal, and D. Querlioz, "Outstanding bit error tolerance of resistive RAM-based binarized neural networks," in *International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 288–292, IEEE, 2019.

[106] C. Schorn, T. Elsken, S. Vogel, A. Runge, A. Guntoro, and G. Ascheid, "Automated design of error-resilient and hardware-efficient deep neural networks," *Neural Computing and Applications*, vol. 32, no. 24, pp. 18327–18345, 2020.

[107] W. Li, X. Ning, G. Ge, X. Chen, Y. Wang, and H. Yang, "FTT-NAS: discovering fault-tolerant neural architecture," in *25th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 211–216, IEEE, 2020.

[108] G. B. Hacene, F. Leduc-Primeau, A. B. Soussia, V. Gripon, and F. Gagnon, "Training modern deep neural networks for memory-fault robustness," in *International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, IEEE, 2019.

[109] D. Stutz, N. Chandramoorthy, M. Hein, and B. Schiele, "Bit error robustness for energy-efficient DNN accelerators," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 569–598, 2021.

[110] Y. Sakai, B. U. Pedroni, S. Joshi, A. Akinin, and G. Cauwenberghs, "DropOut and DropConnect for reliable neuromorphic inference under energy and bandwidth constraints in network connectivity," in *International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pp. 76–80, IEEE, 2019.

[111] Y. Sakai, B. U. Pedroni, S. Joshi, S. Tanabe, A. Akinin, and G. Cauwenberghs, "DropOut and DropConnect for reliable neuromorphic inference under communication constraints in network connectivity," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 4, pp. 658–667, 2019.

[112] S. Burel, A. Evans, and L. Anghel, "MOZART: Masking outputs with zeros for architectural robustness and testing of DNN accelerators," in *27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 1–6, IEEE, 2021.

[113] A. Gebregiorgis and M. B. Tahoori, "Testing of neuromorphic circuits: Structural vs functional," in *International Test Conference*, IEEE, 2019.

[114] S. Kundu, S. Banerjee, A. Raha, S. Natarajan, and K. Basu, "Toward functional safety of systolic array-based deep learning hardware accelerators," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 29, no. 3, pp. 485–498, 2021.

[115] F. Meng, F. S. Hosseini, and C. Yang, "A self-test framework for detecting fault-induced accuracy drop in neural network accelerators," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, pp. 722–727, 2021.

[116] A. Chaudhuri, C. Liu, X. Fan, and K. Chakrabarty, "C-testing of AI accelerators," in *29th Asian Test Symposium (ATS)*, pp. 1–6, IEEE, 2020.

[117] C. Liu, C. Chu, D. Xu, Y. Wang, Q. Wang, H. Li, X. Li, and K.-T. Cheng, "HyCA: A hybrid computing architecture for fault-tolerant deep learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 10, pp. 3400–3413, 2021.

[118] M. A. Hanif and M. Shafique, "SalvageDNN: Salvaging deep neural network accelerators with permanent faults through saliency-driven fault-aware mapping," *Philosophical Transactions of the Royal Society A*, vol. 378, no. 2164:20190164, 2020.

[119] M. Sadi and U. Guin, "Test and yield loss reduction of AI and deep learning accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 1, pp. 104–115, 2021.

[120] M. Horowitz, "Computing's energy problem (and what we can do about it)," in *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, IEEE, 2014.

[121] E. H. Lee, D. Miyashita, E. Chai, B. Murmann, and S. S. Wong, "Lognet: Energy-efficient neural networks using logarithmic computation," in *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5900–5904, IEEE, 2017.

[122] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "HAQ: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8612–8620, 2019.

[123] V. Camus, L. Mei, C. Enz, and M. Verhelst, "Review and benchmarking of precision-scalable multiply-accumulate unit architectures for embedded neural-network processing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 4, pp. 697–711, 2019.

[124] Y. LeCun, J. Denker, and S. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems*, vol. 2, pp. 598–605, 1989.

[125] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 1387–1395, 2016.

[126] X. Dong, S. Chen, and S. Pan, "Learning to prune deep neural networks via layer-wise optimal brain surgeon," in *Advances in Neural Information Processing Systems*, vol. 30, pp. 4857–4867, 2017.

[127] K. Ullrich, E. Meeds, and M. Welling, "Soft weight-sharing for neural network compression," *arXiv preprint arXiv:1702.04008*, 2017.

[128] D. Molchanov, A. Ashukha, and D. Vetrov, "Variational dropout sparsifies deep neural networks," in *International Conference on Machine Learning*, pp. 2498–2507, PMLR, 2017.

[129] G. Li, C. Qian, C. Jiang, X. Lu, and K. Tang, "Optimization based layer-wise magnitude-based pruning for DNN compression," in *International Joint Conference on Artificial Intelligence*, pp. 2383–2389, 2018.

[130] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," *Advances in Neural Information Processing Systems*, vol. 29, pp. 2074–2082, 2016.

[131] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, "Network trimming: A data-driven neuron pruning approach towards efficient deep architectures," *arXiv:1607.03250*, 2016.

[132] M. Babaeizadeh, P. Smaragdis, and R. H. Campbell, "NoiseOut: a simple way to prune neural networks," *arXiv:1611.06211*, 2016.

[133] Z. Mariet and S. Sra, "Diversity networks," in *International Conference on Learning Representations*, 2016.

[134] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *International Conference on Learning Representations*, 2017.

[135] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," in *International Conference on Learning Representations*, 2017.

[136] K. Neklyudov, D. Molchanov, A. Ashukha, and D. P. Vetrov, "Structured Bayesian pruning via log-normal multiplicative noise," in *Advances in Neural Information Processing Systems*, pp. 6775–6784, 2017.

[137] C. Louizos, K. Ullrich, and M. Welling, "Bayesian compression for deep learning," in *Advances in Neural Information Processing Systems*, pp. 3288–3298, 2017.

[138] J.-H. Luo, J. Wu, and W. Lin, "ThiNet: A filter level pruning method for deep neural network compression," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 5058–5066, 2017.

[139] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1389–1397, 2017.

[140] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through L_0 regularization," *arXiv:1712.01312*, 2018.

[141] Z. Zhuang, M. Tan, B. Zhuang, J. Liu, Y. Guo, Q. Wu, J. Huang, and J. Zhu, "Discrimination-aware channel pruning for deep neural networks," in *Advances in Neural Information Processing Systems*, pp. 875–886, 2018.

[142] X. Zhu, W. Zhou, and H. Li, "Improving deep neural network sparsity through decorrelation regularization," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pp. 3264–3270, 2018.

[143] S. Lin, R. Ji, Y. Li, Y. Wu, F. Huang, and B. Zhang, "Accelerating convolutional networks via global & dynamic filter pruning," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pp. 2425–2432, 2018.

[144] R. Yu, A. Li, C.-F. Chen, J.-H. Lai, V. I. Morariu, X. Han, M. Gao, C.-Y. Lin, and L. S. Davis, "NISP: Pruning networks using neuron importance score propagation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 9194–9203, 2018.

[145] Y. He, G. Kang, X. Dong, Y. Fu, and Y. Yang, "Soft filter pruning for accelerating deep convolutional neural networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pp. 2234–2240, 2018.

[146] B. O. Ayinde and J. M. Zurada, "Building efficient ConvNets using redundant feature pruning," *arXiv:1802.07653*, 2018.

[147] Z. You, K. Yan, J. Ye, M. Ma, and P. Wang, "Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, pp. 2130–2141, 2019.

[148] M. Kang and B. Han, "Operation-aware soft channel pruning using differentiable masks," in *International Conference on Machine Learning*, pp. 5122–5131, PMLR, 2020.

[149] Z. Wang, C. Li, and X. Wang, "Convolutional neural network pruning with structural redundancy reduction," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 14913–14922, 2021.

[150] J. B. Hough, M. Krishnapur, Y. Peres, and B. Virág, "Determinantal processes and independence," *Probability Surveys*, vol. 3, pp. 206–229, 2006.

[151] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *International Conference on Learning Representations*, 2019.

[152] S. Narang, E. Undersander, and G. Diamos, "Block-sparse recurrent neural networks," *arXiv preprint arXiv:1711.02782*, 2017.

[153] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," *arXiv preprint arXiv:1705.08922*, 2017.

[154] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient DNN inference on GPU," in *AAAI Conference on Artificial Intelligence*, pp. 5676–5683, 2019.

[155] H.-J. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 30, no. 7, pp. 2093–2103, 2019.

[156] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Sparse systolic tensor array for efficient CNN hardware acceleration," *arXiv:2009.02381*, 2020.

[157] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 34–37, 2020.

[158] J. Yoon and S. J. Hwang, "Combined group and exclusive sparsity for deep neural networks," in *International Conference on Machine Learning*, pp. 3958–3966, PMLR, 2017.

[159] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 821–834, 2019.

[160] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge, "Sparse-TPU: Adapting systolic arrays for sparse matrices," in *ACM International Conference on Supercomputing*, pp. 1–12, 2020.

[161] R. Girshick, "Fast R-CNN," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1440–1448, 2015.

[162] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," in *International Conference on Learning Representations*, 2016.

[163] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, "Efficient and accurate approximations of nonlinear convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1984–1992, 2015.

[164] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[165] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "DaDianNao: A machine-learning supercomputer," in *47th Annual International Symposium on Microarchitecture*, pp. 609–622, IEEE/ACM, 2014.

[166] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44-3, pp. 367–379, 2016.

[167] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.

[168] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[169] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *International Symposium on Microarchitecture*, pp. 1–12, IEEE, 2016.

[170] J. P. Kulkarni, C. Tokunaga, P. A. Aseron, T. Nguyen, C. Augustine, J. W. Tschanz, and V. De, "A 409 GOPS/W adaptive and resilient domino register file in 22 nm tri-gate CMOS featuring in-situ timing margin and error detection for tolerance to within-die variation, voltage drop, temperature and aging," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 117–129, 2015.

[171] N. Chandramoorthy, K. Swaminathan, M. Cochet, A. Paidimarri, S. Eldridge, R. V. Joshi, M. M. Ziegler, A. Buyuktosunoglu, and P. Bose, "Resilient low voltage accelerators for high energy efficiency," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 147–158, IEEE, 2019.

[172] A. Mineo, M. Palesi, G. Ascia, P. P. Pande, and V. Catania, "On-chip communication energy reduction through reliability aware adaptive voltage swing scaling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 11, pp. 1769–1782, 2016.

[173] P. Pandey, P. Basu, K. Chakraborty, and S. Roy, "GreenTPU: Improving timing error resilience of a near-threshold tensor processing unit," in *56th Annual Design Automation Conference*, pp. 1–6, ACM/IEEE, 2019.

[174] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 27–39, 2016.

[175] S. Yu, "Neuro-inspired computing with emerging nonvolatile memory," *Proceedings of the IEEE*, vol. 106, no. 2, pp. 260–285, 2018.

[176] Y. Zhang, Z. Jia, Y. Pan, H. Du, Z. Shen, M. Zhao, and Z. Shao, "PattPIM: A practical ReRAM-based DNN accelerator by reusing weight pattern repetition," in *57th Design Automation Conference (DAC)*, pp. 1–6, ACM/IEEE, 2020.

[177] B. Feinberg, S. Wang, and E. Ipek, "Making memristive neural network accelerators reliable," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 52–65, IEEE, 2018.

[178] T. Liu, W. Wen, L. Jiang, Y. Wang, C. Yang, and G. Quan, "A fault-tolerant neural network architecture," in *56th Design Automation Conference (DAC)*, pp. 1–6, ACM/IEEE, 2019.

[179] T. G. Dietterich and G. Bakiri, "Error-correcting output codes: a general method for improving multiclass inductive learning programs," in *Proceedings of the 9th National Conference on Artificial Intelligence - Volume 2*, pp. 572–577, 1991.

[180] M. Liu, L. Xia, Y. Wang, and K. Chakrabarty, "Fault tolerance in neuromorphic computing systems," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pp. 216–223, ACM, 2019.

[181] M. Liu, L. Xia, Y. Wang, and K. Chakrabarty, "Fault tolerance for RRAM-based matrix operations," in *Proceedings of the International Test Conference (ITC)*, pp. 1–10, IEEE, 2018.

[182] C. Liu, M. Hu, J. P. Strachan, and H. Li, "Rescuing memristor-based neuromorphic design with high defects," in *Proceedings of the 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2017.

[183] L. Chen, J. Li, Y. Chen, Q. Deng, J. Shen, X. Liang, and L. Jiang, "Accelerator-friendly neural-network training: Learning variations and defects in RRAM crossbar," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pp. 19–24, IEEE, 2017.

[184] G. Yuan, Z. Liao, X. Ma, Y. Cai, Z. Kong, X. Shen, J. Fu, Z. Li, C. Zhang, H. Peng, N. Liu, A. Ren, J. Wang, and Y. Wang, "Improving DNN fault tolerance using weight pruning and differential crossbar mapping for ReRAM-based edge AI," in *22nd International Symposium on Quality Electronic Design (ISQED)*, pp. 135–141, IEEE, 2021.

[185] A. S. Rekhi, B. Zimmer, N. Nedovic, N. Liu, R. Venkatesan, M. Wang, B. Khailany, W. J. Dally, and C. T. Gray, "Analog/mixed-signal hardware error modeling for deep learning inference," in *56th Annual Design Automation Conference*, pp. 1–6, 2019.

[186] M. Klachko, M. R. Mahmoodi, and D. Strukov, "Improving noise tolerance of mixed-signal neural networks," in *International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2019.

[187] Z. He, J. Lin, R. Ewetz, J.-S. Yuan, and D. Fan, "Noise injection adaption: End-to-end ReRAM crossbar non-ideal effect adaption for neural network mapping," in *Proceedings of the 56th Annual Design Automation Conference*, pp. 1–6, 2019.

[188] Y. Long, X. She, and S. Mukhopadhyay, "Design of reliable DNN accelerator with unreliable ReRAM," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1769–1774, IEEE, 2019.

[189] X. Yang, S. Belakaria, B. K. Joardar, H. Yang, J. R. Doppa, P. P. Pande, K. Chakrabarty, and H. H. Li, "Multi-objective optimization of ReRAM crossbars for robust DNN inferencing under stochastic noise," in *International Conference on Computer Aided Design (ICCAD)*, IEEE/ACM, 2021.

[190] V. Joshi, M. Le Gallo, S. Haefeli, I. Boybat, S. R. Nandakumar, C. Piveteau, M. Dazzi, B. Rajendran, A. Sebastian, and E. Eleftheriou, "Accurate deep neural network inference using computational phase-change memory," *Nature Communications*, vol. 11, no. 1, pp. 1–13, 2020.

[191] Y. Shen, N. C. Harris, S. Skirlo, M. Prabhu, T. Baehr-Jones, M. Hochberg, X. Sun, S. Zhao, H. Larochelle, D. Englund, and M. Soljačić, "Deep learning with coherent nanophotonic circuits," *Nature Photonics*, vol. 11, no. 7, pp. 441–446, 2017.

[192] L. Bernstein, A. Sludds, R. Hamerly, V. Sze, J. Emer, and D. Englund, "Freely scalable and reconfigurable optical hardware for deep learning," *Scientific Reports*, vol. 11, no. 1, pp. 1–12, 2021.

[193] B. J. Shastri, A. N. Tait, T. Ferreira de Lima, W. H. Pernice, H. Bhaskaran, C. D. Wright, and P. R. Prucnal, "Photonics for artificial intelligence and neuromorphic computing," *Nature Photonics*, vol. 15, no. 2, pp. 102–114, 2021.

[194] F. Ashtiani, A. J. Geers, and F. Aflatouni, "An on-chip photonic deep neural network for image classification," *Nature*, pp. 1–6, 2022.

[195] F. P. Sunny, E. Taheri, M. Nikdast, and S. Pasricha, "A survey on silicon photonics for deep learning," *ACM Journal of Emerging Technologies in Computing System*, vol. 17, no. 4, pp. 1–57, 2021.

[196] J. Frankle and M. Carbin, "The lottery ticket hypothesis: Finding sparse, trainable neural networks," in *International Conference on Learning Representations*, 2018.

[197] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[198] "Heaviside step function." https://mathworld.wolfram.com/HeavisideStepFunction.html. Accessed: Apr-4-2023.

[199] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Proceedings of the International Conference on Artificial Neural Networks*, pp. 281–290, Springer, 2014.

[200] I. Bayraktaroglu and A. Orailoglu, "Concurrent test for digital linear systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1132–1142, 2001.

[201] V. Nair and J. A. Abraham, "Real-number codes for fault-tolerant matrix operations on processor arrays," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 426–435, 1990.

[202] F. Chollet, "Keras." https://keras.io, 2015.

[203] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016.

[204] W. Ford, *Numerical linear algebra with applications: Using MATLAB*. Academic Press, 2014.

[205] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT)*, pp. 177–186, Springer, 2010.

[206] J. Shlens, "A tutorial on principal component analysis," *arXiv preprint arXiv:1404.1100*, 2014.

[207] A. Krizhevsky, *Learning multiple layers of features from tiny images*. CiteSeerX, 2009.

[208] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 2001.

[209] "cuBLAS: Dense linear algebra on GPUs." https://developer.nvidia.com/cublas. Accessed: Apr-4-2023.

[210] "NVIDIA deep learning accelerator." http://nvdla.org/index.html. Accessed: Apr-4-2023.

[211] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[212] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in Neural Information Processing Systems*, vol. 29, 2016.

[213] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: an imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, vol. 32, pp. 8026–8037, 2019.

[214] K. E. Batcher, "Sorting networks and their applications," in *Spring Joint Computer Conference*, p. 307–314, 1968.

[215] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," *arXiv preprint arXiv:1602.07360*, 2016.

[216] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference*, pp. 1–10, 2013.

[217] H. Cho, E. Cheng, T. Shepherd, C.-Y. Cher, and S. Mitra, "System-level effects of soft errors in uncore components," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1497–1510, 2017.

[218] D. Williamson, "Dynamically scaled fixed point arithmetic," in *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*, pp. 315–318, 1991.

[219] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[220] M. Bushnell and V. Agrawal, *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*, vol. 17. Springer Science & Business Media, 2004.

[221] J. H. Kim and S. M. Reddy, "On the design of fault-tolerant two-dimensional systolic arrays for yield enhancement," *IEEE Transactions on Computers*, vol. 38, no. 4, pp. 515–525, 1989.

[222] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[223] D. C. Lay, S. R. Lay, and J. J. McDonald, *Linear Algebra and Its Applications*. USA: Pearson Publishing Co., 5th ed., 2016.

[224] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[225] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pp. 249–256, 2010.

[226] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, Y. Guan, Z. Wang, X. Jia, X. Li, M. Guo, and Y. Zhu, "Accelerating sparse DNN models without hardware-support via tile-wise sparsity," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2020.

[227] Z. Wang, "SparseRT: Accelerating unstructured sparsity on GPUs for deep learning inference," in *ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 31–42, 2020.

[228] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 3 ed., 2009.

[229] J. Frankle, G. K. Dziugaite, D. M. Roy, and M. Carbin, "Pruning neural networks at initialization: Why are we missing the mark?," *arXiv preprint arXiv:2009.08576*, 2020.

[230] N. Lee, T. Ajanthan, and P. H. Torr, "SNIP: Single-shot network pruning based on connection sensitivity," in *International Conference on Learning Representations*, 2019.

[231] C. Wang, G. Zhang, and R. Grosse, "Picking winning tickets before training by preserving gradient flow," in *International Conference on Learning Representations*, 2020.

[232] H. Tanaka, D. Kunin, D. L. Yamins, and S. Ganguli, "Pruning neural networks without any data by iteratively conserving synaptic flow," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[233] R. W. Vuduc, *Automatic performance tuning of sparse matrix kernels*. CiteSeerX, 2003.

[234] S. M. Nabavinejad, M. Baharloo, K.-C. Chen, M. Palesi, T. Kogel, and M. Ebrahimi, "An overview of efficient interconnection networks for deep neural network accelerators," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 3, pp. 268–282, 2020.

[235] "ISO 21448:2022 Road vehicles — Safety of the intended functionality." https://www.iso.org/obp/ui/#iso:std:iso:21448:ed-1:v1:en, 2022.