UNIVERSITY OF CALIFORNIA SAN DIEGO

Efficient and Resilient Neural Networks for On-chip Inference

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

in

Computer Science

by

Olivia Weng

Committee in charge:

Professor Ryan Kastner, Chair
Professor Javier Mauricio Duarte
Professor Sicun Gao
Professor Nhan Tran
Professor Jishen Zhao

2026

The Dissertation of Olivia Weng is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2026

DEDICATION

To my mother, my father, and my brother.

EPIGRAPH

... note
the way the soap dish enables you,
or the window latch grants you freedom.

David Whyte, “Everything is Waiting for You”

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGEMENTS

I would like to acknowledge Tom Connolly and Rajiv Gandhi for being my first computer science teachers and encouraging me to find the joy and wonder in doing hard things. They led me to major in computer science in college, which brought me many things to be thankful for. Many thanks to Yanjing Li for being a wonderful computer architecture teacher. Her class truly blew my mind on what a computer was and could be. I am grateful for her patience in training me as a young researcher. I am also indebted to Andrew A. Chien for accepting me into his graduate computer architecture class and helping me publish my first workshop paper. My mom and I loved the opportunity to visit Bologna, where I presented this work at my first conference.

My PhD was filled with wonderful people who encouraged and supported me throughout the years. Thank you to my advisor Ryan Kastner for accepting me into his group and listening to me intently during our weekly one-on-ones, especially as I prepared many contingency plans for worst-case scenario after worst-case scenario. I am especially grateful for his immense support in helping me write my papers when my physical capabilities had other plans. Even before then, I am grateful that he met with me when I was still an undergraduate at UChicago. His kindness and patience during our conversation convinced me to apply to UCSD. We grew to be great collaborators by the end of my PhD, which was great fun.

I was fortunate to have had many collaborators throughout my PhD. Many thanks to Marta Andronic, Nhan Tran, Javier Mauricio Duarte, Dustin Richmond, Colin Drewes, and Gabriel Marcano for showing me how great research collaborations can be. I learned so much from all of them, and each experience made me the researcher I am today. I was also lucky to have TA'd with Steve Swanson and co-taught CSE 29 with Joe Politz. That experience made me the teacher I am today.

I would like to thank my labmates for filling the lab with many jokes and entertaining my many lab outings. I had the joy of watching and experiencing the lab grow during the course of my PhD. Many thanks to Chris Crutchfield and Sean Perry for maintaining our server, Waiter. Raymond Dueñas for always having the best attitude. Danial Zuberi for being a great collaborator

and taking me to the gym. Abarajithan Gnaneswaran, Zhenghua Ma, and Alexander Redding for being my officemates in crime, collaborators, and great friends. In particular, I must thank Andres Meza for being there for me always as a labmate, collaborator, and friend. I also must thank Jennifer Switzer for being a great friend and helping me maintain my sanity throughout my PhD. All of our mutual commiseration and support kept me going the whole time, and I am forever indebted in a way that words cannot describe.

Outside of the lab, I had the fortune of spending time with Amanda Tomlinson, Alex Yen, Prudhviraj Naidu, Mia Syme, and Kate Matthews. In particular, I cherished the times spent with Eric Mugnier chit chatting at his desk. I would also like to thank Sam Chen for commiserating with me and being a great listener and friend. They made UCSD a great place to be a student and a young person.

I would also like to thank my friends who reminded me of life outside of my PhD. Many thanks to Kevin, Isaac, Christopher, Mai, Leshya, and Ellen for always being on the other end of the line and being great listeners and sources of advice.

Finally, I am indebted to my family. Thank you to my mother, my father, and my big brother for being there for me always.

This material is based upon work supported by the NSF Graduate Research Fellowship Program under Grant No. DGE-2038238. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

Chapter 3, in part, is a reprint of the material as it appears in ACM Trans. Reconfigurable Technol. Syst. 17, 1, Article 11 (March 2024). The work in this chapter was done in collaboration with Gabriel Marcano, Vladimir Loncar, Alireza Khodamoradi, Abarajithan G, Nojan Sheybani, Andres Meza, Farinaz Koushanfar, Kristof Denolf, Javier Mauricio Duarte, and Ryan Kastner. The dissertation author was the primary investigator and author of this work.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25). The

work in this chapter was done in collaboration with Marta Andronic, Danial Zuberi, Jiaqing Chen, Caleb Geniesse, George A. Constantinides, Nhan Tran, Nicholas J. Fraser, Javier Mauricio Duarte, and Ryan Kastner. The dissertation author was the primary investigator and author of this work.

Chapter 5, in part, is a reprint of the material as it appears in ACM J. Auton. Transport. Syst. 1, 3, Article 15 (September 2024). The work in this chapter was done in collaboration with Andres Meza, Quinlan Bock, Benjamin Hawks, Javier Campos, Nhan Tran, Javier Mauricio Duarte, and Ryan Kastner. The dissertation author was the primary investigator and author of this work.

Chapter 6, in part, is a reprint of the material as it appears in Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26). The work in this chapter was done in collaboration with Andres Meza, Nhan Tran, and Ryan Kastner. The dissertation author was the primary investigator and author of this work.

VITA

2020 Bachelor of Science in Computer Science, University of Chicago

2023 Master of Science in Computer Science, University of California San Diego

2026 Doctor of Philosophy in Computer Science, University of California San Diego

PUBLICATIONS

Olivia Weng, Andres Meza, Nhan Tran, and Ryan Kastner. 2026. *PrioriFI: More Informed Fault Injection for Edge Neural Networks.* In Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26). https://doi.org/10.1145/3779212.3790204

Olivia Weng, Marta Andronic, Danial Zuberi, Jiaqing Chen, Caleb Geniesse, George A. Constantinides, Nhan Tran, Nicholas J. Fraser, Javier Mauricio Duarte, and Ryan Kastner. 2025. *Greater than the Sum of its LUTs: Scaling Up LUT-based Neural Networks with AmigoLUT.* In Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25). https://doi.org/10.1145/3706628.3708874

Colin Drewes, Tyler Sheaves, Olivia Weng, Keegan Ryan, Bill Hunter, Christopher McCarty, Ryan Kastner, and Dustin Richmond. 2024. *Turn on, Tune in, and Listen up: Maximizing Side-Channel Recovery in Cross-Platform Time-to-Digital Converters.* ACM Trans. Reconfigurable Technol. Syst. 17, 3, Article 49 (September 2024). https://doi.org/10.1145/3666092

Olivia Weng, Andres Meza, Quinlan Bock, Benjamin Hawks, Javier Campos, Nhan Tran, Javier Mauricio Duarte, and Ryan Kastner. 2024. *FKeras: A Sensitivity Analysis Tool for Edge Neural Networks.* ACM J. Auton. Transport. Syst. 1, 3, Article 15 (September 2024). https://doi.org/10.1145/3665334

Colin Drewes, Olivia Weng, Andres Meza, Alric Althoff, David Kohlbrenner, Ryan Kastner, and Dustin Richmond. 2024. *Pentimento: Data Remanence in Cloud FPGAs.* In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24). https://doi.org/10.1145/3620665.3640355

Olivia Weng, Gabriel Marcano, Vladimir Loncar, Alireza Khodamoradi, Abarajithan G, Nojan Sheybani, Andres Meza, Farinaz Koushanfar, Kristof Denolf, Javier Mauricio Duarte, and Ryan Kastner. 2024. *Tailor: Altering Skip Connections for Resource-Efficient Inference.* ACM Trans. Reconfigurable Technol. Syst. 17, 1, Article 11 (March 2024). https://doi.org/10.1145/3624990

Colin Drewes, Olivia Weng, Keegan Ryan, Bill Hunter, Christopher McCarty, Ryan Kastner, and Dustin Richmond. 2023. *Turn on, Tune in, Listen up: Maximizing Side-Channel Recovery in Time-to-Digital Converters.* In Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23). https://doi.org/10.1145/3543622.3573193

ABSTRACT OF THE DISSERTATION

Efficient and Resilient Neural Networks for On-chip Inference

by

Olivia Weng

Doctor of Philosophy in Computer Science

University of California San Diego, 2026

Professor Ryan Kastner, Chair

Scientific applications are increasingly using neural networks (NNs) at the edge as a fundamental tool for advancing fields such as particle physics and materials science. As the scientific instruments used in these experiments become more advanced, they produce a lot more data (e.g., 40 TB/s) than before. As a result, scientists are relying on edge NNs, which have more capabilities than traditional algorithms, to process the data. To process data quickly enough, these scientific edge NNS have unique requirements. They must (1) be heavily quantized and (2) execute fully on chip. Even more so, these scientific NNs often operate in high radiation environments (1000$\times$ that of space). My thesis focuses on using hardware-software co-design to create efficient, fault-tolerant computer architectures for NNs that execute fully on chip so that

they meet the strict latency and throughput requirements laid out by these scientific experiments. I defend the following thesis statement: On-chip neural network inference introduces unique hardware-software codesign challenges and opportunities for building efficient, fault-tolerant neural network architectures. I provide evidence for my thesis in three parts: (1) codesigning residual NNs for efficient inference, (2) scaling up lookup-table NNs, and (3) codesigning fault-tolerant edge NNs. My thesis provides insights and tradeoffs that will help scientists better run their NNs on specialized hardware such as FPGAs and ASICs to advance research in their fields.

Chapter 1

Introduction

As scientific instruments become more advanced, they produce an increasing amount of data. For example, the High-Luminosity Large Hadron Collider (HL-LHC) at CERN is expected to generate up to hundreds of terabytes of data per second in the coming years [61]. Processing this immense amount of data in real-time is crucial for scientific research, but traditional algorithms often fall short in advancing the latest research. As a result, scientists are looking to neural networks (NNs) as a potential solution [61, 81]. NNs are typically built in a brute force way, where researchers add more and more parameters to increase performance, scaling them out to enormous sizes, e.g., GPT-3 has 175 billion parameters, which amounts to 350 GB of data when stored with float16 precision [29]. However, this is not practical when data is coming in at rates of several TB/s because current computer architectures cannot execute such enormous NNs in time without being extremely expensive and impractical. As a result, researchers in these scientific fields are building edge NNs that can be executed on a machine within the computation budget ($O(\text{ns})$) allowed by the massive data rate. This has several implications for the computer architectures that must run these edge NNs. In order for the hardware to meet these extreme low-latency requirements, we must perform inference fully on-chip.

To perform on-chip inference quickly, designers rely on *hardware-software codesign* with custom or reconfigurable logic. Hardware-software codesign involves designing the hardware and

software together to achieve a more efficient design compared with the traditional methodology of designing software for a fixed hardware architecture. By codesigning the hardware and software together, designers can optimize the hardware architecture for the specific software workload, and vice versa, leading to improved performance, energy efficiency, and resource utilization. Oftentimes, hardware-software codesign is the most efficient way to meet the extreme requirements of scientific applications [81].

In hardware-software codesign, the desirable objectives are power, performance, and area (PPA). The goal is usually to minimize power and area (also known as resource utilization) while maximizing performance (e.g., maximizing throughput or minimizing latency), but in most cases tradeoffs across PPA must be made. Due to the extreme data rates, the PPA objectives for scientific applications are often severely constrained. For instance, the encap concentrator (ECON-T) ASIC [81] executes an autoencoder at the LHC that must complete inference in 25 ns within a 4 mm^2 area and 95 mW power budget. Additionally, the ECON-T ASIC operates inside the LHC, which is an extremely high radiation environment, causing hardware failures every 15 seconds [200]. This makes reliability an additional design objective because the hardware must be able to tolerate faults and continue to operate correctly within the existing constraints, extending the PPA tradeoff space to include reliability, which we call PPAR. Considering PPAR is especially crucial to achieve fast and reliable on-chip inference for scientific applications. Fitting an efficient and resilient hardware design within the available resources on-chip is difficult because there are often design decisions for one objective that may conflict with another. Using off-chip resources could help designers balance PPAR more easily, but the extreme latency requirements of scientific applications often make accessing off-chip resources infeasible, forcing designers to fit everything on-chip.

Why on-chip inference? Given the extreme bandwidth and latency requirements of many scientific machine learning tasks [61, 197], on-chip memories such as flip-flops (FFs) and block random access memories (BRAMs) are often the only kinds of memory with both the bandwidth and latency capable of meeting these constraints. To demonstrate, we present Fig. 1.1. We chart

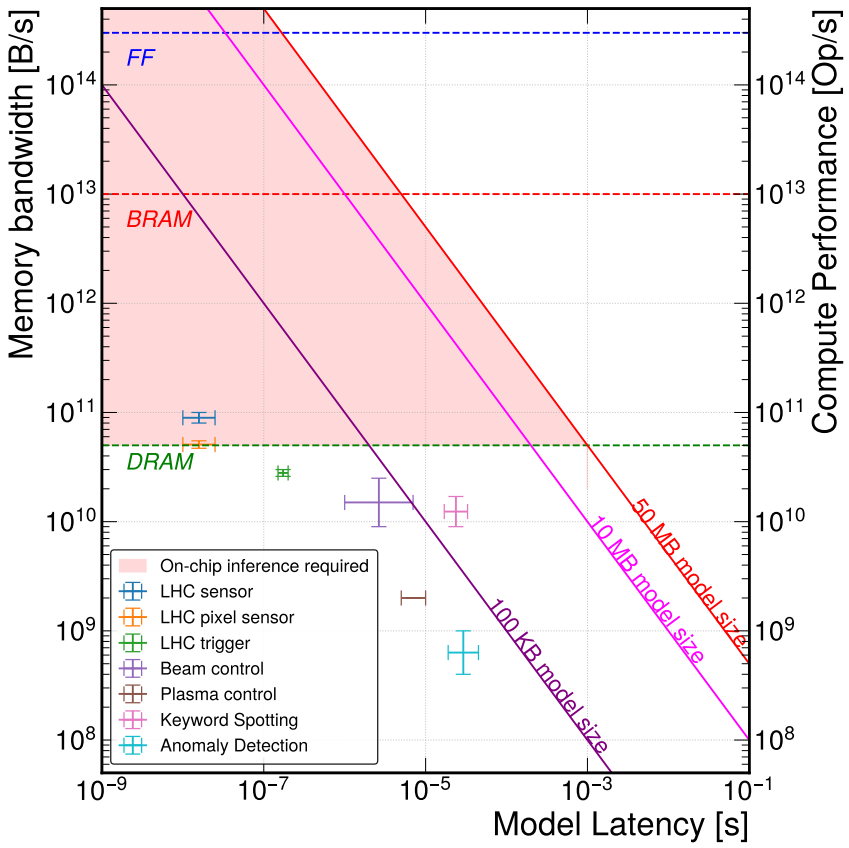


Figure 1.1. Many scientific and edge NNs must process incoming data at a high rate, requiring on-chip inference to process the data at least as fast as it arrives [27, 61, 197]. This leads to extreme low-latency and high-bandwidth requirements.

several scientific and IoT NNs according to their memory bandwidth for reading NN weights and the latency they achieve on field-programmable gate arrays (FPGAs) and application-specific integrated circuits (ASICs), focusing first on latency on the x-axis and memory bandwidth on the left y-axis. For example, the LHC sensor NN requires 80 GB/s memory bandwidth since it loads 2000 one-byte weights every 25 ns. This NN has bandwidth requirements that exceed the capabilities of DRAM (assuming DDR5's 50 GB/s bandwidth noted by the green horizontal dashed line). It is therefore infeasible to access weights off-chip from DRAM. As a result, the LHC sensor NN must rely on on-chip memory technology such as BRAM (red dashed line) and FFs (blue dashed line), which have significantly higher bandwidth at the expense of smaller

Table 1.1. On-chip memory storage, i.e., BRAMs and FFs, offers significantly more bandwidth and speed, making it the ideal option for our high-bandwidth, low-latency benchmarks [102]. The tradeoff, however, is total capacity, as these on-chip options can only store megabytes of data as opposed to DRAM's gigabytes.

	DRAM	**BRAM**	**FF**
Latency	10 - 100 ns	1 - 1.4 ns	1 - 1.4 ns
Bandwidth	10s GB / s	TB / s	100s TB / s
Total capacity	GBs	MBs	100s KBs

memory capacity (Tab. 1.1).

Next, let us focus on latency on the x-axis and compute performance (in operations/s) on the right y-axis to understand the diagonally charted lines. To start, we make some assumptions. First, we assume the NNs for each task are fully connected and the weights are one byte wide, as scientific benchmarks often work with fully connected layers that are heavily quantized, meaning their weights are represented with reduced precision, e.g., 8-bit integer, compared with the typical 32-bit floating point. Also, these NNs process data with batch size 1, as data is usually streamed in one sample at a time. Second, we assume these NNs execute on a performant FPGA, e.g., a Xilinx VU13P, using 5 000 digital signal processors (DSPs) operating at 200 MHz, performs 1 TOP/s. Third, we assume that with one-byte weights and 1 TOP/s, we can handle 1 TB/s of bandwidth at a compute performance of 1 TOP/s. Based on these assumptions, consider a model with 10 MB of weights (the diagonal pink line). Executing this model at a latency of 10^{-5}s requires 1 TOP/s with a memory bandwidth of 1 TB/s (left y-axis). From these assumptions, we can project how much compute performance and memory bandwidth a 10 MB model needs to run at different latencies. We also project the compute and bandwidth for a 100 KB model (purple diagonal line) and a 50 MB model (red diagonal line). Assuming that on-chip storage can only store a maximum of 50 MB of weights, we shade the region of this chart where on-chip inference is required in red. The floor is the maximum DRAM bandwidth and the ceiling is the 50 MB model's memory bandwidth and compute performance.

As such, NN implementations that fall in this shaded region must be executed with

weights fully on-chip, as demonstrated by the LHC sensor and LHC pixel sensor NNs. NN implementations that reside outside this region can instead rely on off-chip memory, e.g., the Anomaly Detection IoT NN.

Many of these scientific NN implementations will likely become more bandwidth-intensive as scientific instruments continue to advance. This implies that the NNs charted in Fig. 1.1 will continue to move up and to the left. Thus, on-chip NN inference will persist and become more crucial, and we must design for it by carefully balancing PPAR.

Balancing PPAR for extreme scientific applications is a challenging task. It often requires designers to make tradeoffs between the different PPAR design objectives, such as reducing latency at the cost of increased area or reducing area at the cost of worse reliability. My thesis aims to ease the burden of codesigning for PPAR by providing insights and tools to help designers navigate the complex design space of on-chip NN inference for scientific applications. A key insight behind my thesis is that the requirement of on-chip inference is not necessarily a hindrance in the design process. While designers cannot merely deploy large and highly accurate NNs fully on chip, edge NNs, albeit small, are still often over-parameterized. This over-parameterization is often crucial for edge NNs to train to a good accuracy [76], but also opens the door to unique hardware-software codesign challenges and opportunities for building efficient and fault-tolerant architectures for NN inference at the edge.

1.1 Key Challenges

There are many challenges to codesigning NNs for on-chip inference. Hardware-software codesign is difficult because it requires knowledge of both NNs and computer architecture. Determining what kind of NN is best for the task, how best to quantize it, what kind of hardware architecture to use, how to map the NN to the hardware architecture, and how to determine the resilience of the NN are all questions that must be answered during the design process. When considering all the variables involved in designing hardware with the NN in mind and vice versa,

the design space explodes.

NN innovations can be costly to hardware. NN innovations help NNs train to reach higher accuracy but can be costly to hardware. For example, residual connections (also known as skip connections) [86] connect the outputs of one layer to the inputs of a later layer to solve the vanishing gradient problem; however, this comes at the cost of additional hardware resources and memory bandwidth for the intermediate outputs. As such, to deploy an NN with skip connections on-chip efficiently, designers must look to hardware-software codesign to redesign the NN and hardware architecture together, reducing the hardware costs of skip connections while maintaining the accuracy benefits they provide. NN innovations are often developed without the hardware in mind, and how to resolve this issue leads to the next challenge.

Hardware-software codesign requires expertise in both hardware and software. To meet the extreme constraints on PPAR for scientific applications, it is in the designer's best interest to have expertise in both hardware and software. This way the designer can maximize the capabilities of both the hardware and software to achieve nanosecond latency or build a design within the resource constraints. This is a challenge for many scientific researchers without a computer science or engineering background. A goal of this thesis is to provide insights and tools for scientists to use to help them navigate the complex design space of on-chip NN inference.

The design space is large. When considering all the design parameters for both hardware and software, the design space becomes exponentially large, making it difficult to explore and find optimal solutions. Design space exploration (DSE) is a crucial step in the hardware-software codesign process, as it allows designers to evaluate different design options and make informed decisions about the tradeoffs between different design parameters. A challenge in DSE is how to explore the design space efficiently. If we tighten the DSE loop seen in Fig. 1.2, we can arrive at a good design more quickly. One approach is to make it easier to build a good design from the outset. A crucial part of the DSE loop is evaluating the design against PPAR. This evaluation can be time-consuming, especially when evaluating the reliability of an edge NN. Making the evaluation process quicker or more informative is thus desirable so that designers can iterate on

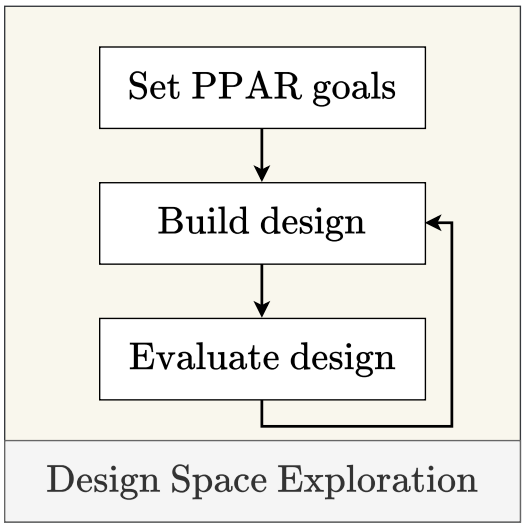


Figure 1.2. The Design Space Exploration (DSE) loop. DSE has four key steps: (1) choose PPAR objectives, (2) build a design, (3) evaluate the design against PPAR, and (4) repeat from step 2. Easing the burden of building the design and evaluating it will help designers arrive at an efficient design faster.

their designs faster.

Evaluating NN resilience is time-consuming. The common way to evaluate the reliability of an NN to faults is to conduct a fault injection (FI) campaign. An FI campaign simulates a fault in software or hardware and runs a number of inputs through the NN to assess performance. This process repeats until the campaign is complete as determined by the researcher. To get a complete picture of an NN's resilience to faults in its parameters, researchers must inject a fault, e.g., flip a bit in a parameter, and evaluate its effect for every single parameter bit. This is computationally intensive because it requires simulating a vast number of faulty scenarios, particularly given the thousands to millions of parameters and operations in an NN. It is not immediately obvious which parameter bits would be more or less likely to be sensitive to faults. Even a common intuition that bits with higher significance, e.g., the most significant bit (MSB), would be more sensitive to faults does not always hold true, as I will show later in the thesis (Chapter 6). A goal of this work is to demonstrate to what extent this intuition does not hold and

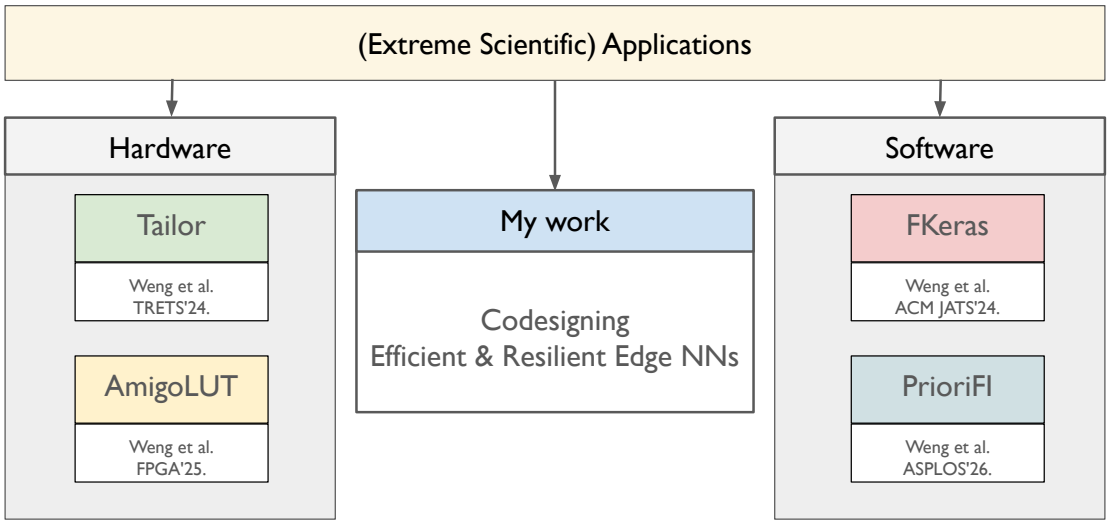


Figure 1.3. My work takes a top-down (software-first) and bottom-up (hardware-first) approach to explore the design space for on-chip neural network inference. My hardware-first work focuses on building efficient NN hardware architectures without sacrificing accuracy. My software-first work introduces new tools and algorithms for analyzing the fault sensitivity of edge NNs.

to provide a more informed way to conduct FI campaigns that can help designers evaluate the reliability of their NNs more quickly and with more insights.

1.2 Thesis

In this dissertation, I argue that hardware-software codesign is crucial for building efficient and fault-tolerant architectures for on-chip NN inference for scientific applications. I defend the following thesis statement: *On-chip neural network inference introduces unique hardware-software codesign challenges and opportunities for building efficient, fault-tolerant neural network architectures.* I explore my thesis through both bottom-up (hardware-first) and top-down (software-first) approaches, as seen in Fig. 1.3.

The bottom-up approach focuses on building efficient NNs and hardware architectures for on-chip inference. I focus on finding methods of building more resource-efficient NNs while

maintaining their accuracy. In Tailor and AmigoLUT, I introduce training algorithms that reshape the NN topology in ways that map them more resource-efficiently for field-programmable gate arrays (FPGAs). FPGAs have unique resource types such as lookup tables (LUTs), flip-flops (FFs), digital signal processors (DSPs), and block random access memories (BRAMs). How to best use them for on-chip inference heavily depends on the architecture used to implement the NN. Based on the architecture, Tailor and AmigoLUT introduces methods for reducing resource usage without completely sacrificing accuracy.

The top-down approach focuses on building software FI tools and algorithms to help designers better evaluate the reliability of their edge NNs to faults. It tackles the problem of how to efficiently evaluate the fault tolerance of edge NNs, as many prior works have introduced methods that work best on NNs represented in floating point. This is not the case for many edge NNs, as they are heavily quantized to less precise data types like fixed point. In FKeras and PrioriFI, I leverage the Hessian, which is a matrix of second derivatives that has been show to capture how well the NN's parameters respond to perturbations [208], to analyze the sensitivity of edge NNs to faults. FKeras uses the Hessian to rank a NN's parameter bits from approximately most to least sensitive to faults, guiding FI campaigns to inject faults in the most sensitive bits first. PrioriFI improves on the shortcomings of FKeras by using the results of prior FIs as a heuristic to guide future FIs to better target the more sensitive bits.

1.3 Insights

My thesis leverages three key insights: (1) edge NNs, despite being small, are over-parameterized and can be redesigned to better fit the hardware constraints of on-chip inference; (2) edge NNs can be analyzed using the Hessian to provide insights into its sensitivity to faults and guide the design of more fault-tolerant architectures; and (3) bit-level monotonicity is not a reliable heuristic for guiding FI campaigns.

Edge NNs are usually over-parameterized, creating opportunities for efficient re-

designs. This is not a shortcoming of the edge NN designer, but more so a side effect of effectively training edge NNs to reach good accuracy. In general, NNs are often over-parameterized, meaning they have more parameters than necessary to achieve high accuracy [76]. Even edge NNs are over-parameterized, because it is usually a necessary step during training to provide the NN with the "necessary scaffolding" to learn the task at hand. It is up to the designer to find ways to redesign the edge NN and its hardware architecture together to be more efficient while maintaining accuracy. Figuring out how to reduce the unnecessary parameters is difficult. There are many ways, such as pruning, quantization, knowledge distillation, and neural architecture search, but it is not always clear which method is best for a given NN and hardware architecture. Tailor uses knowledge distillation to reduce the extra parameters introduced by skip connections. AmigoLUT takes a more clever approach of breaking up one large over-parameterized LUT NN into an ensemble of smaller, i.e., less over-parameterized, LUT NNs. The smaller LUT NNs work together to fill the gaps in each other's performance, improving accuracy without resource usage exploding, which is a common problem with the large LUT NNs.

When evaluating the resilience of edge NNS, it is important to consider methods for gaining insights into its sensitivity without solely relying on time-consuming FI campaigns. If there were a way to analyze the sensitivity a priori, designers could use this information to guide their FI campaigns to be more efficient, e.g., by targeting the more sensitive bits first. This way they can avoid spending time injecting faults into insensitive bits. My thesis demonstrates that **the Hessian is useful for analyzing the fault sensitivity of edge NNs.** The Hessian is a matrix of second derivatives that has been shown to capture how well the NN's parameters respond to perturbations [208]. Previously, the Hessian has been used in the edge NN space to provide insights into how to quantize an NN [56, 207, 208], demonstrating that it effectively captures perturbations from quantization. I demonstrate that the Hessian can also be used to capture perturbations from faults, providing a way to rank NN parameter bits by their sensitivity to faults. In FKeras and PrioriFI, I use this ranking to guide FI campaigns to inject faults in the most sensitive bits first, which can help designers evaluate the reliability of their NNs more quickly,

tightening the DSE evaluation loop.

Prior work on evaluating NN resilience has often relied on the intuition that bits with higher significance, e.g., the MSBs, are more sensitive to faults, which is known as bit-level monotonicity; however, my dissertation shows that **bit-level monotonicity is not a reliable heuristic for guiding FI campaigns.** Bit-level montonicity seems like it would hold because flipping more significant bits should result in larger changes in magnitude for a given parameter. Flipping the MSB should result in a larger change than flipping the second MSB. Yet, this is not always the case due to other NN operations downstream. For instance, the commonly used ReLU activation function does not behave strictly monotonically, leading to non-monotonic behavior when flipping NN parameter bits [40]. Chapter 6 provides detailed evidence for this. One goal of my dissertation is to steer designers away from relying on bit-level monotonicity when evaluating the fault tolerance of their edge NNs. While it provides a reasonable first-order approximation, its inefficiencies are not worthwhile when they can be avoided with FI algorithms like Chapter 6's PrioriFI.

1.4 Contributions

In this dissertation, I make several contributions to building efficient and fault-tolerant architectures for on-chip NN inference.

Chapter 2: Background. I review some background for understanding the rest of the thesis, including NN quantization, various on-chip computer architecture designs, and how prior work has studied NN fault tolerance.

Chapter 3: Altering Skip Connections for Resource-Efficient Inference. Tailor introduces a hardware-aware training algorithm that alters skip connections to be more resource-efficient for on-chip inference while maintaining accuracy. NNs use skip connections to improve training convergence. However, these skip connections are costly in hardware, requiring extra buffers and increasing on- and off-chip memory utilization and bandwidth requirements. In

this chapter, I show that skip connections can be optimized for hardware when tackled with a hardware-software codesign approach. I argue that while a NN's skip connections are needed for the NN to learn, they can later be removed or shortened to provide a more hardware efficient implementation with minimal to no accuracy loss. Tailor's hardware-aware training algorithm gradually removes or shortens a fully trained NN's skip connections to lower their hardware cost. A key insight is that shortened skip connections that skip over only one NN layer have the unique on-chip opportunity to be implemented with more efficient memories such as shift registers instead of BRAMs.

Chapter 4: Greater than the Sum of its LUTs: Scaling Up LUT-based Neural Networks with AmigoLUT. AmigoLUT [199] introduces the idea of using ensemble learning on lookup table (LUT) NNs to scale up accuracy without exponentially increasing resources. Lookup-table NNs achieve extremely high throughput and low latency inference by implementing NNs as lookup tables (LUTs), achieving inference latency on the order of nanoseconds. Since LUTs are a fundamental FPGA building block, LUT-based NNs efficiently map to FPGAs. LogicNets (and its successors) form one class of LUT-based NNs that target FPGAs, mapping neurons directly to LUTs to meet low latency constraints with minimal resources. However, it is difficult to build larger, more performant LUT-based NNs like LogicNets because LUT usage increases exponentially with respect to neuron fan-in (i.e., number of synapses $\times$ synapse bitwidth). A large LUT-based NN quickly runs out of LUTs on an FPGA. I introduce Tailor to address this issue by creating ensembles of smaller LUT-based NNs that scale linearly with respect to the number of models. Tailor improves the scalability of LUT-based NNs, reaching higher throughput with up to an order of magnitude fewer LUTs than the largest LUT-based NNs.

Chapter 5: FKeras: A Sensitivity Analysis Tool for Edge Neural Networks. FKeras introduces a sensitivity analysis tool for edge NNs that uses the Hessian to provide an approximate ranking of the parameter bits from most to least fault-sensitive as a way to guide FI campaigns to stop early and find more sensitive bits faster. Edge computation often requires robustness to

faults, e.g., to reduce the effects of transient errors and to function correctly in high radiation environments. In these cases, the edge device must be designed with fault tolerance as a primary objective. FKeras is a tool that helps design fault-tolerant edge neural networks that run entirely on chip to meet strict latency and resource requirements. FKeras provides metrics that give a bit-level ranking of neural network weights with respect to their sensitivity to faults. FKeras includes these sensitivity metrics to guide efficient fault injection campaigns to help evaluate the robustness of a neural network architecture. I use FKeras to analyze a NN's fault tolerance to consider alongside its accuracy, performance, and resource consumption. The results show that the different NN architectures have vastly differing resilience to faults.

Chapter 6: PrioriFI: More Informed Fault Injection for Edge Neural Networks. PrioriFI improves on the algorithm introduced in FKeras by initially using the Hessian ranking and then using prior FI results as a heuristic to guide future FIs to better target the more sensitive NN bits. In this chapter, I analyze the shortcomings of the common intuition that bits with higher significance, e.g., the MSBs, are more sensitive to faults. This is also known as bit-level monotonicity. I demonstrate that bit-level monotonicity does not always hold. PrioriFI overcomes these shortcomings by using prior FI results as a heuristic, avoiding the pitfalls of assuming bit-level monotonicity to find highly fault-sensitive bits first. With PrioriFI, designers can quickly evaluate different NNs and better co-design fault-tolerant edge NN systems.

Chapter 2

Background

My thesis builds on several techniques in the field of hardware-software codesign for efficient and resilient edge NN computing. In this chapter, I describe NN quantization, an important method found in most edge NN implementations to fit NNs within its given resource constraints—the remaining chapters presuppose a knowledge of quantization. I also review a few different computer architectures for implementing edge NNs with ultra low latency and high throughput, specifically on field-programmable gate arrays (FPGAs), which I target in Chapter 3 and Chapter 4. Finally, I review prior work on NN fault tolerance.

2.1 Quantization

Quantization is defined as reducing the precision used to represent neural network parameters, usually from n bits to m bits, where $n > m$. There is unique opportunity to achieving efficient inference with quantization because hardware such as FPGAs and ASICs can be configured to use any kind of numerical representation such as floating point, fixed point, integer, and even custom data types. Thus, the goal with NN quantization involves not only reducing the number of bits but also converting the original numerical representation to a less precise data type that is cheaper to implement in hardware. Doing so makes it is possible to achieve efficient, real-time inference. For example, a common case in NN quantization involves starting with 32-bit floating point representation (since that is generally what is used to train NNs on GPUs)

and converting it to 8-bit integer representation. This is desirable because integer arithmetic is less complex than floating point arithmetic and thus faster to compute. With fewer bits to compute and a cheaper numerical representation, executing an 8-bit integer NN significantly lowers latency, energy consumption, and resource utilization.

The primary challenge with quantization, however, is maintaining the NN's high accuracy post quantization. When reducing the precision of the network, the NN is effectively losing information it learned during training. This presents the main trade-off involved with quantization: *precision vs. accuracy*. A reasonable expectation is that we trade accuracy for lower precision to achieve smaller models that can fit in hardware; however, sometimes this is not the case.

NNs are often over-parameterized and can thus afford to lose precision with minimal to no loss in accuracy [77]. Since one of the primary overarching goals in NN quantization is maintaining accuracy, we are not particularly wedded to quantizing weights such that their quantized values are as close to their floating point counterparts as permitted by the quantization scheme. We want to quantize the weights in a way that maintains the network's overall classification accuracy. In fact, trying to minimize the distance between the floating point and quantized weight representations does not directly translate to maintaining classification accuracy because of the over-parameterization of neural networks [82, 139]. Therefore, the over-parameterization wrinkle presents opportunities to reduce precision in clever ways without any cost to accuracy—at times, quantizing a NN even improves its accuracy.

2.1.1 Representing Numbers on Machines

In this section, we review how data types are used to represent data, and more specifically real numbers, on machines. We detail how floating point and fixed point data types work because a common quantization scenario involves quantizing floating point to integer/fixed point. It is important to go into detail on the inner workings of floating point and fixed point to understand the costs and benefits of using them with respect to hardware.

Since data on computers is represented in binary, the machine needs to know how to

interpret the binary, i.e, as a character, string, integer, etc. T his is the primary function of data types. Several standards have been developed to dictate how we should use binary code to represent different kinds of data. For instance, the ASCII standard defines a correspondence between natural numbers and individual characters, like letters, numbers, punctuation, etc. The same needs to be done for representing real numbers.

While integers have a straightforward correspondence to binary values, representing real numbers in binary is much more complex. There are several considerations to be made. For instance, how precise we want our real number representations to be, i.e., how many decimal places that should be accounted for. It is so complex that there is an IEEE standard for how real numbers should be represented as well as how arithmetic on this real number representation should work—IEEE Standard 754, also known as IEEE floating point. Since there are infinitely many real numbers and only so many bits that can be allocated for representing each number on machines, we can actually view representing real numbers on computers as a quantization problem itself because we are reducing the precision of the reals [77].

There are two main ways of representing real numbers: floating point and fixed point.

Floating Point

The idea behind floating point is to represent numbers in scientific notation, $n \times 2^m$, wherein we need only keep track of n and m in our representation [30]. IEEE floating point defines a floating point number n as

$$n = (-1)^s \times m \times 2^E \tag{2.1}$$

where

- s is the *sign* bit, 0 for positive and 1 for negative,
- m is the *mantissa*, also called the significand, which defines the precision, and

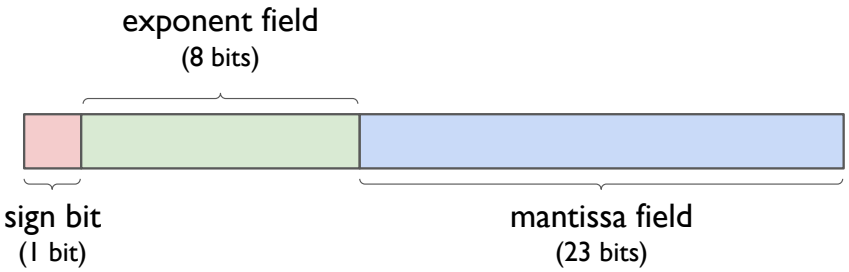


Figure 2.1. IEEE single-precision floating point bit fields. The binary of a floating point value is broken up, from least significant to most significant bit, into 23 bits for the mantissa field, 8 bits for the exponent field, and 1 bit for the sign bit.

- E is the *exponent*, which determines the range.

A floating point number encodes the sign bit, mantissa, and exponent in its binary representation. In single precision floating point—the most common precision used for NN training on GPUs—, 32 total bits are given, in which 8 bits are allocated for the exponent, 23 bits are allocated for the mantissa, and 1 bit is allocated for the sign bit, as seen in Figure 2.1. Depending on the value in the exponent field, there are three ways to interpret the values encoded in the exponent and mantissa bit fields (the sign bit is always interpreted as 0 or 1): 1) normalized values, 2) denormalized values, and 3) special values.

Normalized Values

When the exponent field E is neither 0 (all zeroes) nor 255 (all ones), we are working with normalized values. In this case, the bit fields are interpreted as follows:

- $E = x - Bias$, where x is the unsigned integer actually stored in the exponent bit field, and $Bias$ is $2^8 - 1 = 255$ (since we are given 8 exponent bits). Thus, $-254 \leq E \leq 255$.

- $m = 1 + f$, where f is the fractional binary value actually stored in the mantissa bit field. By fractional, we mean $0 \leq f < 1$. Essentially, f is the fractional number to the right of the radix point. Thus, $1 \leq m < 2$. We implicitly add 1 to gain an extra free bit of precision.

Based on these definitions, we compute normalized values n from the bit fields as

$$n = (-1)^s \times (1+f) \times 2^{x-255} \tag{2.2}$$

As seen in Equation (2.2), the floating point number is effectively scaled based on the value x stored in the exponent field. This allows the radix point to "float" to various positions in the binary value as needed, providing floating point with a large range and thus more precision.

Denormalized Values

When the exponent field E is all zeroes, we are working with denormalized values. In this case, the bit fields are interpreted as follows:

- $E = 1 - Bias$, where $Bias$ is still $2^8 - 1 = 255$, so $E = -254$.
- $m = f$, where f is the fractional binary value stored in the mantissa bit field, as in the normalized value case.

Based on these settings, we compute denormalized values m from the bit fields as

$$n = (-1)^s \times f \times 2^{-254} \tag{2.3}$$

Special Values

When the exponent field E is all ones, we represent special non-numerical values. For completeness, we include these special values. The value is determined by the bits residing in the mantissa bit field:

- $m = 0$. Depending on the sign bit, this value is $-\infty$ or $+\infty$.
- $m \neq 0$. This is NaN, also known as Not a Number.

Based on Equations (2.2) and (2.3), we see that there is extra arithmetic on fractional binary values involved with floating point values. Even though much of this arithmetic has been

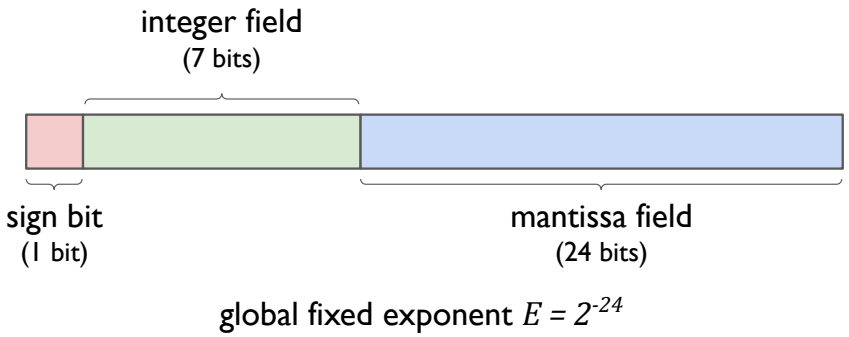


Figure 2.2. Example 32-bit fixed point format, where, from least significant to most significant bit, 24 bits are chosen for the mantissa, 7 bits are chosen for the integer, and 1 bit is for the sign bit. The value of the global fixed exponent depends on the number of bits allocated for the mantissa. In this case, there are 24 mantissa bits, implying that the global fixed exponent is 2^{-24}.

optimized, there is still an overhead cost that needs to be paid when using floating point values; case in point, a floating point unit is required, making floats significantly more expensive to compute in hardware than say integers, which have no such overhead. Moreover, there are as many normalized values ($n \geq 1$) as there are denormalized values ($0 \leq n < 1$), meaning floating point numbers are non-uniformly spaced. There are many more values represented between 0 and 1 such that floating point has higher precision for values that lie in this range. This has many implications for quantization when quantizing from floating point to integer. For instance, if a network has many high-precision values between 0 and 1 and relies on this precision, this presents an extra challenge for quantizing to a less precise data type like integers.

Fixed Point

Fixed point is similar to floating point in that it is also encoded using a sign bit and a mantissa, but it uses a single global fixed exponent value that is shared across all fixed point values. Since the exponent is fixed, there is no need to store it in the binary, and the remaining bits are allocated as the integer field. Figure 2.2 depicts an example 32-bit fixed point value that is broken up into 24 mantissa bits, 7 integer bits, and 1 sign bit. The global exponent effectively places the radix point at a fixed position; hence, "fixed point." The shared exponent is also referred to as a global scaling factor and is typically a power of 2, so that the scaling

multiplication can be implemented using bit shifts [49]. Based on the fixed point definition in Figure 2.2, given 24 mantissa bits, the global scaling factor is 2^{-24}.

Depending on the application, we might want more or less precision in our fixed point values. For instance, Vivado High-Level Synthesis defines a fixed point data type for its FPGA synthesis called `ap_fixed`$< T, I >$ that allows users to choose the fixed point format, where T is the total number of bits allocated for the fixed point number and I is how many integer bits are allocated [1]. Since the global exponent is fixed and is based on how many bits are allocated to the mantissa, there is a trade-off involved in choosing T and I. Fewer integer bits means there is more bits left for the mantissa, implying high precision but a small range. More integer bits means there are fewer bits left for the mantissa, leading to low precision but a larger range.

Fixed point is typically used in embedded systems that cannot afford to have a floating point unit but still need some precision in their computations [49]. Without a floating point unit, the fixed point bit fields are all interpreted as integers (as opposed to fractional binary values like the floating point mantissa is), which are much cheaper to compute on. To explain how this works, let us walk through an example. Based on the fixed point format depicted in Figure 2.2, let us encode $\frac{1}{3}$. Since our value is a fraction, we do not need any integer bits, so this field is all zeroes. For the mantissa field, given 24 bits and a global exponent of 2^{-24}, we store the integer 5,592,405 in the mantissa field because $5592405 \times 2^{-24} \approx \frac{1}{3}$. This can be derived for any real number fraction r and m mantissa bits, where the corresponding fixed point mantissa value is $\lfloor r \cdot 2^m \rfloor$ [62]. As such, to determine based on the sign bit, integer field, mantissa field, and global exponent the fixed point value n, we compute

$$n = (-1)^s \times (I + m \times E) \tag{2.4}$$

where s is the sign bit, I is the integer encoded value, m is the mantissa encoded value, and E is the global exponent. Although this equation looks similar to the floating point equations, note

that all of the values are integers and E is typically a power of 2, so all the computation can be performed using integer arithmetic and bit shifts, which is significantly cheaper than floating point arithmetic.

2.1.2 Quantization Example

To quantize a value, we must follow a quantization function that systematically maps high precision values to low precision values.

Quantization

A common quantization function $Q(r)$ maps a floating point value r to an integer by way of

$$Q(r) = Int(r/S) - Z \tag{2.5}$$

where S is a floating point scaling factor and Z is an integer zero point [77]. By zero point, we mean Z is the integer value that represents 0 in our quantization scheme, which could be 0 or another value. In our scope of quantizing NN for efficient inference, r is either a weight or activation. The $Int(\cdot)$ function assigns the scaled r to an integer, typically via rounding. The rounding function can be as simple as round-to-nearest or something more complex, as seen in [94, 139]. In this case, $Q(r)$ is an example of *uniform quantization* because all of our r input values are scaled by the same value S, implying that the distance between quantized values is equally spaced and thus uniform. It is possible to define non-uniform distances between quantized values, which is known as *non-uniform quantization*; however, this is out of scope.

Choosing a Scaling Factor

Choosing a scaling factor in a way that minimizes accuracy loss is non-trivial. The scaling factor plays a large role in quantization because, as previously discussed, the scaling

factor determines the distance between quantized values, i.e., the step size. The scaling factor is defined as

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{2.6}$$

where β is the upper bound and α is the lower bound of the range of quantized values, and b is the quantization bit width. $[\alpha, \beta]$ is also known as the *clipping range*.

Determining the clipping range is known as *calibration*. When $\alpha = -\beta$, we are employing *symmetric quantization*. Using symmetric quantization means the zero point Z is 0, which is computationally less expensive at inference time, especially since the quantization function is now merely $Q(r) = Int(r/S)$. While symmetric quantization is more inexpensive, it can occur at the cost of accuracy, in particular for NNs that have an imbalance in their weights or activations. This is apparent in NNs that use ReLU activations, a common activation function, that results in only non-negative activation values. In response, we can select a clipping range such that $\alpha \neq -\beta$ that better reflects the imbalance of our weights and/or activations. This is known as *asymmetric quantization*. More information on calibrating clipping ranges can be found in [77].

Dequantization

To go from the quantized value to its original real value, we use a dequantization function, which does the reverse of the quantization function. Dequantization is defined as

$$\hat{r} = S \cdot (Q(r) + Z) \tag{2.7}$$

Note that $\hat{r}$ is not guaranteed to equal the original value r because $Q(r)$ employs a rounding function. The bias introduced by the rounding function introduces some error that is lost and

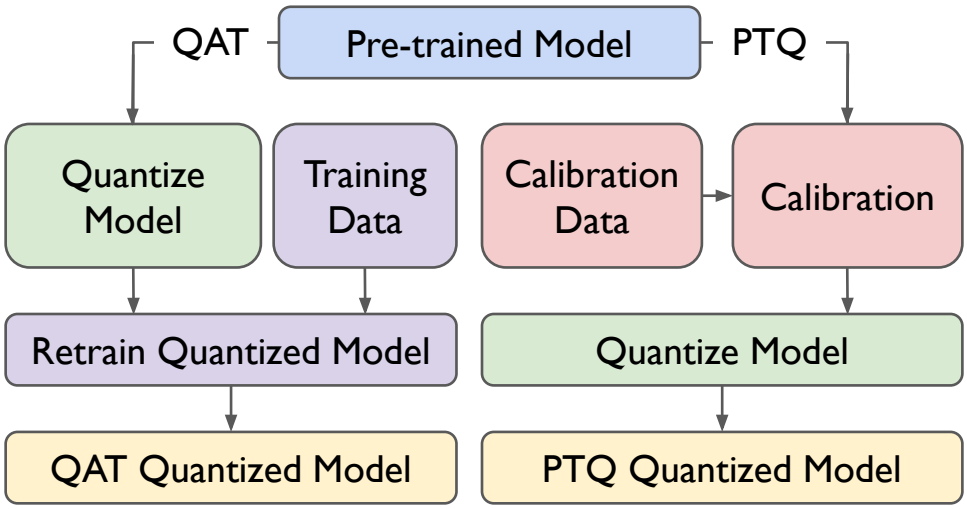


Figure 2.3. How to quantize a pre-trained model via either Quantization-Aware Training (QAT) or Post-Training Quantization (PTQ) [77]. Calibration data can be either a subset of training data or a small set of unlabelled input data. Refer to Section 2.1.2 to review calibration.

cannot be recovered by the dequantization function.[1]

2.1.3 Quantization Procedures

While it is possible to train a quantized NN from scratch, the majority of research has shown that starting with a pre-trained model to be more effective at minimizing accuracy loss when quantizing a NN. There exists two main methods of quantizing pre-trained NNs: quantization-aware training (QAT) and post-training quantization (PTQ). QAT involves quantizing a NN and then retraining it so that it has a chance to adjust and learn according to the newly quantized values. Sometimes a sufficient amount of training data is not readily available, so we use PTQ, which entails quantizing a NN without any extra training. Figure 2.3 gives an overview of QAT and PTQ.

Quantization-Aware Training

Quantization-Aware Training involves retraining a model with quantized parameters so that it can learn and correct any quantization bias that often results from rounding errors. To

[1] Some quantization work has focused on accounting for this rounding error by introducing bias into the NN's parameters [64, 216].

perform QAT, we typically

1. quantize the weights,
2. perform a forward training pass through the model using floating point inputs and activations,
3. perform a backward pass through the quantized model using floating point gradients,
4. update the weights using the floating point gradients, and proceed back to step (1).

until the NN converges.

Regarding step (3), we note that the weights are quantized using a quantization function like the one defined in Equation (2.5), which is non-differentiable. Thus, it is necessary to approximate the gradient. A popular heuristic used for this approximation is called the Straight-Through Estimator (STE) [19]. The STE approximates the non-differentiable parts of the quantization function using the identity function. While it is not clear why the STE works, it has been empirically shown to be effective, except for extreme bit widths (e.g., quantizing to 1-bit binary values).

In addition to correcting the quantization bias in weights, other quantization parameters can be learned during QAT, such as the clipping range α and β as defined in Equation (2.6). For instance, Parameterized Clipping Activation (PACT) [43] learns the activation clipping ranges for each activation layer during QAT.

The main trade-off of the high accuracy afforded by QAT is the long training time, e.g., hundreds of epochs, and the associated computational retraining cost. Moreover, a sufficient amount of training is required to prevent ove-fitting. Nevertheless, this long training time is often worth it for models that will be deployed for long periods of time, wherein the hardware and energy efficiency gains more than make up for the retraining cost.

Post-Training Quantization

QAT not possible when the training data is too sensitive or unavailable at time of deployment. At times like these, Post-Training Quantization is an attractive option for fast quantization. Oftentimes, PTQ uses a small set of calibration data, such as unlabeled input data, to help with choosing the best quantization parameters, e.g., scaling factor. In the past, Post-Training Quantization penalized all quantization errors equally, which is less than ideal because some quantization errors contribute more to altering classification than others. As a result, researchers turned towards QAT to fix the error introduced by quantization by retraining the entire model, as previously discussed. Recently, however, people have been revisiting PTQ in an attempt to quantize in a smarter way when there is limited data available.

To apply PTQ to a NN, we typically

1. calibrate the quantization parameters based on any available calibration data, and then
2. quantize the model.

When there is a lack of training data and fast quantization is needed, PTQ is usually the best option. This is often at the cost of achievable precision in that at least 4 bits are required [140]. Even with 4 or more bits, PTQ tends to be less accurate than QAT, so it is the less popular option because with the same bit precision, QAT achieves higher accuracy.

2.2 Computer architectures for edge NNs

There are a few main approaches to implementing edge NNs fully on chip that are studied in this dissertation: dataflow-style architectures, 2D processing element (PE) array-style, and LUT-based architectures. In this section, I review these styles of architecture that target ASICs and FPGAs, though I primarily target FPGAs in Chapter 3 and Chapter 4. FPGAs are a desirable target for edge NN accelerators because they can be configured to use any arbitrary bitwidth datatype [99], fully exploiting the benefits of quantization to minimize resource utilization and

latency. The main cost is programmability, as FPGAs are notoriously more difficult to program than CPUs and GPUs. Yet, this is worth it if efficiency is a top priority.

2.2.1 2D PE Array-style Architectures

The most popular architecture for NN accelerators tends to involve a 2D array of PEs. In fact, many industry accelerators feature a 2D PE array-style architecture, including the Google TPU [98], Amazon's Inferentia [9], and Meta's MTIA [66]. The 2D PE array-style architecture is popular because it can be used to implement a wide variety of NNs. Although a given accelerator may only target say convolutional NNs, users can run a wide variety of convolutional NNs on the same accelerator. This is unlike the dataflow-style architecture, which must be specially designed for only one specific NN. Thus, they sacrifice some programmability and flexibility for more generalized support. The 2D PE arrays feature efficient data reuse and are optimized for dense matrix multiplications, the main computational workload in many NNs [4]. Most 2D PE array architectures that have been introduced over the years differ in their data reuse pattern, such as input, output, row, and weight stationary [4]. Prior work has introduced its accelerator architectures to be deployed as ASICs, such as DaDianNao [37], ShiDianNao [59], Eyeriss [38], whereas other work has focused on reconfigurable FPGA-based implementations such as Caffeine [214] and CGRA4ML [4].

2.2.2 Dataflow-style Architectures

The dataflow-style architecture dedicates resources to each layer of the NN, spatially laying out the NN across the hardware. This way we can stream the data through the layers, pipelining the execution of the NN layers through the dataflow style and achieving high throughput and low latency [102]. This architecture is commonly used in FPGA-based NN accelerators such as FINN [191] and hls4ml [60]. Spatial dataflow-style architectures trade resources for performance, as precious resources are dedicated per layer.

FINN [191] was developed by Xilinx Research Labs in 2017 and targets NNs quan-

tized to ultra low bitwidths such as binary NNs. Such low bitwidths allow FINN to compute batchnorm-activations using a thresholding unit, saving significant resources and speeding up computation [191]. Additionally, when targeting binary NNs, FINN implements the dot products (the core computation behind matrix-vector multiplications found in NNs) using XNOR and popcount operations. FINN combines these two optimizations in its Matrix-Vector-Threshold Unit (MVTUs), forming the core computation unit of the architecture. The MVTUs can also be folded to save resources, decreasing parallelism by reusing the same MVTUs across multiple data. FINN has led to much follow-on work, expanding the kinds of NNs it supports and optimizing its architecture further [7, 21, 23, 97, 99, 103, 164].

hls4ml [60] was developed by high energy physicists in 2018 to make building edge NN accelerators for physics experiments more accessible to scientists. Since hls4ml was created to target physics experiments that have strict latency and throughput requirements, it also uses a dataflow-style architecture to achieve high performance. hls4ml's strength is its usability because its Python framework for building NN accelerators is easy to use. It exposes a "hardware knob" called *reuse factor* that users can tune to trade some performance for lower resource utilization, similar to folding in FINN. With a reuse factor of 1, hls4ml fully unrolls all multiplications in each NN layer, fully parallelizing computation. With a reuse factor of k, hls4ml reuses the same hardware for k multiplications, reducing parallelism by a factor of k and thus saving resources at the cost of super-linearly increasing latency. Latency increases super-linearly because extra control logic is needed to manage time sharing the reused hardware, which also makes FPGA routing more complex. hls4ml has been improved on over the years, as computer scientists and high-energy physicists have worked together to expand its NN support and further optimize its performance [3, 63, 74, 85, 147, 175].

2.2.3 LUT-based NN Architectures

The LUT-based NN architecture has inspired a growing field of research that uses LUTs as the fundamental basis to implement NNs on FPGAs. LUTs are a desirable target because

they are one of the main resources featured on FPGAs. Even more, they eschew the need for arithmetic units, replacing multiply-accumulate operations with lookups, which is significantly more efficient in terms of latency and throughput. This is particularly desirable for edge NNs that have nanosecond-level latency requirements. The primary challenge with LUT-based NN architectures is that their resources grow exponentially with respect to the number of LUT inputs, which makes it difficult to scale up the model size by increasing the number of parameters. This is an issue because larger models tend to be more accurate, making it difficult to achieve high accuracy with LUT-based architectures. Recent work has sought to address this issue by improving the expressiveness of the LUTs, which allows for fewer LUTs to be used to achieve higher accuracy, thus improving scalability.

LUTNet [195] uses LUTs as the fundamental inference operator to implement NNs on FPGAs. LUTNet seeks to improve binary NNs' reliance on XNOR operations by replacing them with more expressive, learnable K-LUT Boolean operations. While LUTNet improves NN resource efficiency on FPGAs, the number of parameters in LUTNet scales exponentially with respect to the number of LUT inputs, making it difficult to improve accuracy by increasing model size through parameter count.

Weightless NNs (WNNs) are another class of LUT-based NNs [8, 133, 177, 178]. WNNs consist of neurons, but they do not use weights to determine how they respond (i.e., they are weightless). WNN neurons are made up of logical LUTs that represent Boolean functions. WNN neurons concatenate their inputs, which form an address they use to perform a lookup in the associated LUT. WNNs learn by starting with an N-input, 2^N-output LUT that contain 1-bit entries initialized to all zeroes [177]. Then for each training input that has a value of one, it flips the associated LUT entry to one. These logical LUTs can then be mapped to FPGA physical LUTs via a logic synthesis tool. WNNs suffer from exponential scaling with respect to the number of LUT inputs as well as difficulty in generalizing to unseen input data, as they primarily memorize patterns [177]. Recent work, such as differentiable WNNs (DWNs) [17] improve WNN generalization by making them differentiable so that they learn connections between LUTs

rather than using fixed random setups.

LogicNets [190] and NullaNet [143, 144] fully encapsulate a neuron's operation within a logical LUT. Given a trained quantized NN, LogicNets feed all possible quantized inputs to a neuron and captures the quantized outputs it computes, while NullaNet does so in a lossy manner. These inputs and outputs are then enumerated in a logical LUT and then implemented as physical LUTs by a logic synthesis tool.

PolyLUT [10], PolyLUT-Add, NeuraLUT [11], and NeuraLUT-Assemble [12] are successors of LogicNets. Each method improves the accuracy of LogicNets by attempting to capture more complex functions within the logical LUT. PolyLUT learns a NN that is a piece-wise polynomial function whereas NeuraLUT encapsulates an arbitrary NN within a logical LUT. PolyLUT-Add [119] sums smaller PolyLUT logical LUTs together to build larger, more complex LUTs. NeuraLUT-Assemble assembles smaller sub-neural networks to improve neuron connectivity, which is a common issue in LUT-based NNs that leads to accuracy loss. NeuraLUT-Assemble also introduced skip connections into its design to improve training. These designs improve the scalability of LogicNets because fewer logical LUTs are needed to achieve higher accuracy, leading to fewer physical LUTs and lower latency.

Kanele [90] is a recent LUT-based architecture that maps Kolmogorov-Arnold Networks (KANs) to FPGAs. KANs learn one-dimensional splines with fixed domains as edge activations, which can express a variety of functions and map well to FPGA LUTs. Kanele introduces its own quantization and pruning techniques to further improve its latency and throughput on FPGAs.

2.3 Fault tolerant neural networks

In this section, we review how faults manifest, how prior work has evaluated NN resilience in both software and hardware, and how researchers have protected their NNs and associated hardware from faults.

2.3.1 How faults manifest

Particles such as protons and heavy ions cause Single Event Effects (SEEs) when they hit hardware, which can be destructive or non-destructive [46]. Destructive SEEs permanently change how a device operates. For example, stuck-at faults create a permanently stuck bit in a memory element, such as a register. Non-destructive SEEs temporarily change the state of a device. For instance, single event upsets cause "soft" errors that a device reset or rewrite can resolve [46]. SEEs often manifest as a single bit flip or multiple bit flips (e.g., two or more upsets affecting adjacent memory cells known as a multi-cell upset [132]). It is possible for a single particle strike to cause multiple bit flips.

2.3.2 Resilient NN Evaluation

Much work has been done to evaluate the resilience of NNs to faults both at the software and hardware levels [5]. A popular approach is known as a fault injection campaign, which simulates a fault in software [40, 111, 126, 130, 142, 145, 159] or hardware [69, 159] and runs a number of inputs through the NN to assess performance. This process repeats until the campaign is complete based on statistical models [162, 163], specialized fault injection algorithms [40, 200, 201], and so on as determined by the researcher. Fault injection campaigns are computationally intensive because they require simulating a vast number of fault scenarios, particularly given the thousands to millions of parameters and operations in a neural network (NN) [185]. These campaigns can either be performed at the software or hardware level. In addition, researchers have used heuristics and ML modeling to understand a NN's resilience at the software level as well as physical irradiation at the hardware level, e.g., shooting neutron beams at an FPGA [5, 132]. Each approach has its tradeoffs.

Software-level analysis is usually faster than hardware-level analysis, but it assesses many faults at an abstract level and misses (or even mischaracterizes) the behaviors of certain faults [108]. Although hardware-level analysis can evaluate faults at a finer granularity, it is often

slow and difficult to scale up. Recent work has sought to bridge the gap between the two [88, 108], and our project seeks to contribute on this front even further, in particular with respect to edge NNs accelerators targeting ASICs and FPGAs. Many past works have focused on faults in CPUs and GPUs that are application-agnostic or take a security-driven approach [108, 116, 138, 193], but with the proliferation of edge AI accelerators, we will focus on the challenges unique to the edge, such as its strict resource and latency constraints.

Software-level evaluation

Prior work has proposed techniques to accelerate fault injection [40, 69, 129, 130, 200]. However, many of these studies focus on NNs that depend on off-chip memory, which is less commonly used in edge applications with strict latency and throughput constraints [202]. These studies assume a NN's weights are fully protected by error correction codes (ECC) or parity. As a result, they primarily examine faults in NN activations while overlooking faults in weights [15, 40, 113].

Researchers have also introduced heuristic approaches to estimate NN fault sensitivity based on properties such as gradients [44, 127, 169], statistical fault injection [162, 163], and error propagation behavior [170]. Because heuristics are inherently lossy, they generally offer lower accuracy than full FI campaigns but are substantially more efficient, effectively trading some accuracy for improved speed.

In addition, ML models have been proposed to predict which bits in an NN are most likely to cause faulty outputs [36, 186, 187, 215]. These approaches derive fault-relevant features from NNs and train ML models using a relatively small set of labeled ground truth data to identify vulnerable components. Although these methods achieve reasonable accuracy, they are costly to scale, as a separate ML model must be trained for each new NN under evaluation.

Hardware-level evaluation

Researchers evaluate the resilience of NN hardware at different levels of abstraction, including microarchitectural simulation [35, 111], HLS [104], RTL [73, 75, 101, 161, 166], gate level [157], and physical irradiation [5, 132, 180]. Some frameworks [42, 210] even enable injecting faults at many different levels of abstraction. One unique aspect of studying faults in FPGA-based NN accelerators is that faults can manifest in the configuration random access memory (CRAM), where the bitstream resides, and in the design itself [14, 158, 188]. Faults in the CRAM will have drastically different ramifications compared with faults in the design only. As researchers inject faults at lower levels of abstraction, they are able to mimic faulty behavior more accurately. But this comes at the cost of longer fault injection times and significantly more faulty scenarios. For instance, an RTL design contains billions of fault injection locations when considering the permutations of every location and cycle at which to inject a fault [166]. Even more, experiments involving physical irradiation are particularly expensive and complex, where depending on the irradiation settings, e..g, how many neutrons per cm^2, only a few hundred events are observed [42, 71]. In order to gain more data and form a more accurate picture, many days of irradiation may be needed, increasing the cost of the experiment.

2.3.3 NN Fault Mitigation

Mitigations exist at the transistor, circuit, architecture, and/or algorithmic level. Circuit-level mitigations are the most energy efficient, according to [41]. But implementing circuit-level mitigations seems prohibitively expensive/difficult because DSE is not usually making design decisions at the circuit level. We only have control at the architectural level and above—other than selecting which off-the-shelf platform that features different kinds of transistor/circuit hardening already.

Prior research has explored a variety of strategies to protect NNs and optimize those protection mechanisms. Some proactive methods at the algorithmic level, such as fault-aware

training [22, 149, 150, 174, 213], enhance resilience by training a NN's weights to better tolerate faults. One method occurs during compile time, reducing the number of registers used for certain functions, reducing the overall critical area exposed to faults [71, 72]. Other techniques take a more reactive approach at the architecture or algorithmic level, including dual modular redundancy (DMR) and TMR [81, 179], selective DMR/TMR [22, 75, 105, 129], error correction [75], and activation clipping [39, 91, 121].

Activation clipping is a particularly appealing low-cost solution. It works by profiling the model to determine the expected numerical range of activations and then constraining any activation that falls outside this range due to hardware faults. This technique is especially effective for float32 NNs, given the wide dynamic range of float32 representations. However, its benefits are diminished in quantized models, which are most commonly featured at the edge, since quantized data types such as int8 inherently restrict activation ranges [129].

Chapter 3

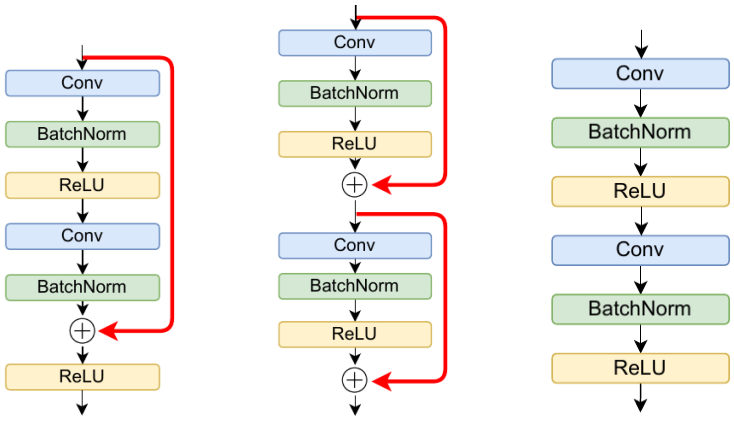
Tailor: Altering Skip Connections for Resource-Efficient Inference

In this chapter, I present Tailor, a hardware-aware training algorithm that alters skip connections to be more resource-efficient for on-chip inference while maintaining accuracy. This chapter provides insights on how to use NN training techniques such as knowledge distillation to massage an NN's topology to better map to its target architecture and hardware, in this case dataflow and 2D PE array-style architectures for FPGAs.

3.1 Introduction

Convolutional neural networks (NNs) often rely on skip connections—identity functions that combine the outputs of different layers—to improve training convergence [86, 192]. Skip connections help mitigate the vanishing gradient problem [20, 78] that occurs when training large CNNs, which helps increase the network's accuracy. Skip connections allow NNs to have fewer filters/weights than architectures that lack skip connections [86], such as VGG [173].

However, skip connections are generally detrimental to hardware efficiency. They have an irregular design that is ill-suited for hardware acceleration. This is due to their long lifetimes, which span several NN layers, increasing memory utilization and bandwidth requirements. This is particularly true in ResNets [86], which introduced skip connections that spanned across five layers: two convolutions, two batch normalizations (BNs), and a ReLU activation [79, 141] (see



(a) Traditional **(b)** Shortened **(c)** None

Figure 3.1. Neural networks with traditional skip connections, like ResNet (a), have inefficient hardware implementations because the skip connection data must be preserved in memory during five layers of computation. This irregular topology increases memory resources and bandwidth. A more regular topology with reduced skip connection lifetimes would use fewer resources. Tailor achieves this by shortening skip connections (b) or by eliminating them completely (c). Skip connections are in red.

Fig. 3.1a). The skip connection involves minimal computation—it is either the identity or a 1×1 convolutional layer for scaling—but it extends the necessary lifespan of the input data. Thus, we must store skip connection data for the duration of time needed to compute the five NN layers. In total, a model's skip connection data accounts for $\sim$10% of its memory bandwidth either on or off chip. Buffering skip connections on chip increases on-chip memory utilization, whereas moving them off chip not only increases off-chip memory bandwidth but also requires extra control logic for scheduling [123, 124].

Optimizing skip connections requires careful *hardware-software codesign*. Skip connections are crucial for model convergence; naively removing them to reduce hardware resources leads to low accuracy [136, 211]. Instead, we must codesign how the model is (1) trained and (2) implemented in hardware to achieve a model that is both accurate and resource-efficient.

We develop Tailor, a codesign method that gradually alters a NN's skip connections during training to produce a highly accurate and resource-efficient NN. Our results in Sec. 3.4 show that Tailor can remove or shorten skip connections to achieve topologically regular NNs (Fig. 3.1b

and 3.1c) that substantially reduce hardware resources, reduce memory bandwidth, and increase performance with minimal to no accuracy loss.

Tailor takes an *existing* pre-trained model and reduces the hardware complexity of its skip connections with minimal to no accuracy loss. Moreover, Tailor exploits the flexiblity of the FPGA architecture to customize the skip connection memories, which is not possible on a GPU or CPU. Tailor accomplishes this dynamically during retraining in one of two ways: (1) SKIPREMOVER removes the skip connections altogether (Fig. 3.1c) to eliminate all associated hardware complications or (2) SKIPSHORTENER shortens each skip connection by splitting it into multiple shorter ones (Fig. 3.1b).

We evaluate Tailor's applicability and benefit on ResNets [86, 87] and QuartzNets [106]: two important classes of NNs that contain skip connections of varying lengths. We also study implementing skip connections with an on-chip, dataflow-style FPGA architecture using hls4ml [3, 63] and a 2D array of multiply-accumulate processing elements. Tailor reduces resource utilization of hls4ml architectures by up to 34% for BRAMs, 13% for FFs, and 16% for LUTs. Tailor increases the performance of 2D array architecture by 30% and reduces memory bandwidth by 45%.

Tailor's hardware-software codesign approach reduces hardware complexity and resources by altering skip connections dynamically during retraining. Our contributions are:

- the Tailor software methodology of removing or shortening skip connections from existing NNs with minimal to no loss in accuracy,
- the Tailor hardware designs that exploit FPGA-specific architecture optimizations, which are not possible on GPU/CPU, to produce less resource-intensive skip connection implementations,
- experiments demonstrating that SKIPSHORTENER and SKIPREMOVER models are implemented more efficiently with better performance and resource utilization than their traditional skip connection counterparts,

- and public release of the Tailor hardware-software codesign framework [2].

3.2 Background

3.2.1 Removing Skip Connections

While skip connection removal has been studied before [52, 114, 136, 211, 212], prior work is lacking in several ways: (1) preliminary work [136, 211, 212] only studies shallow models (up to 34 layers); (2) Li et al. [114] do not remove all of the skip connections in the models they evaluate; (3) Ding et al. [52] and Li et al. [114] both have limited architectural evaluations (e.g., GPU & mobile) that do not consider the highly customized skip connections memories enabled by FPGAs; and (4) Ding et al. [52] require starting with an entirely new NN topology whose skip connections are removable.

Monti et al. [136] start with a standard ResNet and introduce a new training method. This method uses an objective function that penalizes the skip connections and phases them out by the end of the training. This technique has only been applied to smaller ResNets (18 to 34 layers) with a small decrease in accuracy between 0.5 and 3%.

Zagoruyko and Komodakis [211] also develop a method for removing skip connections in a NN. They replace skip connections with Dirac parameterization, creating a new NN called DiracNet. The Dirac parameterization is shown in Eq. 3.1,

$$\text{DiracNet [211]: } y = \sigma(x + Wx) \tag{3.1}$$

$$\text{ResNet [86]: } y = x + \sigma(Wx)\,, \tag{3.2}$$

where $\sigma(\cdot)$ is the nonlinear activation function, W is the layer weight matrix, x is the layer input, and y is the layer output. For ease of comparison with ResNets, Eq. 3.2 is simplified to show only one convolutional layer. In fact, skip connections in ResNets hop over more than one convolutional layer, while in DiracNets, the identity mapping is over one single convolutional layer. Therefore, the weights and the identity mapping of the input can be folded because

$x + Wx = (I + W)x$. This change requires DiracNets to widen the NN layers in the ResNets that they started with. The authors showed that their technique could be used to create models with up to 34 layers. Although it works for shallower models, DiracNets show a decrease in accuracy between 0.5% and 1.5% compared to ResNets. In contrast, SKIPREMOVER eliminates skip connections without widening the layers in the NN and does not need to make this accuracy tradeoff.

Li et al. [114] develop residual distillation (RD), which is a modified knowledge distillation framework. RD starts with a ResNet as the teacher and a plain CNN without skip connections as the student. Unlike standard knowledge distillation, RD passes the teacher's gradients to the student during training. This differs from Tailor because RD starts with a student model without skip connections, whereas Tailor *gradually* modifies a model's skip connections every few epochs during training without sharing gradients. Moreover, while RD removes all skip connections from models evaluated on simpler datasets like CIFAR-10 and CIFAR-100 [107], it fails to remove all skip connections in its ImageNet evaluation, leaving 18% of them in the network, which is a costly choice. In our ImageNet evaluation (see Sec. 3.4.1), our SKIPREMOVER method removes all skip connections with minimal accuracy loss.

Ding et al. [52] introduce a new model architecture RepVGG, which trains using 3×3 convolutional layers that are each skipped over by both a 1×1 convolution and an identity connection. At inference time, these connections can be re-parameterized into the parameters of the 3×3 convolutional layers. While RepVGG is more accurate than our SKIPREMOVER model, it requires starting from their specialized training model architecture. This is costly to developers who have already trained a model with skip connections on their dataset. Similarly, transferring a pre-trained RepVGG model to a new dataset via transfer learning can be time-consuming given the many different methods [151, 198, 219] to evaluate. As such, Tailor is ideal for these developers because it modifies the skip connections of an *existing* pre-trained model to be more resource-efficient with minimal to no accuracy loss. Developers can leverage the training they have already done and need not start from scratch with a brand new RepVGG architecture.

3.2.2 Simplifying Skip Connection Hardware

ShuffleNet [122], DiracDeltaNet [205], and DenseNet [93] simplify skip connections by making them *concatenative*, i.e., they concatenate, rather than add, the skip connection data to the output of a layer. Concatenative skip connections take advantage of the fact that spatially consecutive memory accesses are typically faster than random accesses. This concatenation and off-chip data movement is possible using a simple controller (e.g., DMA engine).

Tailor uses two techniques to simplify the skip connection hardware. SKIPREMOVER eliminates all logic and memory needed for a skip connection, making them less expensive than concatenative skip connections. Careful retraining allows skip connection removal in smaller networks with no degradation in accuracy. For larger networks, SKIPSHORTENER shortens the additive skip connections. By reducing their lifespans, the hardware implementation requires fewer resources. SKIPSHORTENER is not necessarily simpler than ShuffleNet [122] or DiracDeltaNet [205]. However, these concatenative skip connections have only been evaluated on image classification and object detection tasks. In our work, we demonstrate our SKIPREMOVER and SKIPSHORTENER methods on multifarious NNs and classification tasks, namely image-classifying ResNets of varying depths, DNA-basecalling QuartzNet-5×5, and automatic-speech-recognizing QuartzNet-10×5. With respect to DenseNet [93], SKIPSHORTENER ResNets use much less memory and bandwidth because DenseNet relies on significantly more skip connections throughout its NN. Given a NN with L layers, DenseNet needs the memory and bandwidth to execute $L(L+1)/2$ concatenative skip connections, compared with SKIPSHORTENER ResNets' mere L skip connections. With so many more skip connections, DenseNet is more expensive for hardware than SKIPSHORTENER ResNets.

Finally, all these techniques simplify skip connection hardware from the outset, building their models with modified skip connections and then training them from scratch. Tailor differs because its hardware-aware training method *dynamically* alters the skip connections every few epochs during training, taking advantage of what the NN has learned with skip connections.

Thus Tailor allows the NN to gradually adapt to shortened skip connections (SKIPSHORTENER) or none at all (SKIPREMOVER).

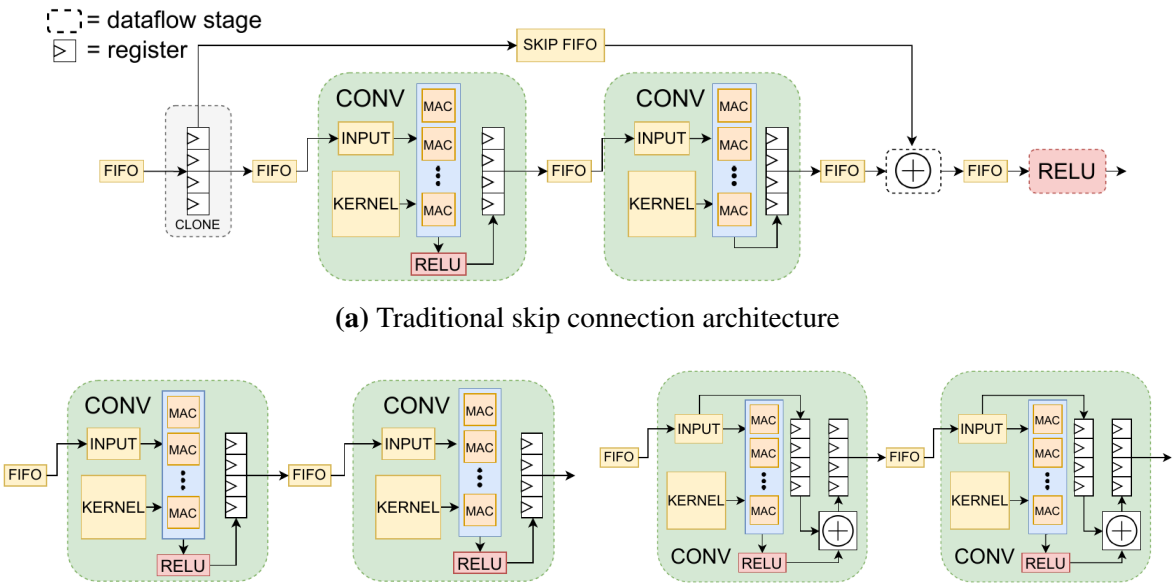
3.3 Tailor

Skip connections are important for training (to provide good accuracy), yet complicate implementation (requiring additional hardware resources and reducing performance). Tailor modifies skip connections to make their hardware implementation more efficient. Tailor uses a retraining method that gradually alters the network, resulting in little to no loss in accuracy.

3.3.1 Hardware Design

Fig. 3.2 shows three hardware implementations for NNs with traditional, shortened, and no-skip connections. The implementations correspond to accelerators produced by hls4ml—a tool that translates Python models into high-level synthesis code [60]. hls4ml creates a separate datapath for each layer and performs task-level pipelining across the layers. The layers communicate using FIFOs (AXI streams). Everything encapsulated by a dashed line resides in one pipeline stage. The inputs are fed into the architecture using a stream, and the results are given as an output stream. The weights are all stored on-chip, and all the internal results are stored on-chip. We evaluate each of these designs on FPGA later in Sec. 3.4.2 along with another style of architecture using a 2D array of processing elements. Tailor allows us to trade off between accuracy, performance, and resource usage through co-design of the neural network using hardware-aware training.

Fig. 3.2a shows the hardware needed to implement a single ResNet's skip connection. Note that in all of the designs shown in Fig. 3.2, we fuse the batch normalization parameters with the kernel, as is commonly done [96]. To be low latency and high throughput, the design uses task-level pipelining (i.e., the HLS dataflow pragma) for each NN layer, or a small grouping of layers, and streams the data between each dataflow stage using first-in first-out buffers (FIFOs). Since FIFOs can only be read from once, skip connections complicate the design. We must



(a) Traditional skip connection architecture

(b) No skip connection architecture

(c) Shortened skip architecture

Figure 3.2. The hls4ml hardware architectures for traditional, shortened, and no skip connections. hls4ml pipelines each layer as is common for latency-critical tasks in resource-constrained environments [3, 63]. The three architectures correspond to a ResNet implemented with a traditional skip connection (a), shortened skip connections (b), and no skip connections (c). Note that we combine the batch normalization parameters with the kernel, as is commonly done [96].

spend a dataflow stage on cloning the skip connection data from its input FIFO into two other FIFOs so that it can be read twice for its two datapaths. The first path goes through a collection of convolutional and ReLU layers, and the second stores the data in a FIFO exclusive to skip connections (skip FIFO). Once the data has gone through the first path, we read from the skip FIFO to perform the addition to complete the skip connection's identity function. As such, implementing a skip connection on chip requires several extra FIFOs for handling the skip connection data, and this in turn increases on-chip memory resource utilization.

Ideally, we would eliminate the skip connections. As seen in Fig. 3.2b, without skip connections, we cut the number of dataflow stages in half (no more Clone, Add, or ReLU stages) and use less than half of the requisite FIFOs compared with Fig. 3.2a. All we need to do is pass the data through the convolutional and ReLU layers. This reduces resource utilization by up to 16% (see Sec. 3.4.2).

It may not be possible to remove the skip connections because they are essential for training convergence. In these cases, shortening the skip connections can simplify their hardware implementation. Fig. 3.2c shows a modified network with shortened skip connections such that *each skip connection's lifespan resides within a single dataflow stage*. We do not need additional dataflow stages to clone skip connection data. The shorter lifespans allow the shortened skip connections to be stored in *shift registers*, which can be implemented using the more abundant FFs as opposed to BRAMs, which is used in the traditional skip connection's hardware design. In this way, we exploit the short skip connections' lifetimes and use simpler, more efficient hardware memories to implement them (see Sec. 3.4.2). As such, we achieve a similar architecture to the version without skip connections (Fig. 3.2b), and similarly reduce resources spent on additional dataflow stages and FIFOs in Fig. 3.2a. SKIPSHORTENER is thus more resource-efficient than the traditional skip connection design. In fact, SKIPSHORTENER provides a tradeoff between the SKIPREMOVER and traditional designs because it uses more resources than SKIPREMOVER but less than the traditional one (see Sec. 3.4.2). But as we later show in Sec. 3.4.1, SKIPSHORTENER maintains accuracy in cases where SKIPREMOVER accuracy drops off. Thus, SKIPSHORTENER allows for design space exploration to balance accuracy and resource usage.

When used with hls4ml, Tailor reduces resource consumption without changing the performance. This is a consequence of hls4ml's dataflow design; the resources we remove are not on the critical path—they are operating in parallel to the critical path. A dataflow design uses task-level pipelining, so reducing the resources spent on stages not on the critical path does not help or hurt overall throughput. Based on our Vivado co-simulation results, the clone stage executes in microseconds while the convolutional layer executes in milliseconds, an order of magnitude difference. Therefore, removing the clone buffer (Fig. 3.2b) or implementing it more efficiently (Fig. 3.2c) will not affect the overall dataflow latency because its latency is an order of magnitude less than the convolution's latency. This means Tailor's resource reductions do not increase or decrease latency or throughput for this architecture style, as later shown in Tab. 3.7.

Another prevalent style of FPGA CNN architectures instantiates a 2D processing element

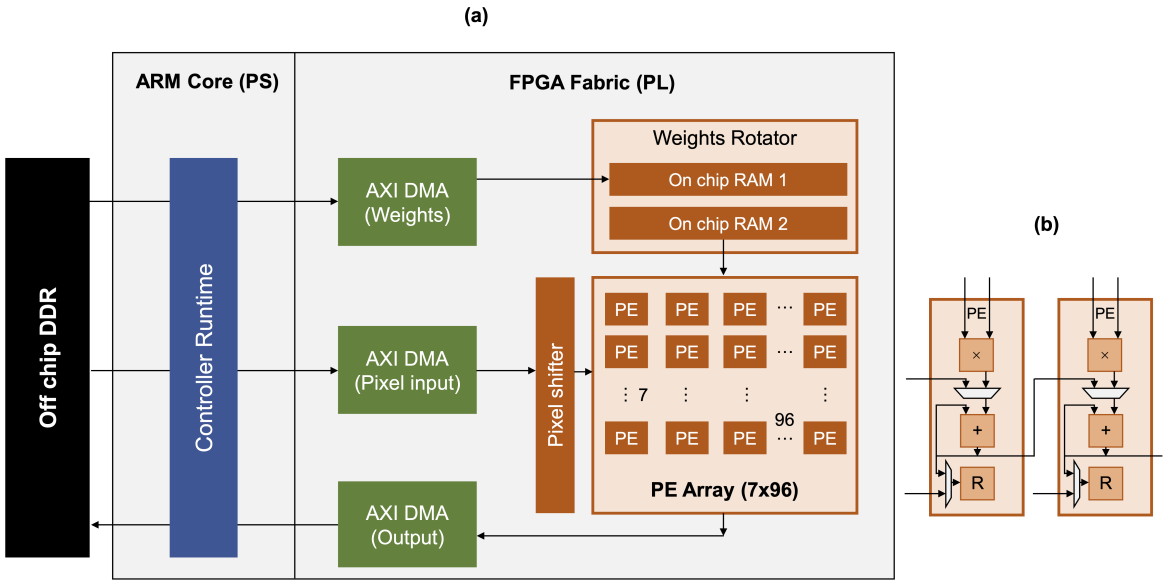


Figure 3.3. (a) A Reconfigurable DNN Architecture synthesized on a ZCU102 FPGA development board. The architecture has a 2D array of processing elements that are iteratively programmed to compute layer operations. The controller runtime programs the DMA engines to load off-chip inputs and weights and store the intermediate and final results off-chip. (b) The processing elements (PE) are a multiply-accumulate datapath.

(PE) array and iteratively programs the convolutions and other operations onto that PE array. We call this style of computation a Reconfigurable DNN Architecture. Fig. 3.3 provides an example architecture used in our experiments. We build this architecture using DeepSoCFlow [1]. Following the taxonomy described in [100], the reconfigurable DNN architecture is a 2D array of processing elements that optimally perform standard convolution and matrix multiplication with high data reuse. The dataflow is primarily output stationary while prioritizing maximal weight reuse and also reusing inputs to an extent. The engine performs fixed-point computations, where the input, weight, and output bit widths are adjustable as synthesis parameters, along with the number of rows and columns of processing elements. The weights rotator prefetches the weights of the next iteration into one block of on-chip memory while the other bank delivers weights, rotating them hundreds of times for maximal data reuse. The Pixel Shifter shifts perform vertical

[1] https://github.com/abarajithan11/deepsocflow

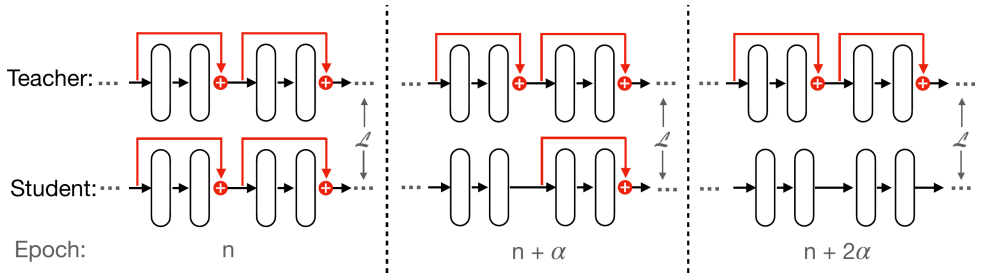
convolution. Partial sums are shifted to the PE on the right to compute horizontal convolution. The results are streamed out through the output DMA to the off-chip memory. The runtime controller would perform the residual addition, quantization, and activation on the processing system side while the engine computes the next iteration. Our implementation uses the ARM processor available in the Zynq chip. FPGAs without processors could instantiate a softcore processor to perform the controller runtime operations.

The Tailor optimizations have different effects on the Reconfigurable DNN architecture as compared to hls4ml architecture. The Reconfigurable DNN architecture computes skip connections by loading input data from off-chip memory and performing the required operations upon it (addition, convoluation) Thus, unlike in hls4ml, removing a skip connection does not change the architecture; instead it changes how computations are mapped to that architecture. Skip connection removal eliminates the need to fetch the skip connection data and perform the associated convolution and addition operations. This increases the overall performance as we describe in Sec. 3.4.

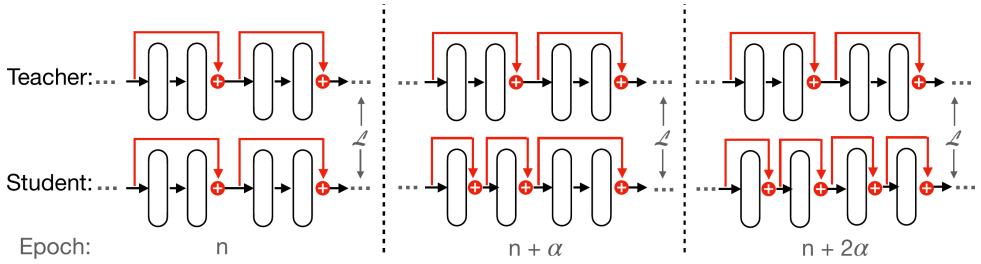
3.3.2 Hardware-aware Training

It is difficult to modify a NN's skip connections without reducing accuracy. Naively removing all skip connections before or after training a NN is detrimental to its accuracy. Instead, Tailor consists of two training algorithms, SKIPREMOVER and SKIPSHORTENER, that gradually alter a NN's skip connections on the fly—removing or shortening them every few epochs—in order to make them resource-efficient. Gradually altering the model during training tempers the performance drop of removing or shortening the skip connections, yielding minimal to no loss in accuracy as well as significant advantages in the hardware implementation, as described above.

Tailor's iterative learning approach finetunes the altered NNs using a compression method known as *knowledge distillation (KD)* [89]. KD distills the knowledge of a larger, more complex NN (the teacher) into a smaller, simpler NN (the student). While the student model is training, it compares its output to the teacher model's output and thus learns from the teacher to perform



(a) SKIPREMOVER



(b) SKIPSHORTENER

Figure 3.4. Three iterations in the SKIPREMOVER and SKIPSHORTENER algorithms as applied to a ResNet. In this example, skip connections are altered every α epochs and $\alpha|n$. Each pill block represents a set of convolutional, BN, and ReLU layers, and the skip connections are in red. $\mathscr{L}$ is the KD loss function defined in Eq. 3.3. Only the student model is used for inference.

better. KD provides impressive results for compressing NNs for various applications [134, 167, 181]. In traditional KD, the teacher model is already trained, and the student model is trained to match the teacher's behavior by replicating its output. The student achieves this by training with a loss function

$$\mathscr{L} = (1-\beta)\mathscr{G}(\ell, s) + \beta\mathscr{H}(t, s) \tag{3.3}$$

where $\mathscr{G}$ and $\mathscr{H}$ are distance functions, s and t are student and teacher output vectors respectively, ℓ is the correct label vector, and β is a tunable parameter [89].

With this idea in mind, both SKIPREMOVER and SKIPSHORTENER start with two identical pre-trained NNs with traditional skip connections, where one serves as the teacher and the other serves as the student. During the retraining stage, SKIPREMOVER removes a given skip connection every few epochs. SKIPSHORTENER takes a similar iterative approach and, every

few epochs, splits a given skip connection into multiple shorter ones. The skip connections are removed or shortened starting from the first skip connection encountered in the NN (from the input) to the last.

Fig. 3.4 visualizes both SKIPREMOVER's (Fig. 3.4a) and SKIPSHORTENER's (Fig. 3.4b) training algorithms for a ResNet-style NN. During training, we remove (SKIPREMOVER) or shorten (SKIPSHORTENER) one of the student's skip connections every α epochs. If n is divisible by α (as in Fig. 3.4), then at epoch n, the student has had n/α skip connections altered, and we are viewing the next two skip connections to be modified in the student model: the $(n/\alpha)+1$st and $(n/\alpha)+2$nd. At epoch $n+\alpha$, the $(n/\alpha)+1$st skip connection is altered (removed under SKIPREMOVER or split into two shorter skip connections under SKIPSHORTENER). The NN then trains for α epochs so that the student model can improve its weights given the latest model topology. Afterwards, at epoch $n+2\alpha$, the $(n/\alpha)+2$nd skip connection is similarly altered. During the entire skip modification retraining process, the student uses the KD loss function $\mathscr{L}$ defined in Eq. 3.3 to learn from the teacher and the true labels. The teacher's model topology and weights remain fixed during training. Once all skip connections have been altered, the student model continues training under KD for the remaining number of training epochs as defined by the user. Only the student model is used for inference.

Tailor is novel because it *dynamically* transforms skip connections every few epochs during training. This is an instance of *hardware-aware training* because the skip connection are slowly altered specifically to reduce hardware resources, as previously discussed in Sec. 3.3.1. The gradual skip connection alterations allow the NN to take advantage of what it has learned with skip connections, so that it can dynamically adapt to shortened skip connections (SKIPSHORTENER) or none at all (SKIPREMOVER). Alg. 1 describes Tailor's hardware-aware training process.

Algorithm 1: HARDWARE-AWARE TRAINING

```
set alter // REMOVE or SHORTEN
let α = how often to modify a skip connection
teacher = pre-trained model
student = pre-trained model
let current-skip = student's first skip connection from the input side
let current-layers = all layers skipped by current-skip
Function SkipRemover(current-skip):
    // see Fig. 3.4a
    remove current-skip
    return student model's next skip connection from the input side
Function SkipShortener(current-skip, current-layers):
    // see Fig. 3.4b
    Split current-skip into len(current-layers) skip connections
    current-skip = student model's next skip connection from the input side
    current-layers = student's next layers skipped by the new current-skip
    return current-skip, current-layers
for i in epochs do
    if i ≠ 0 and i mod α = 0 then
        if alter == REMOVE then
            current-skip = SkipRemover(current-skip)
        else if alter == SHORTEN then
            current-skip, current-layers = SkipShortener(current-skip,
              current-layers)
    end
    train student using Eq. 3.3
end
save the student model
```

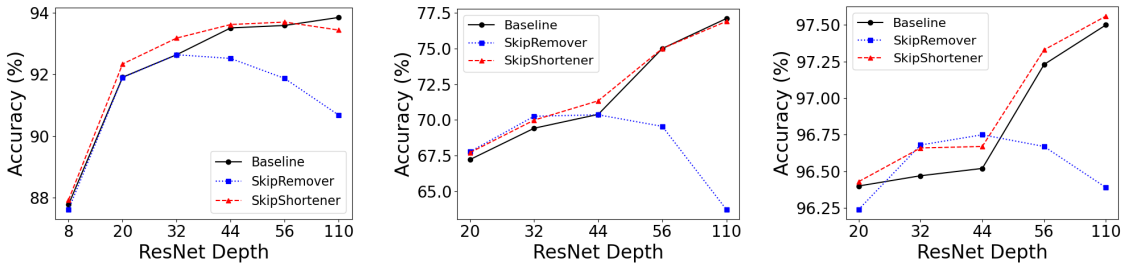
3.4 Results

We evaluate Tailor on two popular kinds of NNs that rely on skip connections: ResNets [86] and QuartzNets [106]. We study the effects of Tailor on model accuracy, quantization, and hardware resource utilization.

3.4.1 Training results

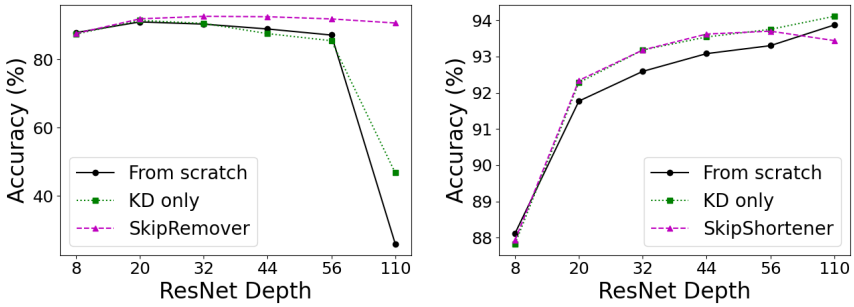
To evaluate how Tailor affects a NN's accuracy, we train ResNets and QuartzNets of varying depths using our SKIPREMOVER and SKIPSHORTENER algorithms in PyTorch [154]. The ResNets range from 20 to 110 layers and are trained on the CIFAR-10 [107], CIFAR-100 [107], and SVHN [146] datasets. We also evaluate ResNet50, which has a different skip connection topology than standard ResNets, on the ImageNet dataset [51]. The QuartzNets span between 29 and 54 layers. Their structure is determined by the number and lifetimes of their skip connections. For instance, a QuartzNet-10×5 has 10 skip connections that each have a lifetime of 5 sets of layers. We train a QuartzNet-5×5 on the Oxford Nanopore Reads dataset [172], a DNA basecalling task. We also train a QuartzNet-10×5 on the LibriSpeech dataset [152], an automatic speech recognition (ASR) task, which converts speech audio to text. ASR tasks are assessed using word error rate (WER), which measures the percent of words that the model predicted incorrectly. In all of our ResNet and QuartzNet-10×5 training experiments, we set $\alpha = 3$ in Alg. 1, so skip connections are removed or shortened every three epochs. For QuartzNet-5×5, we set $\alpha = 1$ instead because it trains better this way. For the ResNets, we set $\mathscr{G}$ and $\mathscr{H}$ in Eq. 3.3 to *categorical cross entropy* and *mean-squared error*, respectively, and set $\beta = 0.35$. For the QuartzNets, we set Eq. 3.3's parameters similarly, except for $\mathscr{G}$, which we set to *connectionist temporal classification loss*, which is used to train difficult tasks involving sequence alignment (like DNA basecalling and ASR). Note that in our training results, "Baseline" refers to the unmodified NN counterpart with conventional skip connections.

Fig. 3.5 shows that SKIPREMOVER works well for ResNet-44 and smaller, at times



(a) CIFAR-10 **(b)** CIFAR-100 **(c)** SVHN

Figure 3.5. Top-1 accuracy of SKIPREMOVER and SKIPSHORTENER ResNets of increasing depth on various datasets. "Baseline" refers to an unmodified ResNet with conventional skip connections.



(a) Skip-less models **(b)** Shortened-skip models

Figure 3.6. Accuracy results for ResNets whose skip connections are all altered before training (apart from SKIPREMOVER and SKIPSHORTENER) on CIFAR-10. "From scratch" means training with randomly initialized weights without KD. "KD only" means training without dynamic skip alterations.

even outperforming its baseline (traditional skip connection model). However, its accuracy drops as the number of layers increases. This indicates that shallower NNs do not need skip connections for these classification tasks, but they become more necessary for deeper networks. SKIPSHORTENER mostly outperforms the baseline on all three datasets, even on deep models.

Ablation Studies

We also perform *ablation studies* in which we remove key parts of Tailor to understand why they are critical to minimizing accuracy loss. One key part of SKIPREMOVER and SKIPSHORTENER is the dynamic skip connection removal/shortening that occurs every few epochs

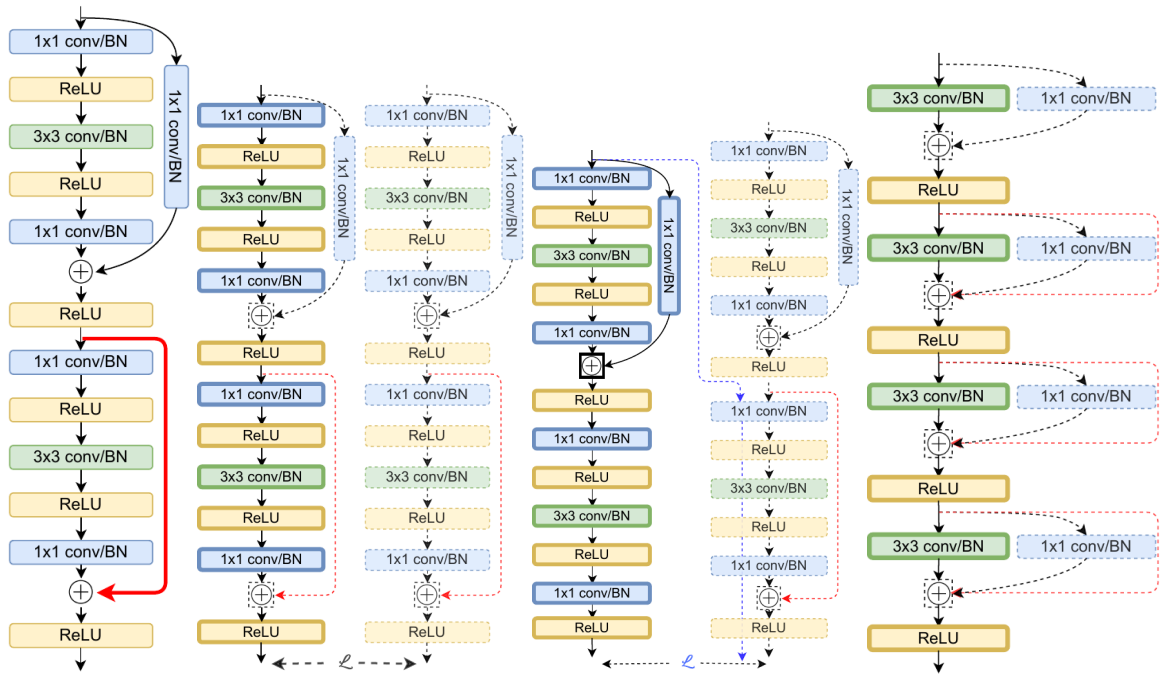
Table 3.1. Top-1 accuracy of ResNet-50 on the ImageNet dataset. *RD [114] only removes 82% of the skip connections.

Model	Accuracy (%)
ResNet-50	75.85
No skips (from scratch)	58.36
No skips (KD only)	69.40
Residual distillation (RD)* [114]	76.08
RepVGG-A2 [52]	76.48
SKIPREMOVER	**75.36**

during training under KD. We thus take away this dynamic model alteration by first altering the NNs to have either no skip connections or shortened skip connections. These pre-modified NNs are then trained under KD only. Another key part of SKIPREMOVER and SKIPSHORTENER is KD. We evaluate how skip-less and shortened-skip NNs perform without KD, training from randomly initialized weights (i.e., from scratch).

For ResNets trained on CIFAR-10, SKIPREMOVER and SKIPSHORTENER usually yield better results than either normal training or using KD-only on a statically pre-modified network on CIFAR-10 per Fig. 3.6a and Fig. 3.6b. The difference between all of the approaches in the figures is minimal for smaller models, but it becomes more apparent as NN depth increases. For instance, skip-less ResNet-110 under regular training yields an accuracy of 26.02% versus SKIPREMOVER, which achieves an accuracy of 90.68%, a 64.66% difference. SKIPREMOVER marginally outperforms regular training and KD-only on smaller skip-less models, but performs much better in comparison as the networks deepen. SKIPSHORTENER also generally performs better than the other two approaches for shortened skip models. Regular training mostly lags behind both KD and SKIPSHORTENER for shortened skip models.

For ResNet-50 on ImageNet, we only apply SKIPREMOVER because it uses an irregular skip connection architecture known as a "bottleneck block" to reduce the number of parameters [86]. This block has a skip connection spanning three layers: a 1×1 convolution, then a 3×3 convolution, then another 1×1 convolution (Fig. 3.7a). This irregular topology is not



(a) ResNet-50 block **(b)** Tailor **(c)** RD [114] **(d)** RepVGG [52]

Figure 3.7. Comparing Tailor's ResNet-50 skip removal method with residual distillation (RD) [114] and RepVGG [52]. The dashed portions are only used during training and are later removed, leaving the final inference NNs, indicated by bolder lines. Note that Tailor (b) removes skip connections from a pretrained ResNet-50 (a). RD does the same but uses a modified KD method that does not remove the 1×1 convolution addition (c). RepVGG starts training from a different NN topology altogether (d).

optimal for SKIPSHORTENER because it requires the majority of the shortened skip connections to pass through extra downsampling 1×1 convolutions to match the activation tensor shapes, significantly increasing the number of model parameters. As such, for ResNets with bottleneck blocks, like ResNet-50, we recommend SKIPREMOVER. As seen in Tab. 3.1, SKIPREMOVER incurs a 0.49% accuracy loss compared to the traditional ResNet-50. Compared to prior work such as RD [114] and RepVGG [52], SKIPREMOVER has slightly lower accuracy (at most 1.12% accuracy difference)[2].

Nevertheless, SKIPREMOVER has two advantages compared with these methods. First,

[2]Ding et al [52] introduce RepVGG models of varying depths. We compare against RepVGG-A2 because it is about the same size as ResNet-50.

Table 3.2. Top-1 accuracy of QuartzNet-5×5 on the Oxford Nanopore Reads dataset.

Model	Accuracy (%)
QuartzNet-5×5	95.107
No skips (from scratch)	94.475
No skips (KD only)	94.863
SKIPREMOVER	**95.086**
Shortened skips (from scratch)	95.019
Shortened skips (KD only)	95.016
SKIPSHORTENER	94.902

SKIPREMOVER removes all skip connections from ResNet-50, whereas RD only removes 82% of them. RD does not remove the 1×1 convolution addition used for downsampling (see Fig. 3.7c), which is particularly detrimental. In our experiments on hls4ml architectures, Vivado HLS estimates that ResNet-50's large 1×1 convolution skip connection consumes as many resources as the layers it skips over, effectively doubling resource consumption for that skip connection block. Although Vivado HLS has a tendency to overestimate the actual place-and-route (P&R) resource utilization, these estimates demonstrate that performing the 1×1 convolution is a nontrivial task that significantly affects resource consumption. Second, SKIPREMOVER removes the skip connections from an *existing* pre-trained model, whereas RepVGG requires developers to adopt a new model topology (see Fig. 3.7d). If developers do not already have a model on hand, RepVGG is a better option. However, if developers already have a ResNet trained for their specific dataset, it is advantageous to use SKIPREMOVER if they can afford a small accuracy loss. This prevents starting from scratch with RepVGG, which could require extensive hyperparameter tuning. Even finetuning a pre-trained RepVGG model to a new dataset using transfer learning is time consuming, as it is unclear which of the many methods [151, 198, 219] would work best. Instead, SKIPREMOVER allows developers to take advantage of their existing work and achieve a more resource-efficient model.

For QuartzNet-5×5, the SKIPREMOVER model performs the best—only 0.021% from the baseline (Tab. 3.2). These results all have high accuracy likely because DNA basecalling is

Table 3.3. Word error rate (WER) of QuartzNet-10×5 on LibriSpeech dataset. This includes clear ("dev-clean") and noisy ("dev-other") audio samples. "—" indicates the model failed to converge.

Model	dev-clean WER (%)	dev-other WER (%)
QuartzNet-10×5	5.56	16.63
No skips (from scratch)	—	—
No skips (KD only)	—	—
SKIPREMOVER	—	—
Shortened skips (from scratch)	**6.40**	**17.68**
Shortened skips (KD only)	7.14	19.95
SKIPSHORTENER	7.86	21.16

an easier sequence alignment task (only four classes) and the model is more than sufficient. For a harder ASR task like LibriSpeech, QuartzNet-10×5 fails to converge without skip connections. Since the model must translate audio samples to text, the audio samples can be noisy, making ASR harder. LibriSpeech, in fact, divides its test samples into "dev-clean" for clearly spoken samples and "dev-other" for noisy samples. With such a challenging task, it is not possible to remove the skip connections (like with DNA basecalling). Nonetheless, QuartzNet-10×5 performs well under SKIPSHORTENER, as it is within 2% of the baseline WER (Tab. 3.3). For both QuartzNet-5×5 and -10×5, the best performing shortened skip connection model was one whose skip connections were shortened first and then trained from scratch. While SKIPSHORTENER has minimal accuracy loss for both QuartzNets, we recommend training a model with shortened skip connections from scratch for this task.

Overall, SKIPREMOVER and SKIPSHORTENER perform better than either training on randomly initialized weights or training with KD only. For harder tasks like ASR though, training a shortened-skip model from scratch is a better choice. Nevertheless, the success of SKIPREMOVER and SKIPSHORTENER lies in augmenting KD with dynamic skip alterations.

3.4.2 Hardware Results

We first quantize ResNets ranging from 20 to 56 layers deep to see how Tailor's accuracy fares under reduced precision. We then evaluate Tailor's effects on hardware resources and latency by performing a case study on ResNet-20-style skip connections implemented using the hls4ml architecture, i.e., the designs illustrated in Fig. 3.2. We select this style of skip connection because it is the fundamental building block of ResNets that range from 20 to 110 layers. In our case study, we vary the bit precision and number of filters to see how Tailor scales up. Based on how Tailor's resource reductions scale, designers can understand how Tailor extrapolates to their own hardware designs. We report latency as well as P&R resource results on the Alveo U200 FPGA accelerator card (part no. `xcu200-fsgd2104-2-e`). For end-to-end application results, we evaluate the benefits of Tailor on two different styles of CNN architectures. The first uses the hls4ml tool to generate architectures. The second is the Reconfigurable DNN Engine—a 2D array of processing elements. Both styles of architectures are described in Sec. 3.3.1.

Quantization

The parameters of a hardware-accelerated NN are typically quantized from floating-point to fixed-point precision [34, 137, 205].

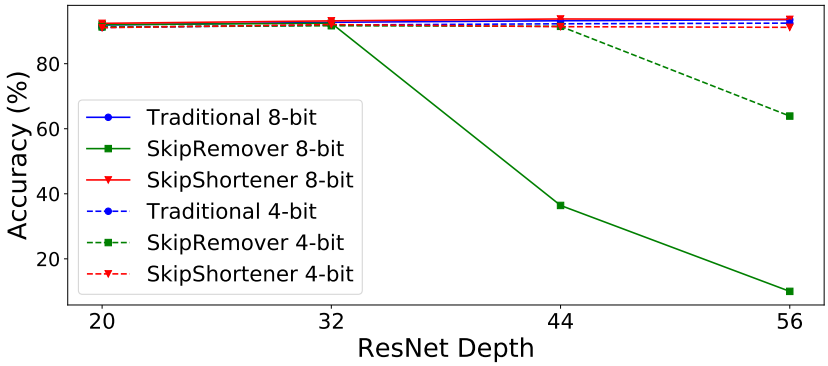


Figure 3.8. Quantized accuracy results for 8-bit and 4-bit fixed point using Brevitas.

Quantizing deep NNs with minimal accuracy loss is a largely manual and time-consuming task [76]. We use Brevitas [153] to quantize our SKIPREMOVER and SKIPSHORTENER ResNets

with depths of 20 to 56 from 32-bit floating-point (float32) to 8-bit and 4-bit fixed-point precision on the CIFAR-10 dataset. We modified Tailor's hardware-aware training algorithm where the teacher continues to use floating-point representation whereas the student is quantized. This results in the student undergoing quantization-aware training. In Fig. 3.8, we find that SKIPSHORTENER ResNets consistently outperform traditional ResNets under Brevitas quantization-aware training by 0.5%. SKIPREMOVER ResNets start to suffer from the lack of bits as they get deeper, with accuracy dropping to random classification for ResNet-56. But, Brevitas is only one of dozens of ways to quantize neural networks [56, 76, 137, 196], so it may be the case that a SKIPREMOVER ResNet-56 requires a different method of quantization to achieve a quantized accuracy similar to its float32 counterpart.

FPGA Evaluation

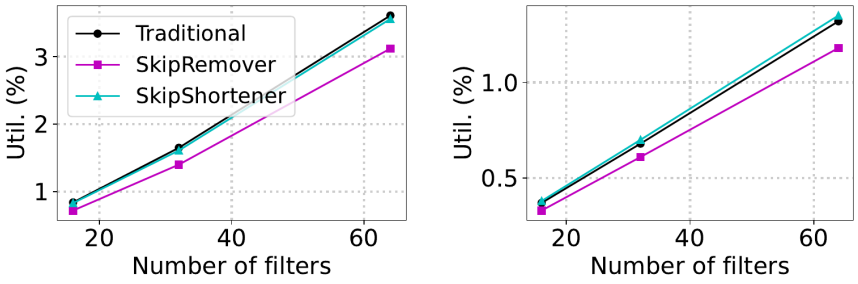
Our first study looks solely at one ResNet block. The second study performs an end-to-end implementation of ResNet8 and ResNet50.

For our case study on a ResNet skip connection blocks (see designs in Fig. 3.2), we evaluate Tailor at `ap_fixed<8,3>` and `ap_fixed<16,6>` precisions using the hls4ml architecture. Under both bitwidths, we increase the number of filters for all designs from 16 to 32 to 64. This way, we can understand how Tailor scales with the number of filters. We use hls4ml [63] to translate these hardware designs into Vivado HLS, targeting the Alveo U200 FPGA accelerator card. hls4ml uses task-level pipelining (i.e., HLS dataflow) for each NN layer, or small group of layers and streams data between dataflow stages using FIFOs. hls4ml also exposes a knob known as *reuse factor*, which determines how often multipliers are reused in a design. To fairly compare our designs as the number of filters increases, we fix the reuse factor to 576. We then synthesize our designs to report P&R resource utilization as well as co-simulation latency results. Lastly, we run the designs on the U200 to verify correctness.

Under 8-bit precision, we find that both SKIPREMOVER and SKIPSHORTENER reduce resources. Tab. 3.4 summarizes our P&R results. Since our model uses 8-bit precision, we

Table 3.4. Place-and-route resource utilization of a skip connection block as the number of filters increases for $\langle 8,3 \rangle$ precision on an Alveo U200. SKIPREMOVER reduces LUT and FF usage, whereas SKIPSHORTENER trades an increase in FFs for a decrease in LUTs. T = Traditional, R = SKIPREMOVER, S = SKIPSHORTENER.

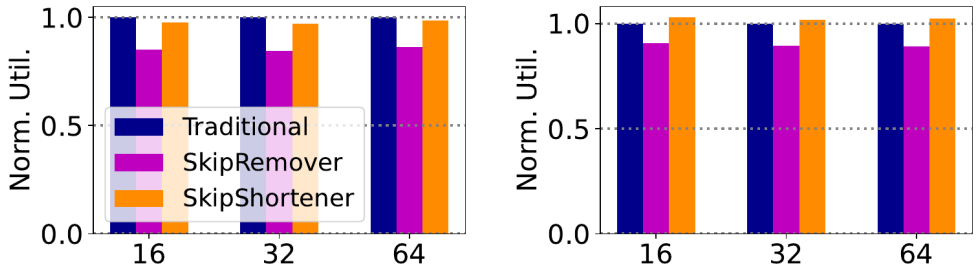
# filters	LUT			FF			DSP	BRAM
	T	R	S	T	R	S	T/R/S	T/R/S
16	9,984	8,482	9,764	8,654	7,841	8,916	0	18.5
32	19,566	16,512	18,993	16,183	14,506	16,489	0	36.5
64	42,688	36,882	42,121	31,124	27,815	31,850	0	82



(a) LUT **(b)** FF

Figure 3.9. Percent resource utilization of a $\langle 8,3 \rangle$ skip connection block at various filter sizes on an Alveo U200. DSPs and BRAMs remain the same across the three designs, so they are not shown. SKIPREMOVER and SKIPSHORTENER LUT and FF reductions scale linearly, as expected.

see that all of our models exhibit low DSP usage and higher LUT and FF utilization. This is because Vivado HLS maps multiplications on datatypes that are less than 10 bits to LUTs instead of DSPs, as noted by [3, 205]. It is possible to pack two 8-bit weights into a DSP [68], but this is out of scope and orthogonal to the effects Tailor has on hardware. Furthermore, all of the traditional and Tailor designs use the same amount of BRAMs with respect to the number of filters because here the BRAMs are used solely for on-chip weight storage, which does not differ across design. Nonetheless, SKIPREMOVER decreases LUT usage by up to 16% and FF usage by up to 11% compared with the traditional design (Fig. 3.10). These resource savings represent the extra hardware needed to implement a skip connection and subsequently the resources saved. As previously mentioned in Sec. 3.3.1, the extra dataflow stages that carry



(a) LUT **(b)** FF

Figure 3.10. Resource utilization normalized to the traditional design of a $\langle 8,3 \rangle$ skip connection block at various filter sizes. DSPs and BRAMs remain the same across the three designs, so they are not shown. SKIPREMOVER and SKIPSHORTENER LUT and FF reductions scale proportionally, as expected.

out a skip connection are no longer necessary. More importantly, SKIPREMOVER's savings scale linearly as the number of filters increases from 16 to 64 (Fig. 3.9). SKIPSHORTENER's resource reductions present a tradeoff, increasing FFs by 2% in exchange for decreasing LUTs by 3% (Fig. 3.10). SKIPSHORTENER lowers LUT utilization because the lifespan of each skip connection lasts only one dataflow stage instead of the traditional two. This means we need not spend extra logic on the dataflow stages needed to copy the skip connections to buffers that last longer than one stage. However, since the shortened skip connection now fully resides in a single dataflow stage (previously described in Fig. 3.2c), this requires some extra FFs. This represents the tradeoff SKIPSHORTENER provides at 8-bit precision: some extra FFs for fewer LUTs. These resource tradeoffs also scale linearly as the number of filters scales up, as seen in Fig. 3.9.

We find more dramatic resource reductions when we look at our 16-bit designs. Tab. 3.5 summarizes our P&R results. In contrast with our 8-bit designs, at higher precision, our designs rely more on DSPs and BRAMs. This time the BRAMs are used not only to store weights on chip but also to implement the FIFOs that connect the dataflow stages. Therefore, as we tailor the dataflow stages according to each design (e.g., SKIPREMOVER or SKIPSHORTENER), the BRAMs now also reflect these changes. At its best, SKIPREMOVER lowers LUTs by 11%, FFs

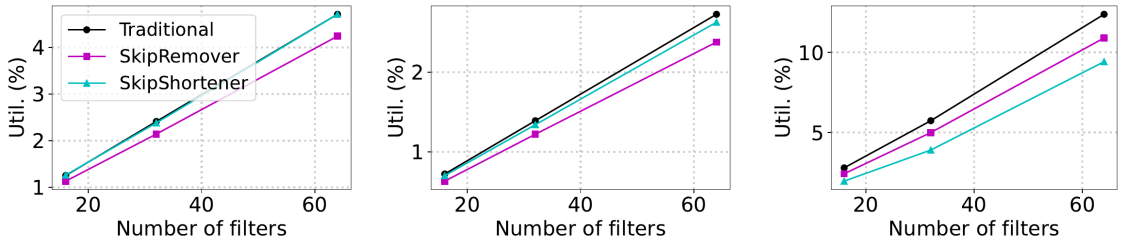
by 13%, and BRAMs by 13%. Without a skip connection to implement, SKIPREMOVER uses fewer resources than the traditional design. The DSPs remains unchanged because they are used solely for the convolutional layers' multiplications and not the skip connection, which is also the case for SKIPSHORTENER.

Similar to the 8-bit designs, SKIPSHORTENER presents a resource tradeoff—this time trading a small increase in LUTs (at most 1%) for decreases in FFs and BRAMs. In the best case, SKIPSHORTENER reduces LUTs by 1%, FFs by 4%, and BRAMs by 34%. While SKIPSHORTENER uses fewer LUTs than the traditional case for 32 filters, SKIPSHORTENER pays about a 1% increase in LUTs for 16 and 64 filters in exchange for decreases in FFs and BRAMs. This small disparity is likely an artifact of the heuristics Vivado P&R uses to allocate resources. Again, these resource tradeoffs and savings are possible because the shortened skip connections can be implemented within a single dataflow stage due to its reduced lifetime. Tab. 3.6 shows that the lifetime of each shortened skip connection is a little less than half the lifetime of the traditional one. With shorter lifetimes, we find that the SKIPSHORTENER's skip connections' FIFOs can now be implemented using shift registers instead of BRAMs, which is what the traditional design still uses (Tab. 3.6). Shift registers are much more efficient memories compared to BRAMs. As such, it is advantageous to hardware designers to consider how SKIPSHORTENER provides opportunity to implement skip connections with a more efficient memory architecture like shift registers. This leads to 30–34% fewer BRAMs than the traditional design, even as the number of filters scales up. While in this case SKIPSHORTENER uses fewer BRAMs than SKIPREMOVER does, SKIPSHORTENER offsets this difference by using more FFs than SKIPREMOVER does. For both SKIPREMOVER and SKIPSHORTENER, resource utilization (and the associated reductions) scale linearly, as seen in Fig. 3.11.

Tailor does not affect latency for hls4ml architectures. As seen in Tab. 3.7, for each number of filters, all designs exhibit the same latency, according to co-simulation on an Alveo U200. The slight decrease in latency as the number of filters scales is due to an increase in DSPs and a higher degree of parallelism. As discussed in Sec. 3.3.1, hls4ml designs pipeline their tasks.

Table 3.5. Place-and-route resource utilization of a skip connection block as the number of filters increases for $\langle 16, 6 \rangle$ precision on an Alveo U200. SKIPREMOVER reduces resources across the board, whereas SKIPSHORTENER trades an increase in LUTs for a decrease in FFs and BRAMs. T = Traditional, R = SKIPREMOVER, S = SKIPSHORTENER.

# filters	LUT			FF			DSP	BRAM		
	T	R	S	T	R	S	T/R/S	T	R	S
16	14,733	13,320	14,933	17,044	14,935	16,438	12	60.5	52.5	42.5
32	28,498	25,330	28,184	32,923	28,747	31,764	48	124	108	84.5
64	55,699	50,074	55,720	64,564	56,263	62,252	192	267.5	235.5	203.5



(a) LUT **(b)** FF **(c)** BRAM

Figure 3.11. Percent resource utilization of a $\langle 16, 6 \rangle$ skip connection block at various filter sizes on an Alveo U200. SKIPREMOVER and SKIPSHORTENER resource reductions scale linearly, as expected.

The convolutions' multiplication tasks dominate the overall dataflow latency. The tasks that SKIPREMOVER eliminates and SKIPSHORTENER implements more efficiently, namely the skip connection cloning and addition stages, have significantly lower latency than the convolutions and are thus not on the critical path. The throughput thus remains the same.

By shortening skip connections, we reduce their lifespans, which provides an opportunity for simplifying their hardware implementation specifically for hls4ml architectures. However, shortening skip connections is not beneficial for all architectures. As seen in Tab. 3.8, shortening skip connections is worse for both GPU and CPU because doing so increases off-chip memory accesses. These extra accesses lower throughput by 5% on GPU and 2% on CPU. On FPGAs with hls4ml architectures, however, we can modify the architecture to take advantage of shortened skip connections, reducing resource consumption without negatively affecting throughput (Tab. 3.8).

We performed two studies to understand how Tailor performs for end-to-end imple-

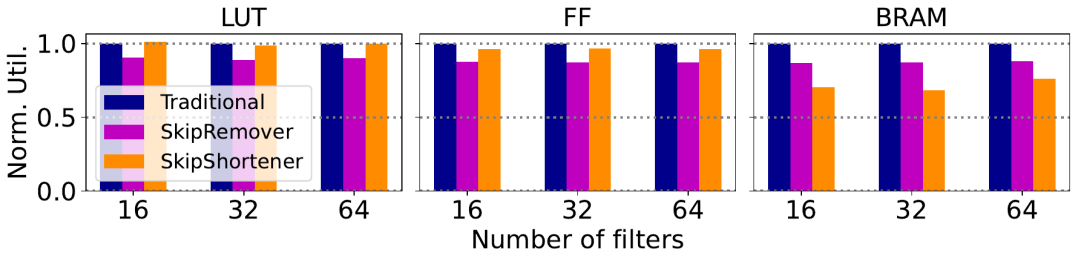


Figure 3.12. Resource utilization normalized to the traditional design of a $\langle 16, 6 \rangle$ skip connection block at various filter sizes. The SKIPREMOVER and SKIPSHORTENER resource savings scale proportionally as the number of filters scales up.

Table 3.6. FIFO depths of a single skip connection hardware design at 16-bit precision. SKIPREMOVER has no skip connections, so it has no skip connection FIFOs.

Hardware Design	**FIFO Depth**	**FIFO Implementation**
Traditional	69	BRAM
SKIPREMOVER	0	—
SKIPSHORTENER 1st skip	33	Shift Register
SKIPSHORTENER 2nd skip	34	Shift Register

mentations of ResNet models. The first is ResNet8 from MLPerf Tiny that was designed in hls4ml [18, 27]. The second is ResNet50 implemented on the Reconfigurable DNN architecture.

The ResNet8 model targets the Alveo U200. It uses 16-bit fixed-point representation with six integer bits. The reuse factor for the layers was hand-tuned to 72, which directly affects the resource usage and latency of the layers. The reuse factor is one of the more important knobs for design space exploration in hls4ml and is often hand-tuned to maximize resource usage of the platform while optimizing the overall network performance.

Tab. 3.9 shows the resource usage results for the ResNet8 model with skip connections, with shortened skip connections, and without skip connections. Removing the skip connections has clear benefits across all the resources. Shortening the skip connections reduces BRAMs while increasing LUTs and FFs. Both the shortened skip connections and the removed skip connections models show improved accuracy over traditional skip connections. In all cases, the latency remains the same, requiring 304,697 cycles running at 100 MHz (approximately

Table 3.7. Latency co-simulation results of a skip connection block at $\langle 8,3 \rangle$ and $\langle 16,6 \rangle$ precision. The latency for the Traditional, SKIPREMOVER, and SKIPSHORTENER designs are the same for each number of filters because they all rely on task-level pipelining that reuses multipliers at the same rate (576×).

# filters	Latency (ms) Traditional/SKIPREMOVER/SKIPSHORTENER
16	23.38
32	23.05
64	22.39

Table 3.8. Normalized throughput of a ResNet20. The GPU and CPU both were run with batch size = 64, whereas FPGA was run with batch size = 1. Throughput is normalized column-wise to the top entry. GPU = 1080Ti. CPU = AMD Ryzen 9 5900X. FPGA = Alveo U200. SKIPREMOVER increases GPU and CPU throughput because it decreases off-chip memory accesses. SKIPSHORTENER, however, decreases GPU and CPU throughput because it increases off-chip memory accesses. For a fully on-chip, dataflowed FPGA architecture, neither SKIPREMOVER nor SKIPSHORTENER have any effect on throughput.

Model	GPU	CPU	FPGA
Traditional skip connections	1×	1×	1×
SKIPREMOVER	1.11×	1.03×	1×
SKIPSHORTENER	0.95×	0.98×	1×

3ms/inference).

Our second full model case study implemented a Reconfigurable DNN architecture on the ZCU102 development board which contains a Zynq UltraScale+ MPSoC. The Reconfigurable DNN array is configured to have 7 rows × 96 columns for a total of 672 of processing elements (PEs) that support 8-bit inputs and 8-bit weights. Each PE contains a multiplier and an accumulator implemented using DSPs on FPGA fabric. Input pixels and weights are streamed into the engine as AXI-Stream packets. Images are processed in batches of 7, to increase the reuse and reduce memory accesses. The Reconfigurable DNN architecture was synthesized, placed & routed at a clock frequency of 250 MHz on a ZCU102. The architecture with $7 \times 96 = 672$ PEs used 49057 LUTs (18%), 81446 flip flops (15%), 114 BRAMs (13%), and 1344 DSPs (53%) on the FPGA fabric.

Table 3.9. MLPerf Tiny ResNet8 model implemented using hls4ml with skip connection, with shortened skip connections, and without skip connections.

	With Skip Connections	Shortened Skip Connections	Without Skip Connections
Accuracy (%)	87.39	87.93	87.62
LUTs	158609	165699	144206
FFs	196012	204914	181768
DSP48s	1083	1083	1043
BRAMs	173	158.5	156

We implemented a ResNet50 model with and without skip connections on a 672-element Reconfigurable DNN architecture running on the ZCU102. Tab. 3.10 shows the performance of ResNet50. Removing the skip connections largely benefits the performance due to the removal of the 1×1 convolution blocks. Removing the skip connections also removes those layers, which no longer need to be scheduled on the PE array. The results are much better performance in terms of all metrics: approximately 30% increases in FPS and latency and approximately 45% decrease in memory accesses.

Table 3.10. ResNet50 performance with and without skip connections on the Reconfigurable DNN architecture. The architecture has 672 processing elements and runs on the ZCU102 development board at 250 MHz.

	With skip connections	Without skip connections
Accuracy (%)	75.85	75.36
Frames per second (FPS)	28.69	37.47
Time per image (s)	0.035	0.027
Latency (s)	0.244	0.187
Memory access per image (Mb)	140.95	92.71

3.5 Discussion

With these results in hand, designers can now consider which accuracy versus resource tradeoffs they are willing to make during the hardware-software codesign process.

SKIPREMOVER provides minimal accuracy loss while reducing resource consumption and increasing performance—a win-win scenario. As seen in Sec. 3.4.1, SKIPREMOVER ResNet-

50 is only 0.49% less accurate than the baseline on ImageNet. But, SKIPREMOVER is less effective on deeper NNs (such as QuartzNet-10×5 and ResNet-110). In fact, QuartzNet-10×5 fails to converge when trained under SKIPREMOVER. For such deep NNs trained on difficult tasks like ASR, skip connections are instrumental in training convergence [86]. By removing skip connections, we expect and see a degradation in accuracy for deeper NNs. This degradation is not as drastic for other tasks. For instance, ResNet-110 still converges when trained using SKIPREMOVER, but it is 3.72% less accurate on CIFAR-10 and 9.61% less accurate on CIFAR-100, compared to the original baseline model. We propose this tradeoff between NN size and SKIPREMOVER performance as an additional consideration during design space exploration. In response, SKIPSHORTENER is more suitable for deeper NNs when SKIPREMOVER is less effective. SKIPSHORTENER maintains accuracy comparable to its original skip connection models and reduces resource requirements by up to 34% compared to the traditional skip connection model.

Based on our hls4ml evaluation, designers can extrapolate to their own designs because, as we have shown in Fig. 3.9 and Fig. 3.11, the resource usage and savings scale linearly as the number of filters grows. We have also shown that at the higher 16-bit precision, Tailor provides significant resource reductions, so if designers need more precision, Tailor's savings will follow. If they need lower 8-bit precision, SKIPREMOVER still manages to lower the 8-bit designs' LUTs by 16% and FFs by 11%. Even SKIPSHORTENER decreases LUTs by 3% despite a 2% increase in FFs, though these smaller resource savings are offset by its overall higher accuracy performance compared with SKIPREMOVER. As a result, it is up to the designer to consider how to best apply Tailor's codesign methods given their accuracy and resource requirements.

3.5.1 Theoretical Understanding

Prior work investigated why skip connections are so helpful to ResNets. Veit et al [192] argue that ResNets behave like ensembles of smaller subnetworks that vary in depth and allow the NN to train and converge more easily. Li et al [115] and Yao et al [208] show that introducing

skip connections make the NN loss landscape to be much smoother and have less nonconvexity. They show that naively removing these skip connections causes an explosion of nonconvexity in the loss landscape, which makes training significantly more difficult. We confirm these results in our ablation studies (Sec. 3.4.1), as accuracy indeed drops when skip connections are removed naively. With both KD and SkipRemover, we see an improvement in accuracy. Since the student is trying to mimic the teacher's outputs, it is possible that the teacher's outputs guide the student in such a way that prevents the loss landscape from becoming less smooth. Theoretical work from Lin et al [118] has proven that a ResNet with one-neuron hidden layers is a universal approximator. This work suggests that adding more neurons to the hidden layers creates an over-parameterized ResNet. Since stochastic gradient descent performs better in the presence of over-parameterization, having more neurons per hidden layer increases training efficiency, making it easier to converge. This work also argues that a ResNet is essentially a sparse version of a fully connected NN because the identity skip connections create simpler paths within the ResNet, which was similarly posited by Lin et al [118]. Given that CNNs and ResNets have both been proven to be universal approximators [118, 155], this implies that there exists a set of parameters for a CNN that can mimic a ResNet such that they equal the same function. It is mainly easier to find a well performing ResNet because Lin et al [118] showed that one-neuron hidden layers is sufficient for a ResNet to be a universal approximator.

3.5.2 Future Work

In our work, Tailor has taken removing and shortening skip connections to their extremes: it either fully removes or fully shortens all the skip connections in a NN. It would be worthwhile to understand the accuracy versus resource utilization tradeoff under less extreme cases, e.g., removing only half of the skip connections. It would also be interesting to mix SKIPREMOVER and SKIPSHORTENER together to try and recover accuracy in the instances when SKIPREMOVER fails. These approaches may help address SKIPREMOVER's scalability issues and strike a balance between SKIPSHORTENER's high accuracy and SKIPREMOVER's resource savings and

performance improvements.

3.6 Summary

Tailor introduces two new methods, SKIPREMOVER and SKIPSHORTENER, that alters NNs with skip connections dynamically during retraining to fit better on hardware, achieving resource-efficient inference with minimal to no loss in accuracy. With SKIPREMOVER, NNs no longer need to rely on skip connections for high accuracy during inference. With SKIPSHORTENER, we retrain NNs to use shorter skip connections with minimal to no loss in accuracy. Shortening skip connections is beneficial for hardware architectures generated by the hls4ml tool as it reduces the skip connection lifetime. We demonstrate FPGA resource consumption reductions of up to 34% for BRAMs, 13% for FFs, and 16% for LUTs. We show that Tailor is also valuable for optimizing 2D PE array architectures. SKIPREMOVER increases performance by 30% and decreases memory bandwidth by 45%. Designers can decide which accuracy versus resource tradeoffs offered by SKIPREMOVER and SKIPSHORTENER are suitable to their design requirements. As a result, Tailor is another tool in the hardware-software codesign toolbox for designers to use when building customized accelerators.

Chapter 3, in part, is a reprint of the material as it appears in ACM Trans. Reconfigurable Technol. Syst. 17, 1, Article 11 (March 2024). The work in this chapter was done in collaboration with Gabriel Marcano, Vladimir Loncar, Alireza Khodamoradi, Abarajithan G, Nojan Sheybani, Andres Meza, Farinaz Koushanfar, Kristof Denolf, Javier Mauricio Duarte, and Ryan Kastner. The dissertation author was the primary investigator and author of this work.

Chapter 4

Greater than the Sum of its LUTs: Scaling Up LUT-based Neural Networks with AmigoLUT

In this chapter, I present AmigoLUT, which introduces the idea of using ensemble learning on LUT NNs to scale up accuracy without exponentially increasing FPGA resources. The goal of ensemble learning is for "the whole to be greater than the sum of its parts," suggesting that when the individual models work together as a whole, it is better than any one individual model. In this case, I want the ensemble of LUT NNs to be greater than "the sum of its LUTs." I study several methods for training LUT NNs to work together as an ensemble. Curiously, the best method combines the models of the ensemble by summing the outputs of the LUT NNs to achieve resource-efficient inference.

4.1 Introduction

Machine learning has become increasingly prevalent in areas such as high-energy physics, cybersecurity, and autonomous vehicles [61, 81, 125, 190, 202]. Many of these domains require extremely high throughput and low latency. For example, at the CERN Large Hadron Collider (LHC), the Compact Muon Solenoid experiment [182] runs particle collision experiments that generate data rates of $\sim$40 TB/s, which physicists need to compress in real time ($\leq$25 ns) to lower the data bandwidth to a level that their system can handle [61, 81].

Recently, lookup-table-based neural networks (NNs) such as LogicNets [190] and NeuraLUT [11] have sought to meet this demand for ultra low-latency, high-throughput NNs. For instance, LogicNets achieve efficient computation by carefully mapping neurons to lookup tables (LUTs), a fundamental building block of field-programmable gate arrays (FPGAs). This makes LUT-based NNs efficient, yielding high throughput and low latency implementations while sacrificing some accuracy compared with dense NNs.

However, it is challenging to create more accurate LUT-based NNs because LUT usage increases exponentially ($\Omega(2^n)$) with respect to neuron fan-in (i.e., n = number of synapses $\times$ synapse bitwidth). As such, LUT-based NNs must be heavily quantized and very sparse to efficiently map to FPGA LUTs, which are 6 inputs or less. LUT-based NNs have a fundamental tradeoff between input size and LUT size because LUT size increases exponentially with input size. The standard method of improving NN accuracy [29, 148] by increasing NN depth, width, and neuron fan-in fails for LUT-based NNs because they quickly run out of the available LUTs on an FPGA due to exponential scaling. As a result, scaling up LogicNets by increasing NN size and density presents a poor tradeoff because we get minimal accuracy increase at the cost of exponential increases in LUT usage as well as decreases in throughput.

Previously, researchers have increased model accuracy through ensemble learning (*ensembling*) [54, 84, 165, 206, 217]. Ensembling increases model accuracy by having multiple, weaker NNs work together to make a decision. The idea behind ensembling is that the whole is greater than the sum of its parts; by having multiple NNs work together, we can achieve better performance than any one model individually. Prior work [28, 67] often forms ensembles of many small and weak models to promote diversity in the decision each model makes (i.e., they must disagree with each other to an extent [183]). Otherwise, if all the models always agree with each other, then there is no benefit to combining them together. Through this "diversity of thought," the ensemble can make a better decision as a group. Since LUT-based NNs are constrained by neuron fan-in, they must be extremely sparse and quantized to avoid an explosion in LUT usage, sacrificing some accuracy and making them relatively weak NNs. As a result,

LUT-based NNs are good candidates for ensembling.

Our work AmigoLUT[1] creates ensembles of smaller LUT-based NNs to improve accuracy, scaling up performance *linearly* with respect to the number of models rather than *exponentially* with respect to neuron fan-in. Although AmigoLUT does not achieve higher accuracy or lower LUT usage than recent state-of-the-art work [17, 119], our results show that, for certain lower accuracy benchmarks, AmigoLUT increases throughput and reduces LUT usage by over an order of magnitude. Thus, the goal of this chapter is to demonstrate the great potential of scaling up LUT-based NNs with ensembling.

How to apply ensembling to LUT-based NNs effectively presents many challenges: (1) which ensembling method should we use for LUT-based NNs? (2) how can we analyze their ensemble performance? (3) how can we implement these ensembles on FPGAs efficiently? Although many ensembling methods have been introduced over the years [206, 217], it is not immediately clear which one would work best on LUT-based NNs and why. Few ensembling studies have been completed on NNs that are both extremely sparse and quantized let alone LUT-based NNs [176, 203, 218]. Moreover, how to map LUT-based NN ensembles to FPGAs while minimizing the resource overhead needed to combine the ensemble members together is non-trivial.

In this chapter, we tackle these challenges by evaluating three fundamental ensemble learning methods: averaging [217], bagging [28], and AdaBoost [67]. We study how LogicNets [190], PolyLUT [10] and NeuraLUT [11] ensemble together on two high-energy physics datasets [61] and the MNIST image classification dataset [109]. From our results, we find that averaging performs the best. We then analyze our ensemble learning results using diversity and disagreement metrics. We introduce a new diversity plot for visualizing ensemble diversity to understand why averaging is the best. Finally, we introduce an open-source tool AmigoLUT[2], a hardware-friendly ensemble learning framework for LogicNets, PolyLUTs, and NeuraLUTs that

[1] "Amigo" means "friend" in Spanish.

[2] https://github.com/KastnerRG/amigolut

minimizes ensemble resource overhead when implementing ensembles on FPGAs.

Our primary contributions are: (1) evaluating various ensemble learning methods on LUT-based NNs such as LogicNets, PolyLUT, and NeuraLUT (2) analyzing ensembles using novel diversity plots and disagreement metrics to better understand their performance, and (3) introducing AmigoLUT, an ensemble learning framework that efficiently maps ensembles of LUT-based NNs to FPGAs.

4.2 Background & Related Work

This section reviews relevant background on ensembling and related work on LUT-based NNs.

4.2.1 Ensembling

Ensembling has been studied in the fields of statistics and machine learning for decades, and it has shown notable improvements in performance [28, 67, 84, 217]. While there are many ensembling methods to choose from, we start with three seminal ensembling methods: averaging [217], bagging [28], and AdaBoost [67]. The goal of each method is to promote diversity among the ensemble members so that together they can make a decision better than any one model could individually.

Averaging involves taking the average of the outputs of all ensemble members. Averaging introduces diversity by randomly initializing each ensemble member's weights differently, training them independently or dependently. In our work, we randomly initialize each model's weights differently and backpropagate the averaged outputs of the ensemble through each model, training the models dependently together as one.

Bagging, an abbreviation of bootstrap aggregating, involves randomly sampling the training set with replacement to create a unique training set per ensemble member, which is then each trained independently. These unique training sets introduce diversity into the ensemble because each ensemble member trains on different distributions of the data. The ensemble

members' outputs can be combined in many ways, but in our work, we average the outputs.

AdaBoost is a subclass of *boosting* [217], which is an ensembling method that trains each ensemble member sequentially, where each model's performance informs how the subsequent model will be trained. AdaBoost, or adaptive boosting, achieves this by assigning weights to each training sample, starting with equal weights. AdaBoost uses these weights to randomly sample from the training set. As each model trains, the weights of the training samples are adjusted based on how the model performs on them. The samples with poor performance are assigned higher weight so that they are sampled more frequently when the next model trains. This way, each subsequent model targets training samples that the previous models performed poorly on, in an effort to "boost" each other's performance. To combine the ensemble's outputs, AdaBoost weights each model based on its performance, computing a weighted average of each model's outputs to produce the final output.

Bagging and AdaBoost were first studied on decision trees, which are relatively weak, increasing accuracy remarkably [28, 67]. Prior work [217] has attributed this increase in accuracy to the combination of weak models along with their unique methods of promoting diversity in training. Based on this observation, we hypothesize that sparse, quantized LUT-based NNs should ensemble together well because they are weaker than their dense, full-precision counterparts. Few works [65, 176] have evaluated ensembling NNs that are both sparse and quantized. We evaluate our hypothesis on averaging, bagging, and AdaBoost in Sec. 4.3.

4.2.2 LUT-based NNs

Many kinds of LUT-based NNs have been introduced in the literature.

LUTNet [195] uses LUTs as the fundamental inference operator to implement NNs on FPGAs. LUTNet seeks to improve binary NNs' reliance on XNOR operations by replacing them with more expressive, learnable K-LUT Boolean operations. While LUTNet improves NN resource efficiency on FPGAs, the number of parameters in LUTNet scales exponentially with respect to the number of LUT inputs, making it difficult to improve accuracy by increasing

model size through parameter count.

Weightless NNs (WNNs) are another class of LUT-based NNs [8, 133, 177, 178]. WNNs consist of neurons, but they do not use weights to determine how they respond (i.e., they are weightless). WNN neurons are made up of logical LUTs that represent Boolean functions. WNN neurons concatenate their inputs, which form an address they use to perform a lookup in the associated LUT. WNNs learn by starting with an N-input, 2^N-output LUT that contain 1-bit entries initialized to all zeroes [177]. Then for each training input that has a value of one, it flips the associated LUT entry to one. These logical LUTs can then be mapped to FPGA physical LUTs via a logic synthesis tool. WNNs suffer from exponential scaling with respect to the number of LUT inputs as well as difficulty in generalizing to unseen input data, as they primarily memorize patterns [177]. Recent work, such as differentiable WNNs (DWNs) [17] improve WNN generalization by making them differentiable so that they learn connections between LUTs rather than using fixed random setups.

Ensembling WNNs has been done before [65, 176]. ULEEN [176] ensembles WNNs by training many and selecting a few of the best performing models to train together, summing their outputs in the end to make decisions. Filho et al. [65] evaluate bagging and boosting on their WNNs; however, they show minimal accuracy gains, quickly hitting diminishing returns as ensemble size increases.

LogicNets [190] and NullaNet [143, 144] fully encapsulate a neuron's operation within a logical LUT. Given a trained quantized NN, LogicNets feed all possible quantized inputs to a neuron and captures the quantized outputs it computes, while NullaNet does so in a lossy manner. These inputs and outputs are then enumerated in a logical LUT and then implemented as physical LUTs by a logic synthesis tool.

PolyLUT [10] and NeuraLUT [11] are successors of LogicNets. Each method improves the accuracy of LogicNets by attempting to capture more complex functions within the logical LUT. PolyLUT learns a NN that is a piece-wise polynomial function whereas NeuraLUT encapsulates an arbitrary NN within a logical LUT. This improves the scalability of LogicNets

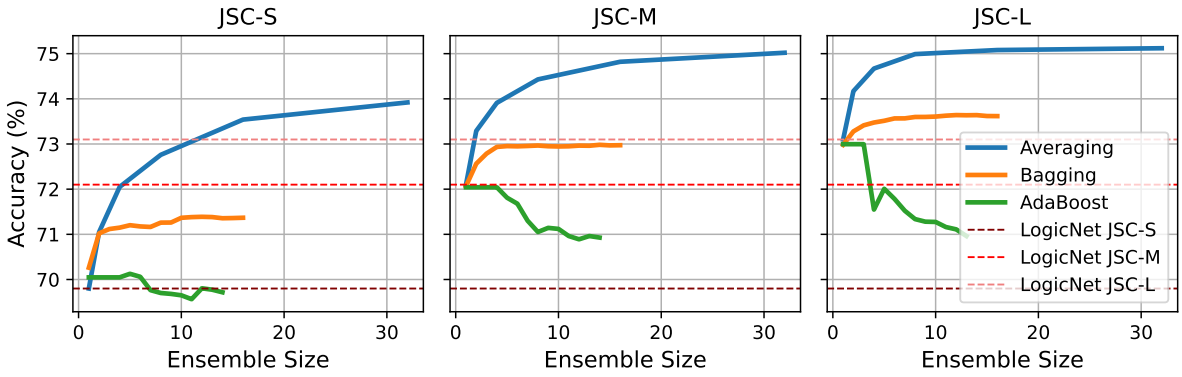


Figure 4.1. JSC Ensembling Comparison. Averaging is the best ensembling method for the JSC task, whereas bagging quickly hits diminishing returns and AdaBoost decreases performance overall.

because fewer logical LUTs are needed to achieve higher accuracy, leading to fewer physical LUTs and lower latency. PolyLUT-Add [119] sums smaller PolyLUT logical LUTs together to build larger, more complex LUTs. However, similar to LogicNets, all three methods still scale exponentially with respect to LUT inputs. We evaluate AmigoLUT on LogicNets, PolyLUT, and NeuraLUT, creating ensembles with them as base models. Although each base model still suffers from exponential scaling with respect to LUT inputs, AmigoLUT alleviates this by relying on weaker models that have smaller LUT fan-in. As a result, AmigoLUT scales performance *linearly* with respect to the number of models.

4.3 Ensembling LUT-based NNs

The first step in ensembling LUT-based NNs is determining which ensembling method increases accuracy the most. We study three fundamental ensembling methods: (1) averaging, (2), bagging, and (3) AdaBoost. We seek to answer: (1) which ensembling method performs the best, and (2) why does one ensembling method outperform another?

4.3.1 Ensembling results

To determine which of our three ensembling methods is the best, we evaluate LogicNets models on three datasets: MNIST [109], high-granularity endcap calorimeter (HGCal) compression [61], and jet substructure classification (JSC) [61].

MNIST is a well-known dataset tasked with classifying handwritten digits. The HGCal and JSC datasets are latency-critical tasks that come from the LHC's Compact Muon Solenoid experiment [182]. At the LHC, physicists run particle collision experiments that generate data rates of $\sim$40 TB/s. To reduce data rates, the physicists plan to deploy tens of thousands NN encoders in the LHC to compress particle collision data from the HGCal [47] sensor instrument into a smaller format. This encoded data is then passed to a decoder off-sensor for further processing in the trigger system. We evaluate HGCal model performance using Earth mover's distance (EMD), a distance measure between two probability distributions [160]. In our case, the EMD measures the distance between the encoder's input energy readings and the decoder's outputs, respectively. Lower EMD is better, and an EMD of 0 indicates the autoencoder is lossless. The trigger system, to which the decoder outputs data, performs many tasks, one of which is JSC. JSC involves classifying particle collision data into five types of jets, which informs physicists about potentially new physics. Of note, the HGCal encoder hardware must accept new input data at 40 MHz and complete inference in 25 ns. For each dataset, we ensemble NNs of different sizes.

From our experiments, we find that averaging works the best. As seen in Fig. 4.1, we see that for the small (JSC-S), medium (JSC-M), and large (JSC-L) models, averaging increases accuracy the most. For instance, averaging JSC-S increases accuracy by 4.1%, whereas bagging and AdaBoost increase accuracy by at most 1.8% and 0.4%, respectively. Similarly for MNIST, the extra small model (MNIST_XS) increases in accuracy by 4.4% at ensemble size 16, whereas bagging and AdaBoost increase accuracy by 3.6% and 3.1%, respectively (Fig. 4.2). For HGCal, averaging the encoders together lowers EMD significantly, where lower is better. Our goal is

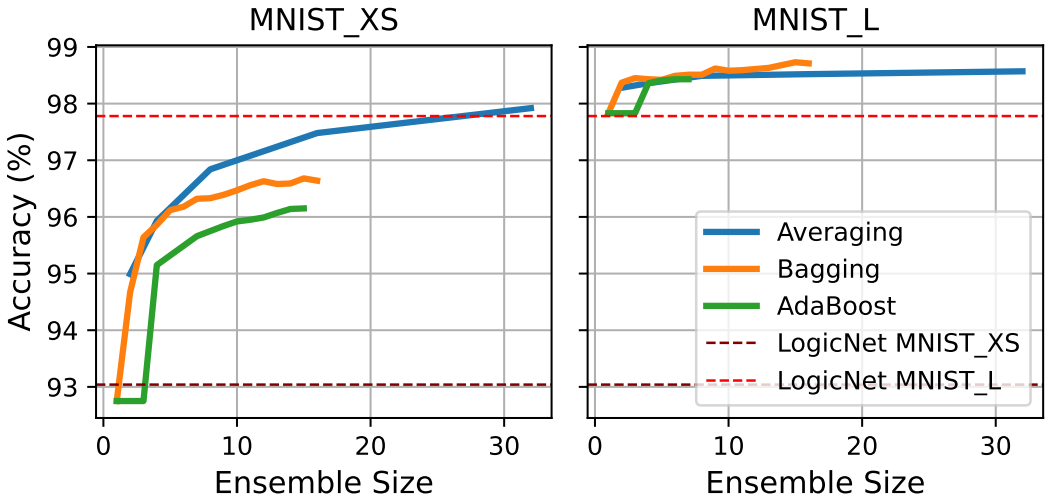


Figure 4.2. MNIST Ensembling Comparison. The extra small model MNIST_XS ensembles better than MNIST_L because as ensemble size increases it increases in accuracy more.

to hit ∼1.1 EMD, which is a dense, `float32` NN's performance. As seen in Fig. 4.3, 32 small models (HGCal_S) decreases EMD by 28%, whereas bagging and AdaBoost decrease EMD by 5% and 9%, respectively. For the large HGCal model (HGCal_L), averaging 16 models decreases EMD the most, but the trend is nonlinear. This is likely due to the difficulty involved with encoding and decoding HGCal data, which has a larger margin of error than classifying data into a few discrete classes. In particular, when performing a NN architecture random search of ∼500 LogicNets to find a low-EMD NN, we found that any NN larger than HGCal_L led to worse EMD. Only by ensembling these models together can we significantly lower EMD to get close to our goal of 1.1 EMD.

We also observe that smaller models benefit from ensembling the most, increasing in performance significantly. In Fig. 4.2, we see that averaging MNIST_XS improves accuracy by 4.9%, whereas the large model (MNIST_L) only increases accuracy by 0.2%. Notably, bagging performs the best, but only outperforms averaging by 0.14%, with all ensembling methods quickly hitting diminishing returns after two models. Similarly, in Fig. 4.3, ensembling HGCal_S improves EMD by 28% whereas the large model (HGCal_L) only improves by 17%.

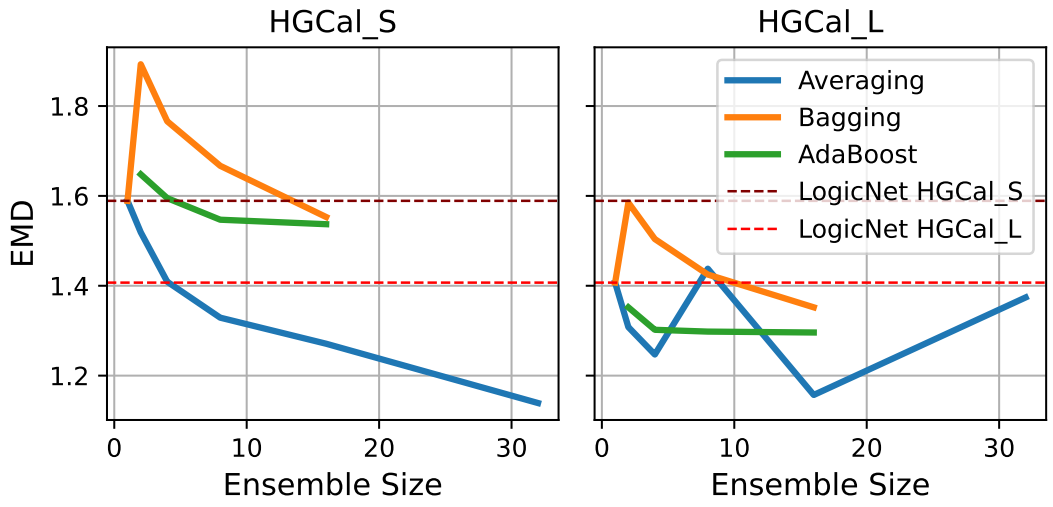


Figure 4.3. HGCal Ensembling Comparison. The small model HGCal_S responds to averaging the best, decreasing EMD (lower is better) by 28% whereas HGCal_L responds nonlinearly, decreasing EMD by only 17%.

4.3.2 Analyzing ensemble performance

We find averaging to perform the best, but we also want to understand why it does well whereas bagging and AdaBoost underperform. For instance, for JSC, AdaBoost decreases accuracy for all three models. As previously stated in Sec. 4.1 and Sec. 4.2, ensembling performs well because it promotes diversity, especially among weaker models, so that they can make a better decision collectively [217]. But, as we have seen, sometimes this is not the case. To that end, we analyze ensemble performance via a diversity analysis.

Diversity analyses have been performed in the machine learning community to better understand ensembling performance, primarily on classification tasks [120, 183]. Previously, researchers have quantified diversity by measuring disagreement among ensemble members as a proxy for diversity. In particular, recent work [183] argues that balancing disagreement with model error is important for good ensemble performance. We conduct a similar diversity analysis on our classification tasks, namely JSC and MNIST, to better understand their ensemble performance. Our diversity analysis can be used to analyze ensemble performance for not only other LUT-based NNs but any NN.

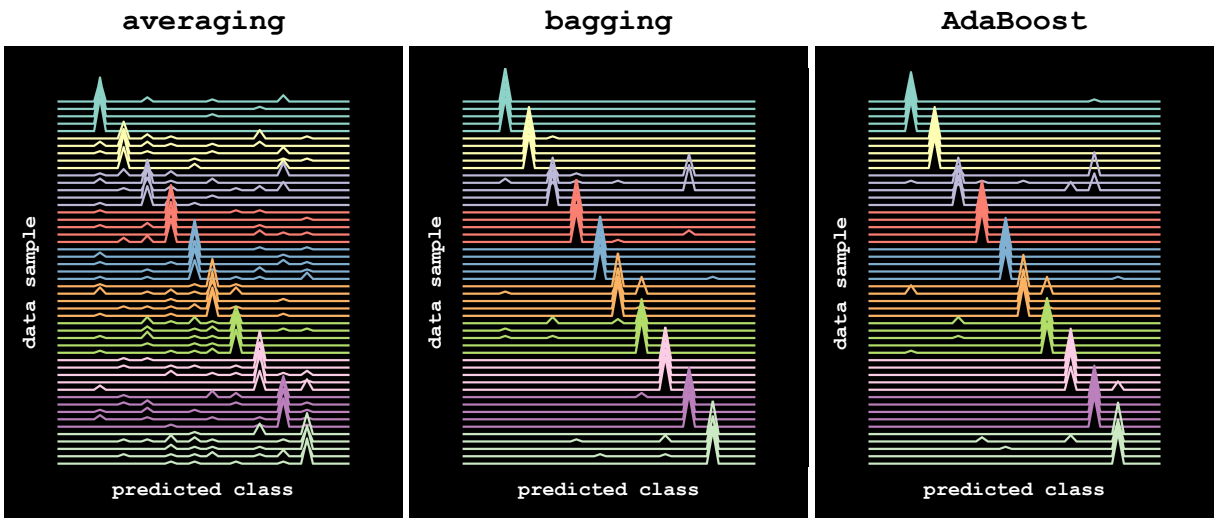


Figure 4.4. Diversity plots for MNIST_XS. Averaging exhibits the most diversity, as seen in the small peaks off the diagonal (which are the incorrect class predictions), meaning there is some disagreement among the ensemble members. Diversity has been shown to improve ensembling, confirming averaging's superior performance.

We introduce a new way of visualizing the diversity of ensemble member prediction, as seen in Fig. 4.4, Fig. 4.5, and Fig. 4.6, by showing ensemble member voting across individual samples (using $n = 50$). Our diversity visualizations reveal the extent to which the members of an ensemble agree with each other. We show diversity plots for the different ensembling methods on the MNIST and JSC tasks in Fig. 4.4 and Fig. 4.5, respectively. For the MNIST task, we show the diversity plots for MNIST_XS and ensemble size 16. For the JSC task, we show the diversity plots for JSC_S and ensemble size 32. Note, the x-axis corresponds to the different classes, and the y-axis corresponds to an input sample we pass through the ensemble members individually. Each line represents a sample, and a peak in class i means an individual ensemble member predicted class i. Higher peaks mean that more ensemble members voted for that class. The samples are ordered by target class, such that the first k samples correspond to the first target ($y = 0$), the second k samples correspond to the second target ($y = 1$), and so on. The lines are grouped by color based on the class the data sample belongs to.

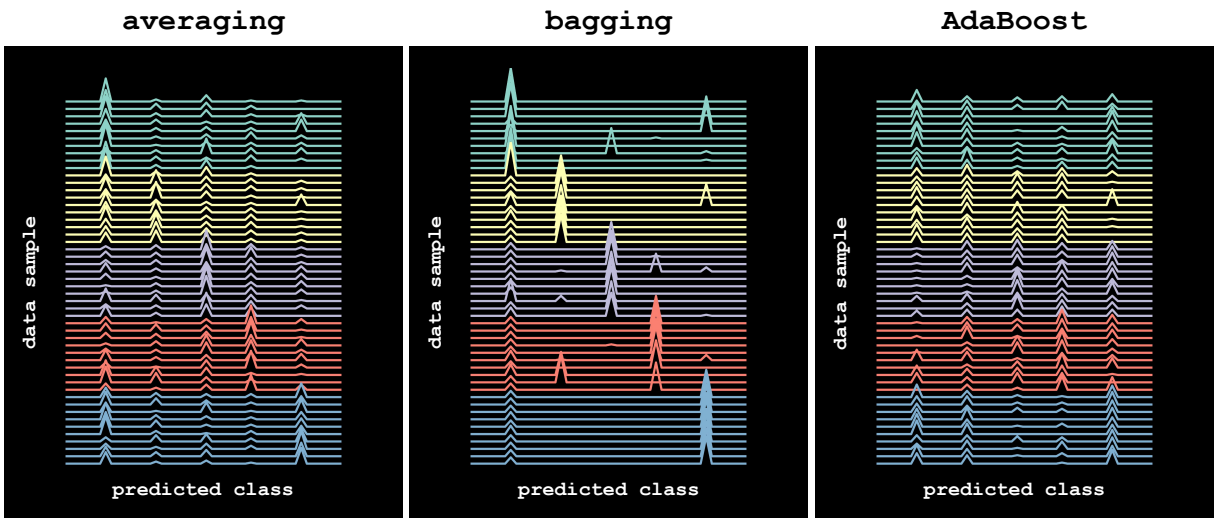


Figure 4.5. Diversity plots for JSC_S. While AdaBoost is very diverse, the high peak density off the diagonal shows that many ensemble members classify incorrectly. Averaging best balances diversity with correctness.

Thus, for an ensemble with no diversity and zero error, we should only see peaks along the diagonal. For the MNIST task where we have ten classes, we use $k = 5$ samples per class. For the JSC task where we have five classes, we use $k = 10$ samples per class. This gives a total of 50 samples per plot. While we observe much lower diversity for the MNIST task compared to the JSC task, we find that the averaging method is consistently more diverse compared to the other ensembling methods, as seen in the small peaks off the diagonal. Another interesting observation is that for the JSC task, while AdaBoost appears to be more diverse than bagging, there appears to be more density off the diagonal, suggesting more incorrect predictions. Moreover, in Fig. 4.6, we observe more diversity in averaging MNIST_XS than in MNIST_L. This corresponds to the large increases in accuracy for MNIST_XS compared with the diminishing returns of MNIST_L we observe in Fig. 4.2.

To further verify our qualitative observations, we quantify model disagreement and

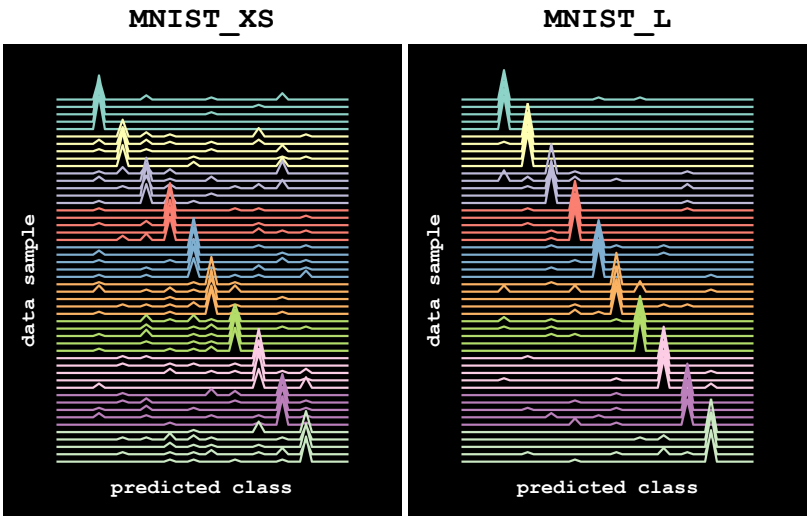


Figure 4.6. Diversity plots for averaging MNIST_XS and MNIST_L. MNIST_XS exhibits more diverse predictions than MNIST_L, suggesting small, weaker models ensemble better.

Table 4.1. Diversity metrics for the MNIST task. Bagging and AdaBoost both have low disagreement and error. Their ensemble members agree with each other a lot, implying less benefit from ensembling as compared with Averaging.

	Disagreement (↑)	**Error (↓)**	**DER (↑)**
Averaging	**0.460**	0.302	**1.522**
Bagging	0.096	**0.078**	1.232
AdaBoost	0.099	0.082	1.208

compute the disagreement error ratio (DER) [183]. DER is defined as

$$\text{DER} = \frac{\mathbb{E}_{m_i,m_j\sim\rho}[\text{Disagreement}(m_i,m_j)]}{\mathbb{E}_{m_i\sim\rho}[\text{Error}(m_i)]} \tag{4.1}$$

provided that $\mathbb{E}_{m_i\sim\rho}[\text{Error}(m_i)] \neq 0$, and where ρ is the distribution of members m_i in the ensemble, $\text{Disagreement}(m_i,m_j)$ is the prediction disagreement between the two ensemble members m_i and m_j, and $\text{Error}(m_i)$ is the prediction error for ensemble member m_i. For a given

Table 4.2. Diversity metrics for the JSC task. Averaging balances disagreement and error the best, achieving high DER.

	Disagreement (↑)	**Error (↓)**	**DER (↑)**
Averaging	0.704	0.620	**1.137**
Bagging	0.316	**0.374**	0.844
AdaBoost	**0.752**	0.693	1.085

data distribution $\mathscr{D}$, the disagreement between any two ensemble members is defined as

$$\text{Disagreement}(m_i, m_j) = \mathbb{E}_{x\sim\mathscr{D}}[\mathbf{1}(m_i(x) \neq m_j(x)] \tag{4.2}$$

where $\mathbf{1}(m_i(x) \neq m_j(x))$ is 1 if $m_i(x) \neq m_j(x)$ else it is 0. The prediction error for a single ensemble member is defined as

$$\text{Error}(m_i) = \mathbb{E}_{x,y\sim\mathscr{D}}[\mathbf{1}(m_i(x) \neq y)] \tag{4.3}$$

where $\mathbf{1}(m_i(x) \neq y)$ is 1 if $m_i(x) \neq y$ else it is 0. Importantly, DER provides a measure of diversity that balances model disagreement with prediction error, and a higher DER suggests that ensembling should improve performance [183].

We report these metrics (based on $n = 680$ samples) for the MNIST and JSC tasks in Tab. 4.1 and Tab. 4.2, respectively. For the MNIST task, we report the metrics for MNIST_XS and ensemble size 16. For the JSC task, we report the metrics for JSC_S and ensemble size 32. Notably, averaging had the highest DER across both tasks. While bagging has the lowest prediction error, it also had the lowest disagreement, implying low diversity. For the JSC task, while AdaBoost had a slightly higher level of disagreement compared to averaging, it also made more errors. Together these metrics reveal differences in the diversity of ensemble members that are related to model performance. We find that averaging balances disagreement and error the best, leading to superior ensembling performance.

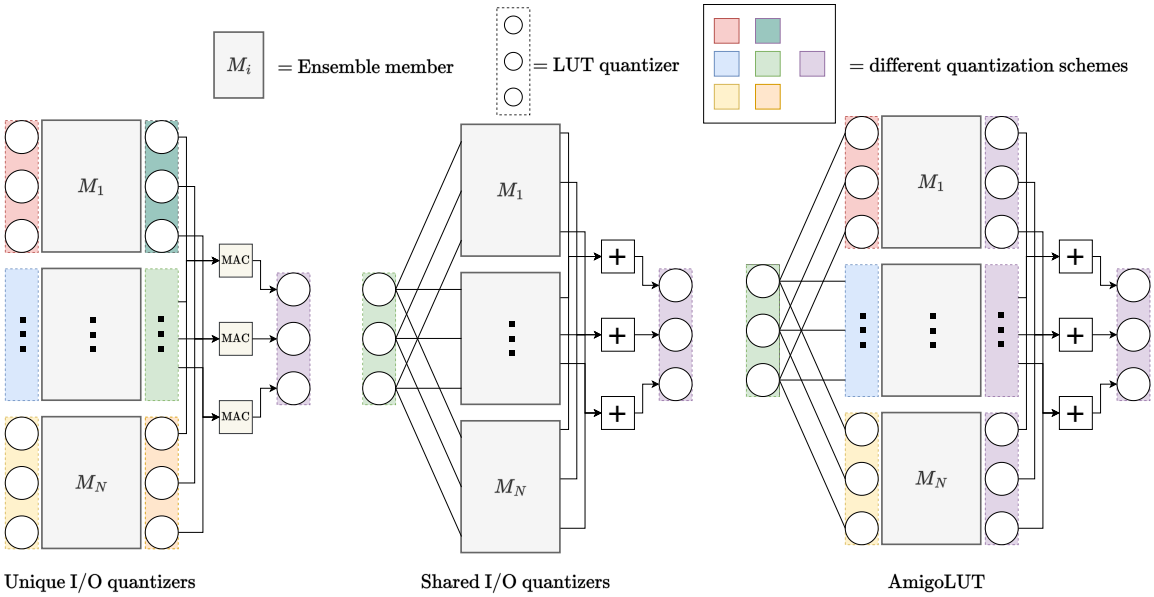


Figure 4.7. Mapping ensembles to FPGAs requires careful consideration of how to handle the different quantization schemes exhibited by each ensemble member M_i's inputs and outputs (a.k.a. I/O). With unique I/O quantizers (left), we must implement both input and output quantizers per M_i. This becomes expensive when quantizing from say JSC's 16-bit input data to the 2-bit precision each M_i expects. To combine outputs of different quantization schemes, we also need to perform a multiply-accumulate (MAC) to scale the outputs based on their learned scaling factor so that they can be combined. Ideally, we would share I/O quantizers (middle), however this decreases ensemble performance, especially for difficult tasks. AmigoLUT (right) balances accuracy and resources by first quantizing the input data to low precision (i.e., 6 bits) with a single shared quantizer (green) before sending the data to each M_i to be uniquely quantized. Each M_i's input neuron can be quantized with only two LUT6 to the lower 2-bit precision it expects. At the output, Tailor implements an extra NN layer that quantizes each M_i's output to the same quantization scheme so that they can be summed together efficiently.

4.4 Quantization Architecture Tradeoffs

As seen in the previous section, averaging works the best among the ensembling methods we study. Next, we want to implement this ensemble on an FPGA; however, this presents many challenges in minimizing resource overhead while preserving ensemble performance.

If we were to directly map the averaging ensembles in the last section to an FPGA, we would need to map an ensemble that looks like the design on the left in Fig. 4.7. As previously stated in Sec. 4.1 and Sec. 4.2, LogicNets are heavily quantized to minimize LUT utilization.

LogicNets are quantized using Brevitas [153], which quantizes values to a given precision, say 6-bit integers. However, to improve quantized NN performance, Brevitas learns a floating point scaling factor. The Brevitas quantizer works by mapping activations to a limited set of discrete integer levels based on the specified bit-width (e.g., 2 bits). However, the output of the quantizer remains in floating-point format because these integer levels are scaled by a learned floating-point scale factor and shifted by a learned zero-point (bias). This becomes an issue when we want to combine ensemble outputs efficiently. When we naively train an ensemble of LogicNets, Brevitas will learn different floating point scaling factors for each ensemble member's input and output quantizers—even though the inputs and outputs are quantized to the same bitwidth. As a result, in hardware, we would have to implement a unique input quantizer per model in the ensemble. This is resource-intensive. For instance, the HGCal data is quantized to 8 bits and each LogicNet model expects say 4-bit inputs, so we would have to implement N $8 \rightarrow 4$ bit input quantizers, which costs $N\times$ as many resources compared with mapping a single LogicNet to an FPGA. Moreover, since each model's output also exhibits a different quantization scheme, we would need to scale each model's output by the floating point scaling factor Brevitas learns. This requires multiplication using either DSPs or LUTs, if we can manage to also quantize the floating point scaling factors to integers. As seen in Fig. 4.7 (left), combining all ensemble outputs requires implementing multiply-accumulate (MAC) operations, costing extra resources.

Ideally, we would want to implement the ensemble such that all models share a single input quantizer and quantize the outputs using the same scaling factor, as seen in Fig. 4.7 (middle). By quantizing the outputs using the same scaling factor, we would only need to sum the ensemble member's outputs to get the final result. This would drastically reduce the resource overhead needed to quantize the inputs and combine the outputs. However, quantizing all of the inputs and outputs in a similar manner leads to poor performance, even undercutting the benefits of ensembling. In Fig. 4.8, we compare the performance of using a unique input/output (I/O) quantizer per ensemble member versus an I/O quantizer shared between all ensemble members. While for MNIST sharing I/O quantizers does not reduce accuracy by much (¡1%), performance

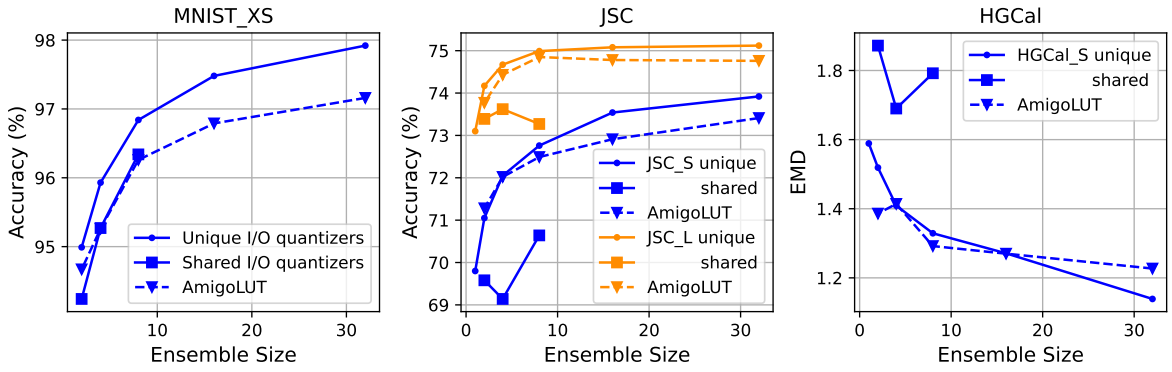


Figure 4.8. Unique quantizers vs. naively shared quantizers vs. AmigoLUT case study on LogicNets. The three graphs show the accuracy or EMD (y-axis) vs. the ensemble size (x-axis) for three different tasks (MNIST, JSC, and HGCal). Each graph compares the different input and output quantization methods as described in Fig. 4.7. The JSC application shows two base model—small (S) and large (L). Lower EMD results (HGCal) indicate better model performance.

reduces drastically for JSC and HGCal tasks. This is likely because MNIST is an easier task than JSC and HGCal. When ensembling eight JSC_S models, using unique quantizers improves accuracy by 2.1%, while using shared I/O quantizers only improves accuracy by 0.8%, reducing the benefit of ensembling. Similarly, for HGCal, sharing I/O quantizers reduces HGCal_S's EMD by 35% to 1.792, which is worse than its single model EMD of 1.589, resulting in no benefit from ensembling. As a result, we need a way to maintain ensemble performance when mapping ensembles to FPGAs without significantly increasing resource overhead.

4.5 AmigoLUT

To solve the challenges presented in the previous section, we introduce AmigoLUT, a method for creating ensembles of LUT-based NNs that map to FPGAs efficiently. The idea behind AmigoLUT is to balance the performance gains from unique I/O quantizers with the resource efficiency of shared I/O quantizers. AmigoLUT quantizes the input data to low precision (i.e., 6 bits) with a single shared quantizer before sending the data to each ensemble member. Then, each ensemble member's input neuron can be quantized uniquely with a single LUT6 per bit to achieve the lower precision it expects. At the output, AmigoLUT implements an extra NN

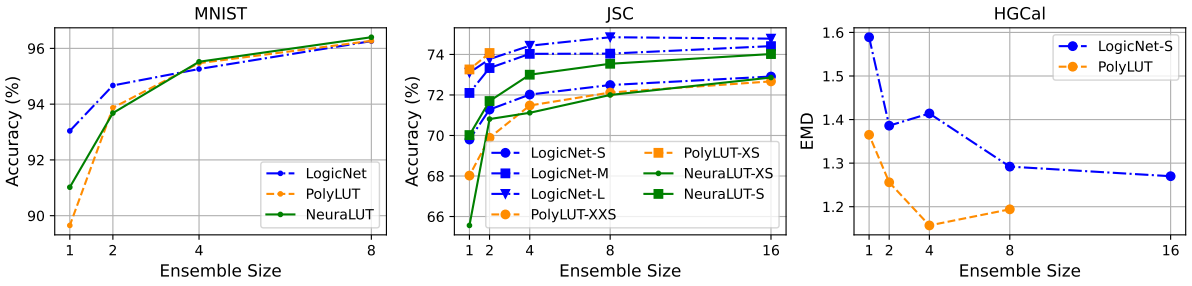


Figure 4.9. AmigoLUT can ensemble different LUT-based NNs (LogicNet, PolyLUT, and NeuraLUT) with different sizes (XXS, XS, S, M, L) to increase their accuracy. The accuracy improvements depend on the difficulty of learning the underlying task (MNIST, JSC, HGCal). In general, ensembling has greater relative improvement with smaller models compared to larger models.

layer that quantizes each ensemble member's outputs to the same precision so that they can be summed together efficiently on an FPGA. Fig. 4.7 (right) describes our design.

4.5.1 Training results

Fig. 4.8 shows the accuracy improvements that AmigoLUT achieves with ensembling. We consider three tasks: MNIST, JSC, and HGCal; each graph corresponds to one task. We only consider LogicNets here and use averaging as it gives the best results. As before, we see that the accuracy generally increases as we add additional ensemble members until it typically plateaus. Note that a higher accuracy percentage is better in the MNIST and JSC graphs, but in HGCal, a lower EMD is better.

The graphs show three different methods of quantization: unique I/O quantizers, shared I/O quantizers, and the hybrid quantization scheme that we use in AmigoLUT (see Fig. 4.7). The general trend is that the unique I/O quantizer gives the best results, which is unsurprising given that it is unconstrained; each input and output can be uniquely quantized, providing the most flexibility.

In general, AmigoLUT's quantization scheme sits between the unique I/O quantization scheme and the shared I/O quantization scheme. In MNIST, AmigoLUT and shared quantization are very similar. Both provided less than a 1% accuracy drop from the unique I/O quantization.

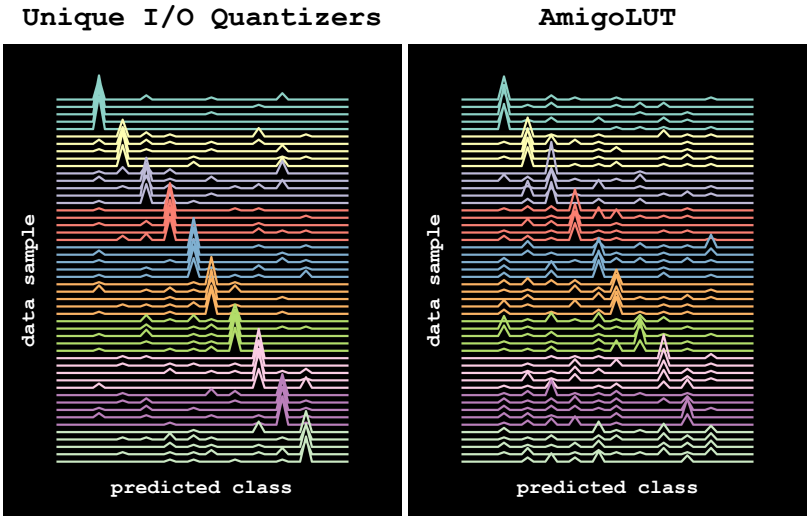


Figure 4.10. Diversity plots for MNIST with unique I/O quantizers versus AmigoLUT.

The JSC results show two models: small (S) and large (L). The larger models perform better than the small models across all ensemble sizes. Ensembling the small model achieves the accuracy of the baseline large model (ensemble size 1). AmigoLUT ensembling has very similar accuracy compared to unique I/O quantization. The shared I/O quantization method performs poorly for JSC. In general, AmigoLUT reduces ensemble performance by ¡1% compared with unique I/O quantizers for MNIST and JSC. HGCal results show a similar trend. The EMD is lower (better) for the unique and AmigoLUT quantization schemes, with the shared scheme being noticeably worse. The AmigoLUT quantization has the best results of all three quantization schemes for some ensemble sizes. For example, when ensembling two models together, AmigoLUT improves EMD by 9% compared with unique I/O quantizers. In summary, AmigoLUT significantly improves ensemble performance compared to the naive approach of shared I/O quantizers.

Similar to Sec. 4.3.2, we visualize ensemble member prediction diversity by showing ensemble member voting across individual samples (using $n = 50$). We compare the diversity of averaging models with unique I/O quantizers to models ensembled with AmigoLUT on the MNIST and JSC tasks in Fig. 4.10 and Fig. 4.11, respectively. We observe similar diversity patterns, suggesting that the training with AmigoLUT preserves ensemble diversity in addition

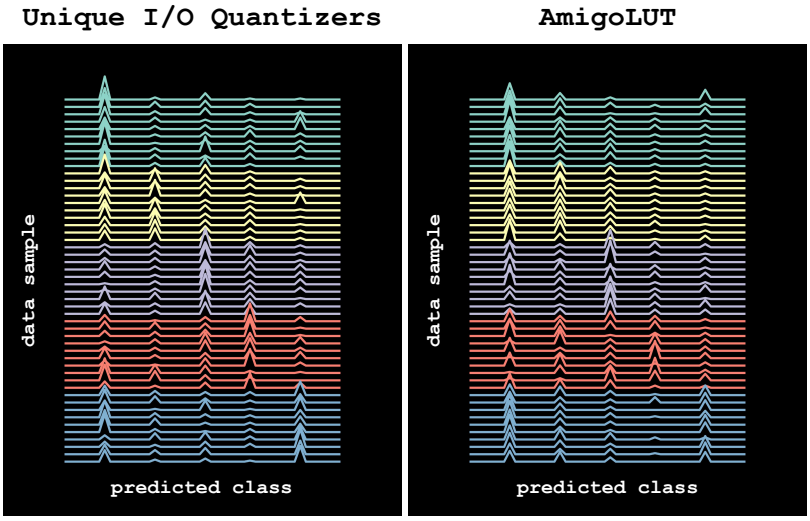


Figure 4.11. Diversity plots for JSC with unique I/O quantizers versus AmigoLUT.

to performance.

4.5.2 Hardware results

We study the effect of ensembling different types of models in terms of their neuron structure (LogicNet, PolyLUT, NeuraLUT) and size (number of neurons/LUTs). Fig. 4.9 presents accuracy results across three tasks (MNIST, JSC, and HGCal) with different NN model architectures.

Tab. 4.3 shows the base model architecture parameters considered throughout the experiments. These models were selected by performing a model architecture search to find high-quality models that provided tradeoffs between model complexity/size and accuracy. The base models are replicated for each ensemble member; the structure of the models of an ensemble are identical though their weights will differ.

The general trend is that ensembling increases the accuracy regardless of the underlying NN model. However, the rate of improvement from ensembling depends upon the task being learned by the NN and the base ensemble model size (S, M, L, etc.). Ensembling generally sees its greatest relative gains with smaller ensemble members. For example, MNIST sees a

Table 4.3. Base model architecture used for AmigoLUT ensembling, (i.e., the parameters of each ensemble member). "Width" and "degree" are NeuraLUT and PolyLUT parameters, respectively.

Dataset	Model	Neurons per layer	Input bit width	Input fan-in	Layer bit width	Layer fan-in
JSC	AmigoLUT-LogicNet-S	64, 32, 32, 32, 5	2	3	2, 2, 2, 2, 2	3, 3, 3, 3, 3
	AmigoLUT-NeuraLUT-XS (width = 16)	64, 5	2	3	2, 2	3, 3
	AmigoLUT-NeuraLUT-S (width = 16)	64, 5	4	3	2, 2	3, 3
	AmigoLUT-PolyLUT-XXS (degree = 3)	64, 5	2	3	2, 2	3, 3
	AmigoLUT-PolyLUT-XS (degree = 3)	64, 5	2	3	2, 2	3, 3
MNIST	AmigoLUT-LogicNet-XS	1024, 1024, 128, 10	1	8	1, 1, 1, 4	8, 8, 8, 8
	AmigoLUT-NeuraLUT (width = 16)	600, 300, 300, 10	2	4	2, 2, 2, 2	4, 4, 4, 4
	AmigoLUT-PolyLUT (degree = 3)	600, 300, 300, 10	2	4	2, 2, 2, 2	4, 4, 4, 4
HGCal	AmigoLUT-LogicNet-S	128, 256, 128, 128, 16	4	2	2, 2, 2, 6, 6	6, 5, 4, 2, 6
	AmigoLUT-PolyLUT (degree = 2)	512, 512, 64, 16	6	2	3, 3, 4, 5	4, 5, 3, 3

several percentage point increase when going from one ensemble member (baseline) to two ensembles in Fig. 4.9, which is true across all of the different types of models (LogicNet, PolyLUT, NeuraLUT). Increasing the ensemble size from two to four members increases the accuracy further but not as dramatically. The returns of increasing the ensemble size generally continue to diminish with more ensemble members.

The JSC task considers more than one model for LogicNet, PolyLUT, and NeuraLUT; denoted by the appended relative sizes (XXS, XS, S, M, L) from smallest to largest w.r.t. the number of LUTs. Consider LogicNet models with ensemble size 1. LogicNet-S has the lowest accuracy, LogicNet-M is approximately 2% more accurate, and LogicNet-L is the most accurate. As we ensemble each of these models, they all get more accurate at relatively the same rate, with the rate of change in accuracy being higher at lower ensemble members and relatively unchanged at ensemble sizes 8 and 16. The PolyLUT-XXS and PolyLUT-XS models follow a similar trend.

The HGCal task is likely the most challenging of the three tasks. First, as an autoencoder, the output result is much more continuous than the rather small, discrete number of output results

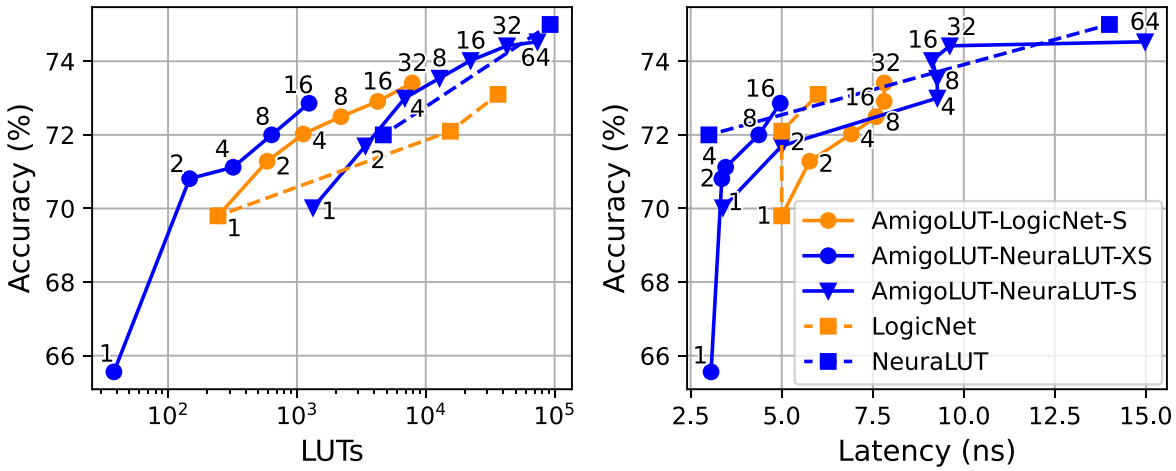


Figure 4.12. Jet Substructure Pareto Front. The results compare accuracy, LUTs, and latency. The solid lines are the AmigoLUT models with labels indicating the number of ensemble members. The dotted lines are the standalone versions of the LUT-based NNs we ensemble. AmigoLUT with NeuraLUT as its ensembling model gives small, accurate, and low latency results. The latency only increases slightly as ensemble size increases until we hit large ensembles (i.e., 64 models).

for MNIST and JSC, 10 and 5 classes, respectively. Regardless, we see that ensembling HGCal encoders yields better EMD. The EMD reduces as the ensemble members increase, meaning the error is lower; an EMD $= 0$ is the most accurate result.

Our next series of experiments focuses on comparing the best models across LUT-based NNs and model sizes. We compare our AmigoLUT results for ensembling against the standalone models that we ensemble, showing how AmigoLUT improves the scalability of LogicNets and NeuraLUT. We also compare with the published related work. We synthesize and place-and-route our ensembles with Vivado 2020.2, targeting part `xcvu9p-flgb2104-2-i`, a Xilinx Virtex UltraScale+ FPGA. We compile the projects using Vivado's `Flow_PerfOptimized_high` setting and execute synthesis in `Out-of-Context` mode.

We note that prior work assumes that the inputs are quantized and this quantization is done offline, (i.e., there is no quantization hardware on the inputs). This is a reasonable assumption, but the reality is that this quantization must be performed somewhere. To make a

Table 4.4. Comparing model resource utilization and performance metrics across the three datasets/tasks with prior work.

Dataset	Model	Accuracy /EMD	LUT	FF	DSP	BRAM	Latency (ns)	FMax (MHz)	Area × Delay (LUT × ns)
MNIST	**AmigoLUT-LogicNet-XS** (2 models)	94.7	9 711	9 047	0	0	12.3	569	119 445
	AmigoLUT-NeuraLUT (4 models)	95.5	16 081	13 292	0	0	**7.6**	**925**	122 216
	PolyLUT [10]	96	70 673	4 681	0	0	16	378	1 130 768
	NeuraLUT [11]	96	54 798	3 757	0	0	12	431	657 576
	PolyLUT-Add [119]	96	14 810	2 609	0	0	10	625	148 100
	DWN [17]	**97.8**	**2 092**	**1 757**	0	0	9.2	873	**19 246**
JSC	**AmigoLUT-NeuraLUT-XS** (4 models)	71.1	320	482	0	0	3.5	**1 445**	1 120
	AmigoLUT-NeuraLUT-XS (16 models)	72.9	1 243	1 240	0	0	5.0	1 008	6 215
	AmigoLUT-NeuraLUT-S (32 models)	74.4	42 742	4 717	0	0	9.6	520	410 323
	LogicNet-L	73.1	36 415	2 790	0	0	6	390	218 490
	PolyLUT	72	12 436	773	0	0	5	646	62 180
	PolyLUT	75	236 541	2 775	0	0	21	235	4 967 361
	NeuraLUT	72	4 684	341	0	0	**3**	727	14 052
	NeuraLUT	75	92 357	4 885	0	0	14	368	1 292 998
	PolyLUT-Add	75	36 484	1 209	0	0	16	315	583 744
	PolyLUT-Add	72	895	1 649	0	0	4	750	3 580
	DWN	73.7	**134**	**106**	0	0	3.7	1 361	**496**
	DWN	**76.3**	6 302	4 128	0	0	14.4	695	90 749
HGCal	**AmigoLUT-LogicNet-S** (2 models)	1.386	**26 400**	4 049	0	0	15.6	**512**	411 840
	AmigoLUT-LogicNet-S (16 models)	**1.270**	195 724	23 515	0	0	24.0	334	4 697 376
	LogicNet-L	1.407	32 529	**2 340**	0	0	**12.2**	323	**396 854**

fair comparison, we assume that the AmigoLUT inputs are similarly quantized to 6 bits offline; however, we do add the hardware required to quantize those 6 bits into say unique 2 bit inputs for each ensemble member (see Fig. 4.7).

Tab. 4.4 shows a selection of ensembling models developed in this work and results presented in previous works across the three tasks. The table provides precise numbers for all the relevant metrics (accuracy, LUTs, FFs, latency, FMax, and area delay product) typically discussed in related work. Note that the high FMax numbers are a direct result from synthesizing in `Out-of-Context` mode. Most FPGA devices do not provide a 1 GHz clock, so in a real system we would be limited by the global clock network, which is around ∼800 MHz for the VU9P device we target. In Tab. 4.4, we see that DWN and PolyLUT-Add provide superior performance for some benchmarks compared with AmigoLUT; however, we demonstrate AmigoLUT's potential for improving these works' state-of-the-art performance in the following figures and discussion. In particular, we show how AmigoLUT significantly reduces LUT utilization for the models we ensemble, specifically LogicNet and NeuraLUT.

Fig. 4.12 shows different models' accuracy versus latency and LUTs. The models include

our AmigoLUT ensembling results (solid lines) and their standalone versions (dotted lines). The labels indicate the number of ensemble members. The different models in the standalone LUT-based NNs architectures are generated by changing the model architecture, (e.g., the number of layers, the number of neurons per layer, the layer bit width, and the layer fan-in). Our results show that selecting the best model architecture parameters is challenging; it requires extensive tuning and trial and error.

Fig. 4.12 shows that adding more ensemble members increases the accuracy, resulting in a linear increase in LUTs. We also see over an order of magnitude drop in LUT utilization when comparing a standalone LogicNet versus AmigoLUT-LogicNet at 72% accuracy. Except for large ensembles (i.e., ensemble size 64), the latency stays relatively the same as ensemble size increases. Note that the latency for larger non-ensembled models increases substantially; thus, there is a clear benefit for increasing model size/accuracy using AmigoLUT compared to the more traditional NN architecture search strategy done in previous work (modifying the layers, neurons per layer, etc.). This is seen in the drastic increase in latency for the higher accuracy dotted line results.

The AmigoLUT architectures scale well regardless of the baseline architecture model characteristics, though tuning the baseline architecture does affect the overall quality. A better baseline model will generally ensemble better than a lower quality baseline model.

Fig. 4.13 shows results for MNIST, which are relatively similar to JSC. Ensembling has a linear relationship between ensemble members and LUTs. The latency increases as more ensemble members are added, but generally, ensembling provides a good tradeoff between accuracy, LUTs, and latency.

The HGCal dataset was not considered in previous LUT-based NN publications. We feel this is a good benchmark for LUT-based NNs because it has become a canonical task in the FastML community. It has very high constraints on latency and throughput and thus is a task well-matched for LUT-based NNs. Furthermore, it has a more continuous output space as it is trying to compress data.

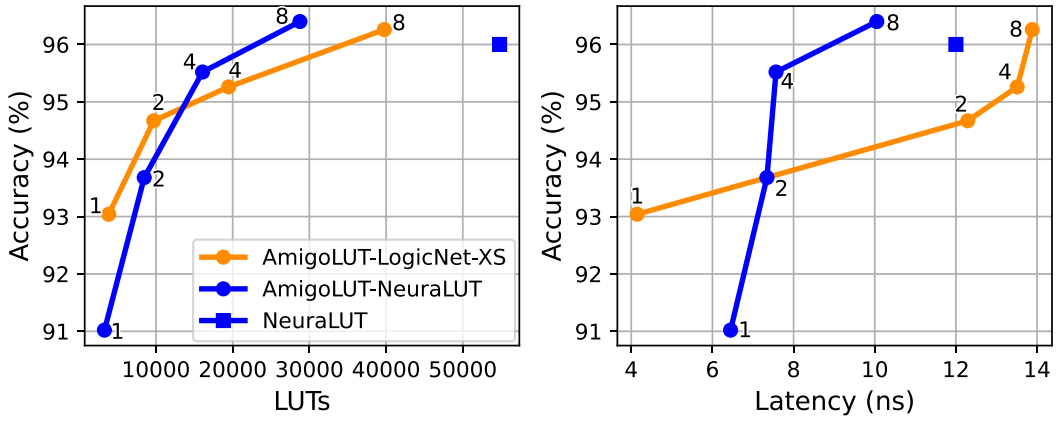


Figure 4.13. MNIST Pareto Front comparing accuracy, LUTs, and latency. Solid lines are different ensemble sizes (noted by the labels) generated from the same base model. The remaining point is a standalone NeuraLUT.

We generated the LogicNets results by performing a NN architecture search that modified the number of layers, the number of neurons per layer, the input size, etc. We performed a design space exploration to find architectures that would train well and provide good tradeoffs between accuracy and size. We could not find a larger LogicNets architecture that provided better accuracy with a reasonable number of LUTs. We considered nearly 500 models in this search exploration, which highlights the challenges of scaling up LUT-based NN architecture. As our results show in Fig. 4.14, we used ensembling to generate larger models with better EMD.

4.6 Summary

We address the challenges of implementing large, high-performance NNs for tasks that require high throughput and low latency. AmigoLUT creates ensembles of smaller LUT-based NNs that map efficiently to FPGAs. AmigoLUT provides a straightforward methodology to increase model accuracy while maintaining a linear scaling in LUTs, minimally increasing latency, and creating a high-frequency design. We evaluate three ensemble methods (averaging, bagging, and AdaBoost) and show that averaging outperforms the other methods. We analyze the ensemble's performance using novel diversity plots and disagreement metrics, which help explain

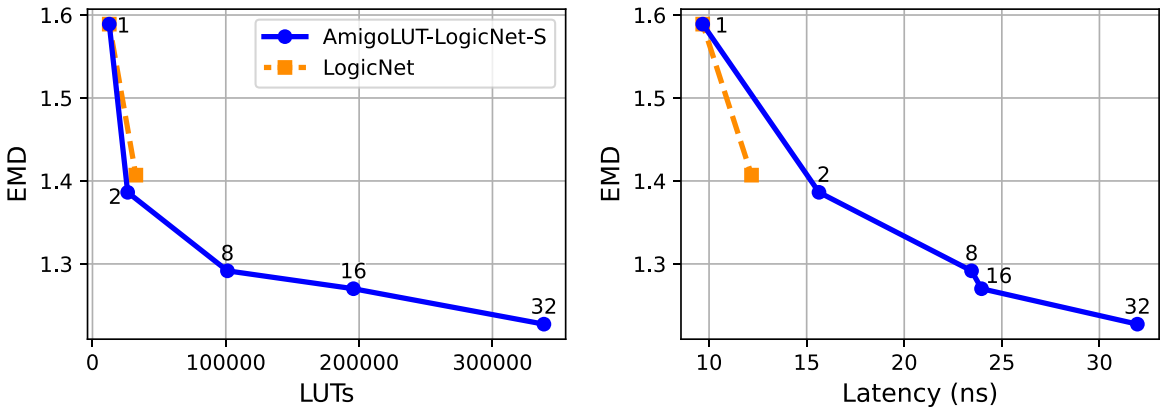


Figure 4.14. HGCal Pareto Front comparing accuracy, LUTs, and latency. The labels on the solid lines indicate the number of ensemble members. We only achieved better EMD through ensembling. We did not find a larger LogicNet model that provided better EMD with a reasonable number of LUTs.

why averaging is effective, and provide tools to evaluate the general quality of ensembling LUT-based NNs and NNs in general. We describe architecture tradeoffs for efficiently implementing the ensembles, developing a quantization architecture that allows for effective ensemble scaling to increase accuracy while minimizing resource usage. We perform a comprehensive study on three tasks, showing that AmigoLUT can effectively ensemble existing LUT-based NN models, demonstrating the potential that ensembling has to scale up state-of-the-art and future LUT-based NNs.

Chapter 4, in part, is a reprint of the material as it appears in Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '25). The work in this chapter was done in collaboration with Marta Andronic, Danial Zuberi, Jiaqing Chen, Caleb Geniesse, George A. Constantinides, Nhan Tran, Nicholas J. Fraser, Javier Mauricio Duarte, and Ryan Kastner. The dissertation author was the primary investigator and author of this work.

Chapter 5

FKeras: A Sensitivity Analysis Tool for Edge Neural Networks

In the previous two chapters, I studied how to implement edge NNs for efficient inference on FPGAs. I altered the hardware architectures to best suit the given edge NNs. Now, I focus on how to design resilient edge NNs, in particular how to effectively assess their fault tolerance in software. Assessing an edge NN's resilience efficiently is a critical step to designing with PPAR in mind because it often involves time-consuming fault injection campaigns. FKeras takes a first step in this dissertation to conduct fault injection campaigns in a way that targets fault-sensitive bits first. This way designers can avoid injecting faults in insensitive bits.

5.1 Introduction

Machine learning (ML) is increasingly used in safety-critical applications, including autonomous vehicles [24, 31, 184], healthcare [6, 13, 171], and scientific experiments [33, 50, 61]. In these domains, the ML computation must act reliably in the face of errors. Soft errors are a common source of unreliability [26, 53, 92, 95], which are difficult to avoid and often require mitigation. For example, a particle strike can cause a bit flip in neural network (NN) weights, which can lead to incorrect results.

Prior work has investigated the fault tolerance of NNs extensively; however, most studies have focused on faults occurring in the outputs of NN layers, namely in pipeline registers and

arithmetic logic units (ALUs). They also assume that the weights reside in memory that is protected by error correction codes or parity [15, 40, 113]. This is often the case for large NNs that rely on retrieving weights from off-chip memory, e.g., DRAM, when they are run on GPUs because all the weights cannot fit on chip [128]. This "off-chip inference" is a popular case, especially as researchers add increasingly more parameters to improve NN performance, e.g., GPT-3 has 175 billion parameters, which amounts to 350 GB of data when stored with float16 precision [29, 202]. By focusing on NNs that execute via off-chip inference, previous studies have neglected to study fault tolerance in an emerging class of NNs where inference occurs *completely on chip* [18, 27].

On-chip inference, wherein a NN executes entirely on chip, is increasing in demand, particularly in scientific fields like high energy physics and electron microscopy [27, 50, 61, 81, 197]. As scientific instruments improve, they are able to produce data at extremely high rates, e.g., TBs/s. With so much data, scientists are contending with how to best collect and analyze so much data. Many are looking to NNs to process the data because traditional methods have reached their limits [202].

Consider the CERN Large Hadron Collider (LHC) Compact Muon Solenoid (CMS) experiment [182], which runs particle collision experiments that generate data rates of $\sim$40 TB/s. To reduce data rates, LHC physicists plan to deploy tens of thousands of encap concentrator (ECON-T) ASICs [81], each running a NN encoder, to compress experimental data from the high-granularity endcap calorimeter (HGCAL) [47] into a smaller format for easy filtering in the trigger system. The ECON-T encoder hardware must accept new input data at 40 MHz and complete inference in 25 ns within an area budget of 4 mm^2 [81]. To meet these constraints, the ECON-T:

1. is a small two-layer NN with $\sim$2000 parameters,
2. is quantized to have 6-bit fixed-point weights, and
3. operates completely on-chip.

To complicate matters further, the ECON-Ts operate in a high-radiation environment due to their close proximity to particle collisions in the LHC. High radiation causes transient hardware errors, which can lead to incorrect application output (silent data corruptions) if the hardware is not designed robustly. The ECON-Ts filter terabytes per second of data for high-energy physics studies, and faulty execution is unacceptable. Only the NN weight parameters are vulnerable to faults because the activations are not stored in on-chip memory for longer than a cycle, as inference completes in a single cycle.

From this example, we see that on-chip inference NNs must be small enough to be run fast enough in the given resource constraints. These small NNs are often referred to as "edge NNs." Many edge NNs have unique characteristics that have not been considered extensively in prior work, namely:

1. heavily quantized ($\leq$ 8 bits) fixed-point weights, and
2. fully on-chip inference that expose weights to faults.

Some techniques from prior work translate well to edge NNs while others do not. Most prior work has studied NN faults by conducting fault injection (FI) campaigns. FI simulates faults in software or hardware and passes a number of inputs through the NN to assess performance. This is computationally expensive because an enormous amount of faulty scenarios must be simulated. Previous work has sought to limit the search space [40, 162, 215]. One method [40] exploits how a more significant bit is at least as sensitive as a less significant bit—a finding that is consistent with our own results and that we exploit too. But, this method and others find that much of the improvements in reducing the FI search space is a direct result of using float32 to represent values. The search space of float32 is large and prior work [215] trivially cuts down the space by focusing on only the eight exponent bits. A more clever method of statistically reducing the search space directly hinges on how flipping bits in float32 values results in massive changes in magnitude [162]. This big swing in magnitude does not occur with low-precision fixed-point/integer data types because the range of values that they can represent

is considerably smaller, e.g., ECON-T's 6-bit weight contains only 1-integer bit, leading to a maximum difference of 2. Our work therefore seeks to fill this gap and *study how faults affect edge NNs built for on-chip inference*.

We introduce FKeras, a tool that assesses the sensitivity of NN weights to faults, specifically targeting edge NNs for on-chip inference.[1] FKeras includes fast and efficient metrics that provide a bit-level sensitivity ranking of the weights, including a novel metric based on the Hessian (second-order partial derivatives of the NN). Our fault tolerance metrics reveal under the hood how sensitive a NN is to faults, facilitating the addition of fault tolerance into the codesign problem by allowing the designer to consider how different quantized edge NNs handle faults. Furthermore, our metrics provide a way to evaluate fault tolerance alongside performance, resource usage, and power consumption, which can result in hardware accelerators that are smaller, more performant, and more resilient. For instance, the ECON-T uses triple modular redundancy (TMR) to protect the NN weights against faults [81]. TMR is effective but incurs a 200% overhead [22, 185], which is particularly costly when every resource counts in ECON-T's 4 mm^2 area budget. Many interesting design tradeoffs emerge when considering fault tolerance: Can one TMR a subset of the parameters to optimize the NN architecture in another manner? Which computation and data are the most important to protect against faults? How does quantization affect the NN's fault resilience? How do different NN architectures compare with respect to accuracy, performance, and fault tolerance?

FKeras facilitates fault analysis with hls4ml [60]. hls4ml targets edge ML applications with low latencies, high throughput, minimal power budgets, and low resource usage [61]. FKeras extends the hls4ml workflow to assess the sensitivity of NN weights to faults, perform efficient FI campaigns, and facilitate design space exploration that considers fault tolerance alongside accuracy, performance, and resource usage.

hls4ml designs that target FPGAs/ASICs must satisfy unique requirements uncommon in other architectures like GPUs and TPUs [98]. First, hls4ml implementations are highly quantized,

[1] https://github.com/KastnerRG/fkeras

often using unique arbitrary precision fixed-point data types in each NN operation. Second, hls4ml implementations hold most of their data on-chip, including inputs, outputs, weights, and internal state. Third, they often operate in high-radiation or safety-critical environments. For example, the radiation of the ECON-T in the LHC is approximately 1000$\times$ that of the radiation in space. Thus, understanding the potential effects of faults is crucial.

hls4ml accelerators are often heavily quantized to meet stringent performance, power, and area requirements. Quantization reduces the computational and storage costs and modifies the sensitivity of computations to faults. QKeras [45] is a tool developed by Google and the hls4ml community to handle custom hardware data types. QKeras provides drop-in replacements for NN operations, e.g., from `Dense` to `QDense`. FKeras is modeled after QKeras, providing similar replacements for NN operators (e.g., `FQDense`). FKeras allows designers to consider fault tolerance in the context of fixed-point computations.

FKeras includes several bit-level sensitivity metrics, including

1. most significant bits first to least significant bits last (MSB $\rightarrow$ LSB),
2. the gradient, and
3. the Hessian.

We introduce a new metric based on the Hessian. The Hessian (second-order partial derivatives) captures the curvature of a NN's loss landscape, providing insight into how the NN will react to perturbations to the weights. The Hessian has been shown to capture NN sensitivity and is useful at quickly quantizing a NN to mixed precision bitwidths [55, 56, 204, 207, 208]. FKeras efficiently calculates the Hessian, which gives a highly competitive ranking in all our considered networks.

Our findings show that individual bits matter—some bits are more important than other bits. Within a weight, the importance tends to be monotonic [40], i.e., a more significant bit is at least as sensitive as a less significant bit, though there are exceptions. We then compare the relative effect of the bits across the weights using the Hessian or gradient.

FKeras can guide FI campaigns to inject faults on the most sensitive parameters first. The sensitivity of the weights is variable; many do not lead to faulty behaviors. Thus, a campaign should focus on injecting faults into weights that are the most vulnerable to faults. We can use our Hessian sensitivity metric to determine the most sensitive weights and flip those first. Fig. 5.9 shows that the Hessian sensitivity metric performs substantially better at guiding the FI towards faulty behaviors than randomly or statistically injecting faults as prior FI tools and studies do [57, 111, 159, 162].

We demonstrate the value of FKeras in considering fault analysis in the design space exploration process for a NN. We perform a design space exploration on three different NN architectures for the ECON-T autoencoder. We use FKeras to assess the resilience of these different architectures to faults alongside accuracy, performance, and resource usage. We also analyze the resilience of an edge convolutional neural network (CNN) trained on CIFAR-10 [107] that operates also completely on-chip. Our results show that different-sized networks have vastly different levels of fault tolerance, e.g., a smaller, less accurate NN has more sensitive bits than a larger, more accurate NN.

FKeras is a tool that helps design fault-tolerant NNs for on-chip inference. The primary contributions of FKeras are:

- Providing bit-level weight sensitivity metrics tailored for on-chip inference edge NNs.
- Using sensitivity metrics to guide fault injection campaigns that consider the most sensitive bits first.
- Performing NN architecture design space exploration that considers fault tolerance alongside traditional optimization criteria like performance and area.

5.2 Related Work

Prior work has sought to understand NN fault tolerance and to develop techniques to protect NNs from faults, but few have considered edge NNs that are quantized and run on

Table 5.1. Comparison of FKeras to prior work. If the work presents an FI method and is not a tool, then we list the Bit Target-ability and Quantization Support as not applicable (N/A).

FI Tool/Method	Bit Target-ability	Quantization Support	FI Speedup Method	Sensitivity Metrics
Ares [159]	Random	Fixed Point/Int	No	No
TensorFI [112]	Single, Random	No	No	No
PyTorchFI [126]	Single	Integer	No	No
GoldenEye [130]	Single	Custom	No	No
enpheeph [48]	Single, Multiple, Random	Custom	GPU support	No
[44]	N/A	N/A	N/A (Heuristic)	Gradient
BinFI [40]	Single	Limited	Element-wise binary search	No
StatFI [162]	N/A	N/A	Statistical sampling	No
FKeras	Single, Multiple, Random	Fixed Point/Int	Metric-guided FI	Hessian, Gradient, MSB $\rightarrow$ LSB

specialized hardware, such as FPGAs and ASICs, completely on chip [129, 130, 194]. As we review related work, we will point out some of the shortcomings of prior work when their fault analysis methods are applied to on-chip edge NNs. We then describe how FKeras addresses these shortcomings.

It is well known that NNs have many redundant parameters [76, 83]. Faults in different weights are expressed differently, especially depending on the size of the NN, as our results later show (see Fig. 5.11). Furthermore, the bit position of the weights is important—within a weight, there is a significant range of effects for the different bits. Our results show that the most significant bits are the most sensitive, and the least significant bits are fairly insensitive to faults, as confirmed by prior work [40]. To optimally compress quantized NNs, it is crucial to have a *bit-level* understanding of faults, especially when we want to run an edge NN entirely on chip. Thus, we develop FKeras to quantify the fault tolerance of an hls4ml edge NN for on-chip inference.

Tab. 5.1 compares FKeras to prior work. It categorizes FI tools and methods based on if a tool targets single or multiple bits or randomly selects bits to fault inject (Bit Target-ability), if a tool supports different data types for quantized NNs (Quantization Support), if a tool/method presents a way to speedup a FI campaign (FI Speedup Method), and if a tool/method introduces any sensitivity metrics (Sensitivity Metrics). Next, we elaborate further on prior work.

5.2.1 Assessing NN Resilience

Researchers have studied the resilience of NNs to identify silent data corruptions (SDCs)– soft errors that lead to incorrect NN output [111, 185]. They take three primary approaches: fault injection (FI) campaigns, heuristics, and ML modeling [129, 135, 185].

FI campaigns simulate a fault either in software [40, 111, 126, 130, 142, 145, 159] or hardware [69, 159] and run a number of inputs through the NN to assess performance. Such campaigns are computationally expensive because they must simulate an enormous amount of faulty scenarios, especially when considering the thousands to millions of parameters and operations present in an NN [185]. Researchers have developed methods to accelerate fault injection campaigns [40, 69, 129]; however, many of their evaluations are often limited to NNs that rely on off-chip memory accesses, meaning they assume the weights are fully protected by error correction codes (ECC) or parity and thus only study the effects of faults to NN activations [15, 40, 113]. FKeras instead focuses on a fault model that targets NN weights that fully reside on chip, where ECC or parity bits require extra resources, which is costly when most resources are preciously devoted to implementing the edge NN.

Researchers have also developed heuristics that measure NN sensitivity through characteristics such as the gradient [44, 127] or how errors propagate through a model [170]. Since heuristics are naturally lossy, they are less accurate than FI campaigns, but significantly cheaper to run. They trade some accuracy for speedup.

Researchers have further developed ML models to predict which bits in a NN are mostly likely to induce faulty output [36, 186, 187, 215]. These methods extract salient features of NNs relevant to fault analysis and train a machine learning model to predict the faulty parts of the NN on a relatively small amount of ground truth data. While these models are fairly accurate, they are expensive to scale up because they need to train a new ML model for each new NN they want to evaluate.

Researchers have created a fault injection method called BinFI [40], which relies on the

monotonicity of the bit order, i.e., higher order bits are at least as sensitive as lower order bits, to reduce the number of faults to inject by performing a binary search within each NN output. This is an effective approach because they reduce the search space by $O(\log(n))$ per NN output where n is the number of bits representing the output. However, our results show that this assumption of monotonicity leads to false negatives, as seen in our results in Sec. 5.4.2.

Researchers have looked into statistical fault injection [162], which we call "StatFI", to reduce the FI search space of a NN's weights. However, StatFI is primarily effective in reducing the search space because it relies on how flipping bits in float32 format leads to large changes in the magnitude of the weights, as previously discussed in Sec. 5.1. Based on this information, StatFI selects a subset of the bits in each bit position, e.g., MSB, in each layer to randomly flip. These large changes in magnitude do not occur with low-precision fixed-point data types because the range represented is considerably smaller. Our results show that StatFI fails to identify many sensitive bits. It is not as effective as any of FKeras's sensitivity metrics when considering low-precision quantized NNs for on-chip inference. This is because StatFI is selecting which bits to flip in each bit position in each layer *randomly*, whereas FKeras's metrics are selecting which bits to flip using the Hessian for instance, which captures how sensitive a given NN's parameter is to faults (see Fig. 5.8). We present a more detailed comparison later in Sec. 5.4.2.

In response to these shortcomings, FKeras provides a bit-level ranking using the Hessian matrix of second partial derivatives of the loss function to identify the most crucial weights and bits. The Hessian reveals how sensitive a weight is to faults. As a result, FKeras's Hessian-based bit-level ranking uses fine-grained information on a NN's sensitivity, rather than more coarse-grained approaches that rely solely on bit order monotonicity or statistical sampling that do not translate as well to quantized, on-chip edge NNs. Even more, FKeras presents the opportunity to combine the FI campaign with a sensitivity metric and guide the campaign to search for more sensitive bits first. FKeras combines the monotonicity of the bit order with its Hessian sensitivity metric to rank the most significant bits based on its Hessian sensitivity score first before doing the same for each lower order bit. Our results show this bit-level Hessian metric can identify

the most sensitive parameters with high accuracy, and thus serves as a valuable guide for FI campaigns.

5.2.2 Optimizing NN Resilience

Prior work has taken several approaches to protect NNs and optimize said protection. Some proactive approaches, such as fault-aware training [22, 149, 150, 174, 213], prevent faults by training the weights themselves to be more resilient. Other approaches are more reactive, including DMR and TMR [81, 179], selective DMR/TMR [22, 105, 117, 129], and activation clipping [39, 91]. Activation clipping is an attractive and inexpensive option. It involves profiling the model to capture what the numerical range of the activations should be and then clipping an activation if it falls outside the range due to faulty hardware. It is particularly effective at protecting float32 NNs as the range of float32 values is very large. However, this is less effective in quantized models because quantized data types, like int8, naturally clip the activations [129]. hls4ml networks are usually heavily quantized, making activation clipping not an option. Prior work [22, 129] also combines a number of these techniques to improve error coverage while minimizing protection overhead costs.

FKeras can help determine how to best protect NN weights, which is especially important for edge NNs that run fully on-chip. As we show in our experiments (Sec. 5.4), certain bits are much more sensitive to faults than others. Protecting only those most sensitive bits can reduce the overhead of fault-tolerant mitigation. For example, our results show that it is possible to selectively perform TMR on the ECON-T weights with minimal effect on the encoder's resilience to faults. We can use the FKeras sensitivity metrics to find the most important bits to protect, which is especially valuable when resources are precious for on-chip inference.

5.3 FKeras

NNs are often over-parameterized, and not all weights are equally important [76, 83], which indicates that some weights are more sensitive to faults than others. FKeras is a codesign

tool for designing fault-tolerant NNs in hls4ml. It provides a *sensitivity score* that ranks the NN parameters based on their sensitivity to faults and supports modeling single- and multiple-bit fault injection campaigns on NN weights. FKeras can use the sensitivity score to speed up fault injection campaigns by quickly and accurately identifying the most important NN parameters. FKeras is also valuable for NN co-design problems for applications that require fault tolerance.

5.3.1 NN Sensitivity Scores

To analyze NN sensitivity, we want to understand how a NN performs under faulty conditions, e.g., bit flips in the weights. Previously, researchers have used the gradient as a metric to capture a NN's resilience to faults [44, 127]. A NN's gradient with respect to the parameters is a vector of size n, defined as

$$\frac{\partial L}{\partial \theta} \in \mathbb{R}^n \tag{5.1}$$

where L is the NN's loss function and θ represents the n parameters of the NN. The gradient provides information about the steepness of the loss function.

The Hessian matrix H describes the steepness and curvature, providing additional insight into the NN behavior. It is an $n \times n$ matrix of the second-order partial derivatives of the loss L:

$$H = \frac{\partial^2 L}{\partial \theta^2} \in \mathbb{R}^{n \times n} \tag{5.2}$$

The Hessian captures the local curvature of the loss function, as it shows the rate at which the gradient changes. The local curvature of the loss reflects the sensitivity of a NN's parameters [56, 208]. A steep curvature around a given parameter indicates that it is highly susceptible to noise. Perturbing this parameter even slightly will result in significant changes to the loss, implying that the model will behave worse and lead to incorrect output. Conversely, a relatively flat curvature around a given parameter indicates that it is insensitive to noise. Small perturbations to it will result in minimal changes to the loss, i.e., the model's behavior remains about the same. Since the Hessian models parameter sensitivity, researchers have relied on it to successfully quantize

NNs to mixed precision [32, 56].

Despite how valuable the Hessian is, it is not commonly used because of the misconception that computing Hessian information for a large NN is infeasible, given that it requires $\mathcal{O}(n^2)$ memory [208]. However, extracting the Hessian eigenvalues and eigenvectors takes $\mathcal{O}(n)$ memory in $\mathcal{O}(n)$ time using techniques from randomized numerical linear algebra (RandNLA) [16, 58, 131, 189]. The eigenvectors and eigenvalues capture the relevant Hessian information.

FKeras provides a Hessian-based sensitivity score for each bit of every NN weight. The sensitivity score provides a quick and accurate method to assess the fault tolerance of an NN. This allows us to speed up fault injection campaigns and perform codesign considering fault-tolerance as a constraint.

FKeras uses the power iteration method to compute the top k eigenvalues and eigenvectors of the Hessian in $\mathcal{O}(n)$ time, where n is the number of parameters [208]. Based on these k eigenvalues, we compute a parameter score:

$$H^{'} = \sum_{i=1}^{k} \lambda_i (v_i \cdot \theta) v_i \in \mathbb{R}^n \tag{5.3}$$

where λ_i is the ith eigenvalue, v_i is the ith eigenvector, and θ is a vector representing the model parameters (of which there are n). The parameter i sensitivity score aims to identify which parameters contribute most significantly to the Hessian by weighting it by the eigenvalue along the most sensitive direction (eigenvector). Fig. 5.1 visualizes how we compute our Hessian score.

We sort the parameters' most significant bits (MSBs) by the parameter sensitivity score to get a bit-wise ranking. Then, we do the same for the parameters' ith MSB until we reach the least significant bit (LSB). We sort from MSB to LSB, where we consider MSBs to be the most sensitive bits because they cause the most significant perturbation in the weights of the NN when flipped (based on a twos-complement representation).

FKeras also provides a sensitivity score based on the gradient. This works in a similar

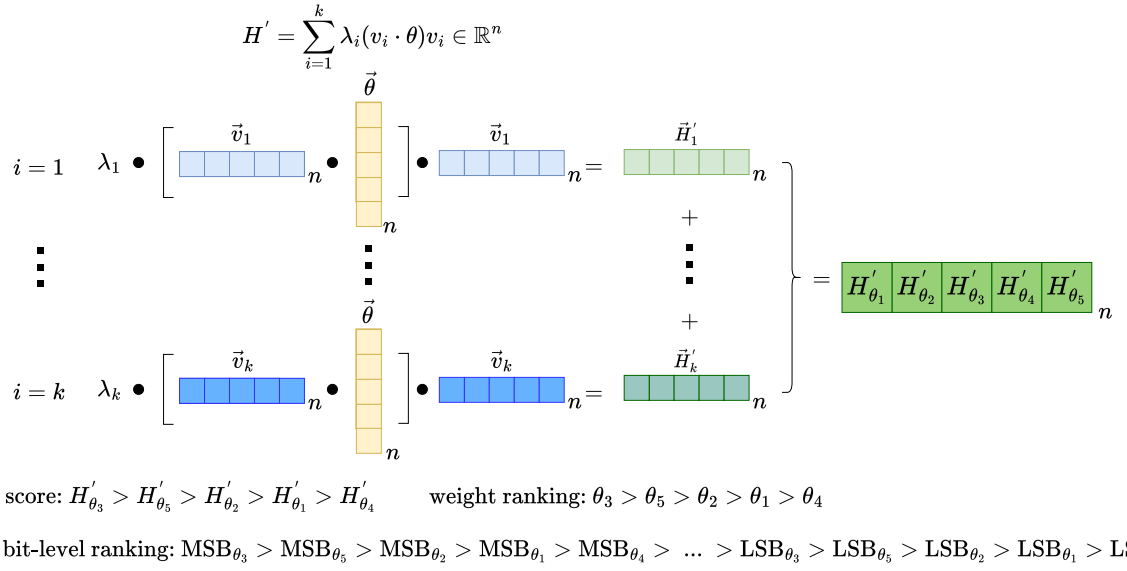


Figure 5.1. How to compute our Hessian sensitivity score. We compute our Hessian sensitivity score using the top k eigenvalues and eigenvectors of the Hessian, where λ_i is the ith eigenvalue, $\vec{v}_i$ is the ith eigenvector, and $\vec{\theta}$ is a vector representing the model parameters (of which there are n). For instance, for $i = 1$, we multiply the top-1 eigenvalue λ_1 with the dot product of top-1 eigenvector $\vec{v}_1$ and the parameters $\vec{\theta}$. We then multiply this constant result element-wise with the top-1 eigenvector v_1 to get the Hessian score $\vec{H}_1'$ based on the top-1 eigenvalue and eigenvector. We then sum these top-k Hessian score vectors together to get the final Hessian score $H' \in \mathbb{R}^n$ such that H'_{θ_i} is the Hessian score for the ith parameter θ_i. We rank our weights based on this score, where a higher score means the weight has higher sensitivity to faults. Then we rank all of the MSB's by the weight sensitivity score, then the second MSB's, and so on to form our Hessian bit-level ranking.

manner as the Hessian, but instead uses the gradient value from Equation 5.1 and sorts the bits from MSB to LSB in a similar manner as the Hessian.

FKeras can compute the Hessian trace in $O(n)$ time, where n is the number of parameters, using the Hutchinson method [208]. FKeras provides the trace per layer so the user can compare the layer sensitivity. A higher trace implies that a layer is more sensitive to faults and other weight perturbations.

5.3.2 Fault Model

Different environments have different fault rates. For example, at the LHC, high energy physicists expect a fault to occur in their NN hardware every 15 seconds whereas in data centers, system administrators expect faults to occur once every few thousand events [53, 92]. Thus, we want to model these fault rates using a *fault model*, which describes how often a bit flip occurs.

A designer can use FKeras to perform experiments on two kinds of fault models: the single-bit flip model and the multi-bit flip model.

Single-Bit Flip Model

A common fault model is the single-bit flip model [69, 111, 129], which represents the case when only one bit flips at a time. The single-bit flip fault model can be applied to NN weights, activations, both, or at an even finer grain [88]. We limit our scope to only the weights of a NN, as motivated by the hardware implementations of on-chip inference for edge NNs. In particular, our fault model is motivated by the ECON-T's ASIC hardware implementation, as seen in Fig. 5.2. In this diagram, we see that the entire NN is implemented fully on chip, and inference completes in a single cycle. Faults in the activations only affect a single inference, whereas faults in the weights will affect billions of inferences before the weights are refreshed every four hours [81]. Thus, we focus our fault model on bit flips in the NN weights.

Multi-Bit Flip Model

We also consider a multi-bit flip fault model to represent high radiation conditions. This is motivated by our running example—the ECON-T autoencoder ASIC operating in the LHC HGCal [81], as seen in Fig. 5.2.

The multi-bit flip model for the HGCal expects $\sim$0.06 bit flips per ECON-T per second, or one bit-flip every 15 seconds for an individual ECON-T chip. Particles go through the HGCal detector at a rate of 10^8 particles/cm^2/second [70]. However, not every particle will flip a bit. The rate at which any single ECON-T flip-flop will get hit depends on a number of factors,

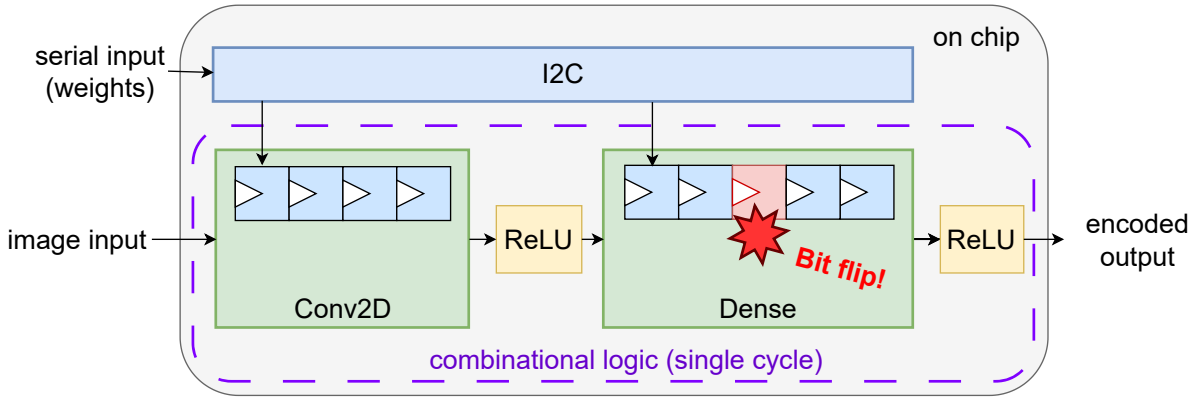


Figure 5.2. ECON-T ASIC hardware design and fault model. The ECON-T ASIC is designed to run inference completely on chip in a single cycle. The weights are thus the most vulnerable to bit flips caused by high-energy particle strikes because they persist over billions of cycles before getting refreshed every four hours; whereas, the activations last a single cycle only.

such as the material (silicon) and the ASIC technology node (65 nm). Moreover, the ECON-T's weights are refreshed every four hours. Thus, we define our multi-bit flip model to expect 960 bit flips when there is no fault protection in between the weight refreshes.

5.4 Experimental Evaluation

This section describes the experimental setup and results.

5.4.1 Experimental Setup

We demonstrate how FKeras can efficiently analyze a NN's fault sensitivity by performing experiments on CIFAR-10 [107] and an HGCal dataset. CIFAR-10 is a popular image classification dataset. The HGCal dataset contains vectors of high-energy particle collision sensor data. We use FKeras to understand the fault tolerance of four different models: (1) an edge CNN trained on CIFAR-10, specifically hls4ml's submission to the MLPerf Tiny Inference Benchmark [27], (2) a medium ECON-T NN, (3) a large ECON-T NN, and (4) a small ECON-T NN.

The three Pareto-optimal ECON-T models (Small, Medium, and Large) represent trade-offs between model accuracy and size. All ECON-T models were trained on the HGCal dataset.

We evaluate model performance using Earth mover's distance (EMD), a distance measure between two probability distributions [160]. In our case, the EMD measures the distance between the encoder's input energy readings and the decoder's outputs, respectively. Lower EMD is better and an EMD of 0 indicates the autoencoder is lossless. The three models were found using a Bayesian optimization neural architecture search. We used HAWQ-V2 [55] to quantize the large and small models (the medium model was hand-tuned from prior work). Fig. 5.3 provides details on the NN topology and quantization for the three ECON-T NNs.

ECON-T Medium is the model described in the paper by Di Guglielmo et al [81] - a 2D convolution layer followed by a dense layer using a 6-bit arbitrary precision fixed point data type. It balances between accuracy and model complexity. ECON-T Medium has 2 120 weights (180 for the convolution and 1 940 for the dense layer) for a total of 12 720 weight bits.

ECON-T Large has the same two-layer structure but larger convolution and dense layers. ECON-T Large uses a 5-bit arbitrary precision fixed point data type in the convolution layer and a 7-bit arbitrary precision fixed point data type in the dense layer. ECON-T Large has 800 weights in the convolution layer, 8 192 weights in the dense layer for 61 344 total weight bits.

ECON-T Small has two dense layers both using an 8-bit arbitrary precision fixed point data type. It has 1 280 weights and 10 240 total weight bits. The first dense layer is 64×16 and the second is 16×16. There are 10 240 total weight bits.

The final benchmark is the hls4ml CIFAR-10 submission to the MLPerf Tiny Inference benchmark [27]. It uses a two-stack model with no skip connections (five convolutional layers with 32, 4, 32, 32 and 4 filters, kernel size of 1, 4, 4, 4, and 4, and strides of 1, 1, 1, 4, 1, respectively). It achieves an accuracy of 83.1%.

We used FKeras to perform the single- and multi-bit flip fault injection campaigns. We conduct all of the fault injection campaigns on Google Cloud's `c3-highcpu-88` compute engine which utilizes an Intel Sapphire Rapids with 88 vCPU, 44 cores and 176GB of memory. We first generate oracles for the single-bit fault models by exhaustively performing single-bit flips on the weights and determining their effect. We create the single-bit flip oracle for CIFAR-10 by

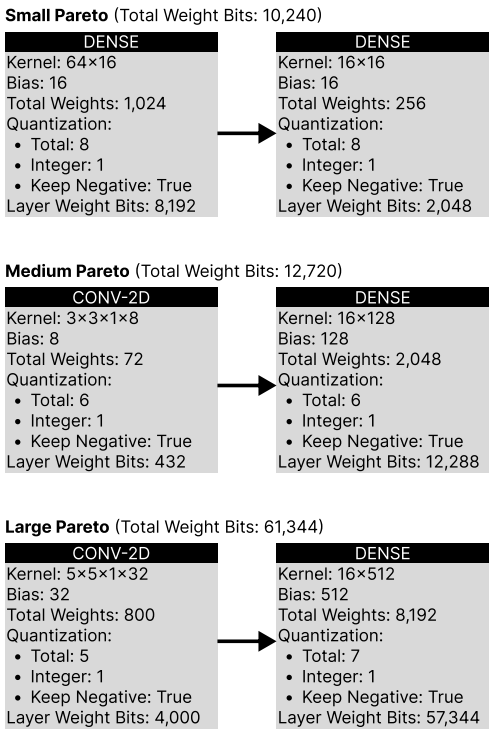


Figure 5.3. A layer-by-layer overview of the three ECON-T models. Each box represents a layer in the given model. At the top of a box, we list what kind of layer it is, e.g., Dense or Conv2D. Within a box, we list details about each layer, namely the size of the kernel, the number of biases, the total number of weights, and quantization information on how each weight is quantized to fixed point according to QKeras. For example, a quantization scheme of `Total: 8, Integer: 1, Keep Negative: True` means there are a total of 8 bits, one of which represents the integer portion. When `Keep Negative` is `True`, the value is signed, and this sign bit is stolen from the fractional portion. Thus, out of a total of 8 bits, one bit represents the sign bit, one bit represents the integer part, and the remaining six bits represent the fractional part.

flipping a parameter bit and evaluating the model on 8 313 test images. This is a subset of the 10 000 images provided by the test dataset. This subset only includes the images that the CNN correctly classifies under non-faulty conditions. If flipping a bit causes the model to mispredict an image, we classify that bit as sensitive. To generate the single-bit flip oracle for the HGCal dataset, we flip a bit and evaluate the model on 20 000 validation inputs. We classify a bit as sensitive if flipping it causes the model error to exceed the average non-faulty model error.

We perform our multi-bit flip experiments on the ECON-T Medium model. The extremely high-radiation environment of the HGCal makes the ECON-T models subject to multi-bit flips.

We flip 960 random parameter bits and evaluate ECON-T Medium on 20 000 validation inputs. We perform this process 14 000 times for the baseline ECON-T model, which corresponds to a 98% confidence level with 1% confidence intervals. This 14 000 sample size was determined based on a statistical model defined by [110].

5.4.2 Single Bit Flip Results

ECON-T Medium Model Resilience

We perform the first set of experiments on the ECON-T Medium Pareto autoencoder. Fig. 5.4 shows the fault sensitivity of the encoder's parameter bits. The first 180 weights correspond to the encoder's convolutional layer, and the remaining weights correspond to the encoder's linear layer. The bit index is the index of the bit that was flipped, where bit 0 is the sign bit, bit 1 is the integer bit, and bits 2–5 are the fractional bits. As expected, the sign bit and integer bit create the largest changes in the magnitude of the parameters, so those bit flips lead to higher EMDs. The non-faulty model has a non-zero EMD whose value is 1.10. Overall, 63.5% of the bits exceed the baseline EMD.

Not surprisingly, the largest EMD values, corresponding to most faults, occur when faults are induced on the most significant bits. The MSB (Bit 0) visually has higher EMD values than the other bits. The LSB (Bit 5) barely has any visibly discernible change from the baseline EMD value. This is not surprising, and this monotonicity has been used previously to guide fault injection campaigns [40].

The first 180 weights correspond to the convolutional layer; the remaining 1940 weights are for the dense layer. Faults in the convolutional weights generally lead to more errors than faults in the dense layer. This is especially visible in the MSB. There are a lot of weights in the dense layer where a fault does not induce any additional error, and some in the convolutional layer. The variability of the EMD across bits of the same significance can vary greatly. For example, many of the dense layer Bit 0 weights have high EMD, but many have low EMD.

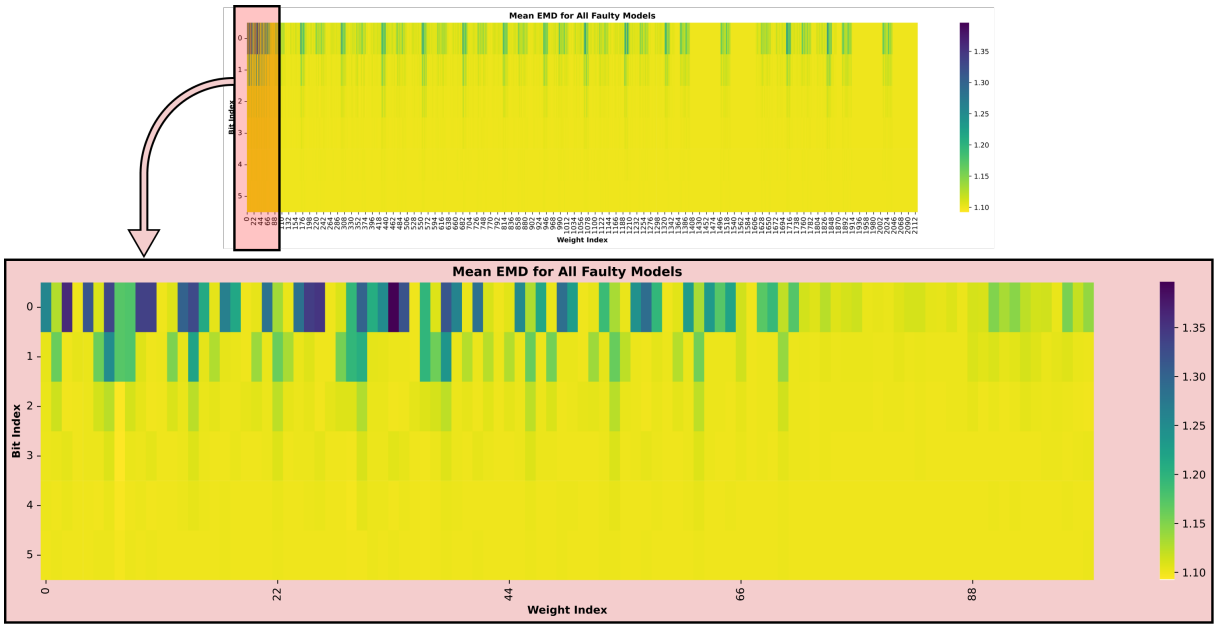


Figure 5.4. The average EMD for the ECON-T medium Pareto NN under a single-bit fault model. The x-axis corresponds to the NN weight. The y-axis represents the bit index of the weight where 0 is the MSB and 5 is the LSB. Note that the top heatmap reports the average EMD over all weight bits whereas the bottom heatmap only reports the average EMD for the first 600 weight bits (NN weights 0 - 100).

Sensitivity Ranking Accuracy

Our second set of experiments evaluates the ability of the Hessian and gradient to provide a bit-level sensitivity ranking. In particular, we aim to understand how well the model error (e.g., EMD) is captured by the Hessian and gradient sensitivity metrics from Sec. 5.3.1. We calculate these two bit-level metrics for the ECON-T Medium Pareto model and compare them with the single-bit fault EMD values from the experiments in the previous section.

Fig. 5.5 compares the EMD values versus the Hessian and gradient bit-level sensitivity ranking values. We separate out the rankings by bit, i.e., the first column is the MSB, and the sixth column is the LSB. These match the bit indices from Fig. 5.4. The first row plots the EMD (y-axis) against the Hessian ranking (x-axis) for each of the bits.

Generally speaking, we want a metric that ranks the weights that cause little change in EMD lower than those that have a higher change. Consider the Hessian MSB (bit 0) shown in

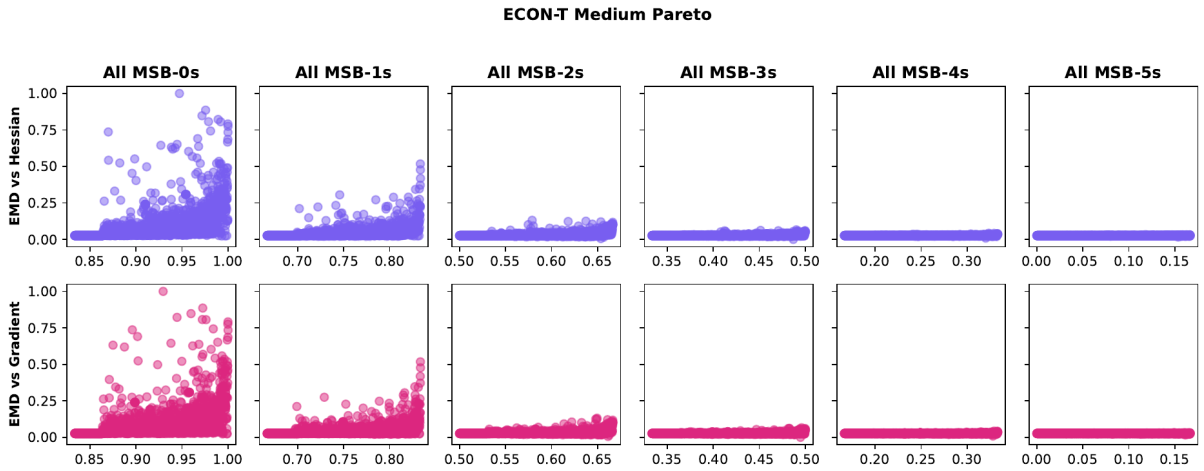


Figure 5.5. A comparison of the normalized EMD (y-axis) versus the normalized Hessian and gradient sensitivity metrics (x-axis). The first column is the most significant bit and is ordered to the least significant bit in the sixth column.

the upper left plot. The EMD values of the lowest-ranked bits ranked are generally very small. After that, the EMD generally increases though there are certainly outliers. The Spearman's rank correlation coefficient $\rho = 0.508$ shows a high correlation between the Hessian ranking and the EMD. In the Hessian LSB (upper right figure), almost all bits have a very small EMD; thus, the ranking is somewhat arbitrary ($\rho = 0.098$). The major takeaway is that the more significant bits matter more. and the errors in the least significant bits have relatively little effect on this model. The first three significant bits cause the most faults, whereas faults in the least three significant bits induce little error.

The second row plots the EMD against the gradient sensitivity ranking. For this model, the Hessian performs better than the gradient in the most significant bits. The performance of the least significant bits is less important since they induce few errors.

The Hessian and gradient are measures of the loss landscape that provide valuable insight into how the NN behaves with respect to the different weights. The Hessian and gradient provide a metric to compare different weights. They do not have the ability to rank individual *bits* across those weights. Our Hessian and gradient metrics rank the most significant bits highest (Bit 0), followed by the second most significant bit (Bit 1), all the way down to the least significant bit

(Bit 6 in this case). This is not ideal, and a better bit-level ranking can be achieved by mixing bits of different significance. This is clearly shown in Fig. 5.5. Some bit 1 EMD values are higher than the bit 0 values in the ECON-T Medium model. Some bit 2 EMD values are higher than bit 1 and bit 0, and so on. Those bit 1 variables should be ranked higher by the sensitivity metric, and both of our metrics do not allow for this. We believe that a similar approach to BinFI [40], which attempts to find the inflection point within a weight, could lead to a productive metric. We leave that as future work. However, the ranking of MSB to LSB works well as a first-order approximation. We compare the benefits in the next set of experiments.

Sensitivity Metric Comparison

Our next set of experiments aims to understand how different metrics perform at identifying the bits most sensitive to single-bit faults. We use four different models—three different ECON-T autoencoder models and a CIFAR-10 edge CNN, specifically hls4ml's submission to the MLPerf Tiny Inference Benchmark [27]. We compare the abilities of four metrics to rank the weights: *random*, *MSB to LSB*, *Hessian*, and *gradient*. *Random* picks a bit at random. *MSB to LSB* selects the most significant bits first, followed by the second most significant bit, all the way to the least significant bits. The bits are selected in the weight index provided by Keras after flattening a layer's weight matrix, e.g., the ECON-T Medium NN has the weight ordering shown in Figure 5.4. *Hessian* uses the Hessian-based sensitivity score as computed in Equation 5.3. *Gradient* uses the parameter's gradient value from Equation 5.1 and sorts the bits from MSB to LSB in a similar manner to *Hessian* (see Section 5.3.1).

We compare the ability of these metrics to predict the sensitivity accurately. Fig. 5.6 shows the Spearman's rank correlation coefficient ρ for the four different models. The results compare the four metrics and an *Oracle* (a perfect ranking derived from the brute-force single-fault experiments) to predict the bit sensitivity. The ρ values are shown for all the bits, followed by them being broken out into the individual bits in order of most significant to least significant. As expected, the random ranking has a near zero correlation, and the oracle has a perfect

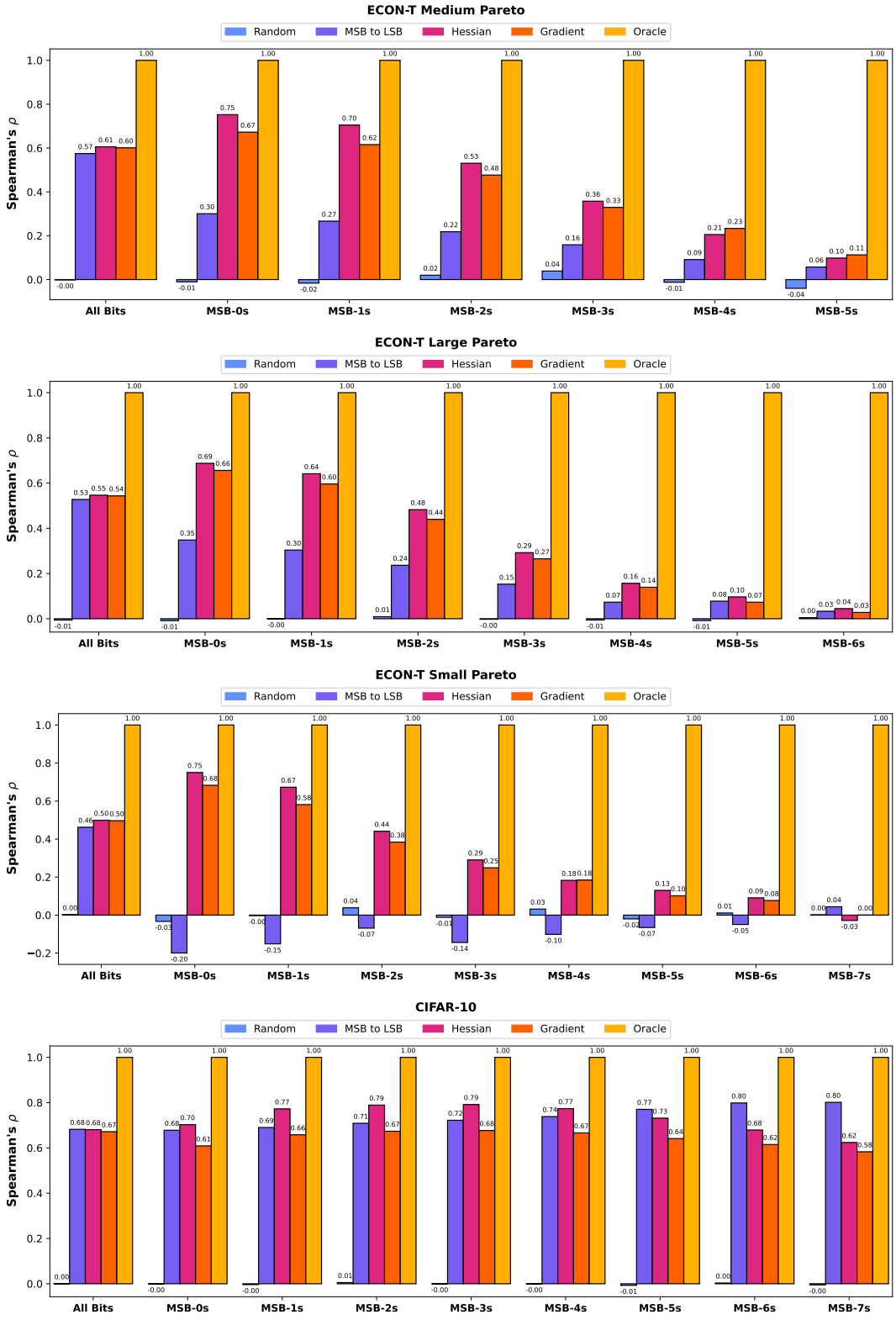


Figure 5.6. The ability of four sensitivity metrics (random, MSB to LSB, Hessian, and gradient) to rank the bits whose faults induce the most errors. Spearman's rank correlation coefficient ρ is provided for all the bits in the first column and then broken down by individual bits from most to least significant.

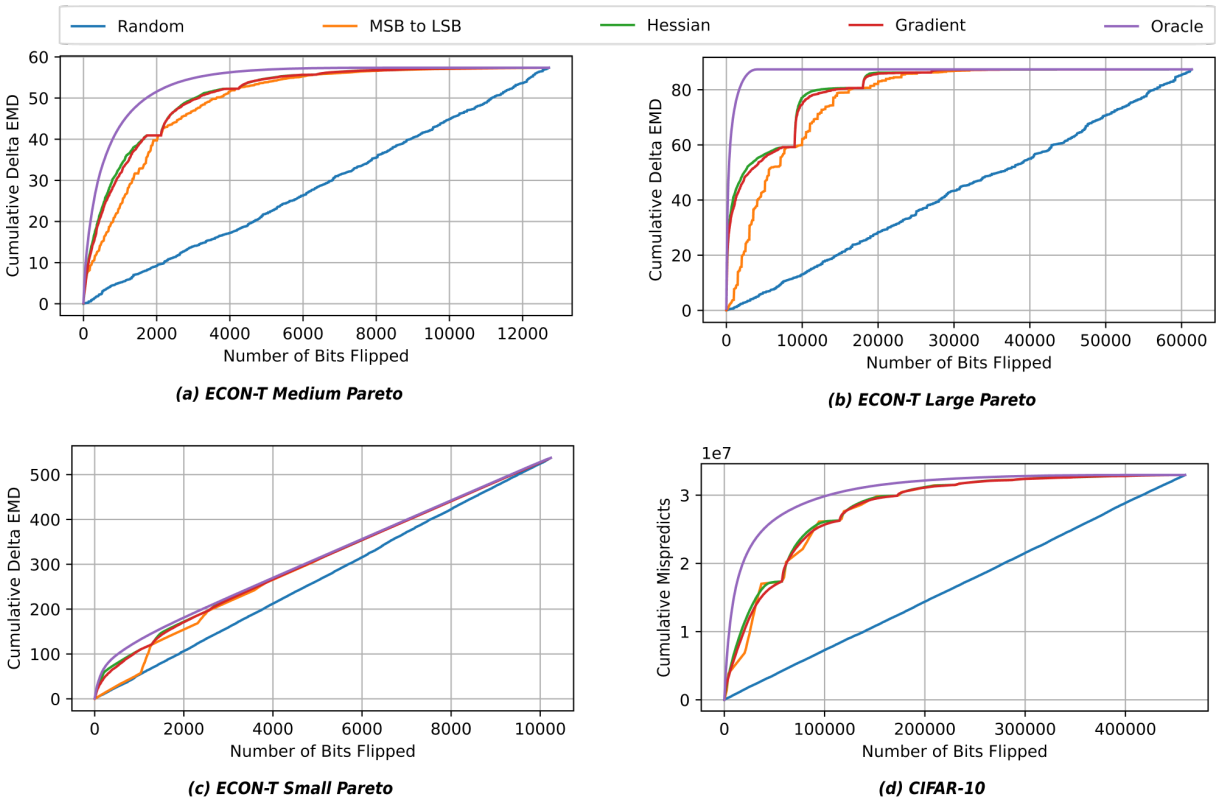


Figure 5.7. The magnitude of the error vs. the number of bits flipped for the four NN models. The three ECON-T models use cumulative ΔEMD, where $\Delta\text{EMD} = |\text{EMD}_{\text{bitflip}} - \text{EMD}_{\text{non-faulty}}|$ as the error measure and the CIFAR-10 CNN uses the cumulative number of mispredicts. In both cases, larger indicates more errors. Each model plots five ranking metrics which attempt to order the NN weight bits from those to contribute most to error to those that contribute little or nothing to the error.

correlation across all the models. The Hessian is consistently the best, especially in the more significant bits. The gradient also provides good performance close to the Hessian but clearly lower in most cases. The MSB to LSB ranking performs quite well and provides a relatively simple way to consider bit sensitivity, e.g., for a fault injection campaign.

Next, we consider the relative magnitude of the error and not just the relative ranking. Fig. 5.7 shows the results of an experiment that plots the cumulative error when ranking the sensitivity of the weight bits. A larger error indicates that flipping that particular bit increases the error of the overall model. The error measure depends on the model. The three ECON-T autoencoder NNs use EMD for error. Recall that EMD is a measure of error where larger indicates

worse autoencoder performance. The CIFAR-10 CNN uses the number of mispredictions for the error where larger indicates more error.

Consider first Fig. 5.7a that plots the cumulative ΔEMD versus the number of bits flipped for the four metrics and an oracle on the ECON-T Medium Pareto NN. The oracle is the optimal or best-case ranking calculated from the brute-force single-fault experiments (e.g., from Fig. 5.4 for ECON-T medium). The oracle ranks the bits with the largest mean ΔEMD first. The cumulative EMD provides the difference between the faulty model EMD and the EMD of a model with no faults. The EMD for the non-faulty model is 1.100. The cumulative ΔEMD for the oracle results quickly approaches the maximum cumulative ΔEMD of 57.37. Only 63.5% (8 080/12 720) of the bits are sensitive, i.e., they have a nonzero ΔEMD. The remaining 36.5% do not affect the autoencoder EMD.

The *random* metric is the worst of the metrics showing that chance alone provides roughly an equal chance of guessing the bits that contribute most to the EMD. *MSB to LSB* performs significantly better than random. This shows that the bit order matters. The most significant bit has the lion's share of the cumulative ΔEMD (40.91 of the 57.37). The impact on ΔEMD falls quickly; weights from last few significant have little effect on the EMD. *Hessian* and *gradient* both perform better. *Hessian* does perform better at ordering the MSB weights with *Hessian* being slightly better as indicated by the separation between the two lines. In particular, *Hessian* is more accurate for the first 2 120 bits (corresponding to the weights of the most significant bit). The subsequent bits are approximately equal between *Hessian* and *Gradient*. These bits contribute less to the overall EMD and thus are overall less sensitive.

It is interesting to compare the difference between the *random* and *oracle* on the three ECON-T NNs. ECON-T Small NN (Fig. 5.7c) has a much smaller spread due to the fact that the model is smaller and all of the weights are more sensitive. Conversely, the spread in the large ECON-T NN (Fig. 5.7b) is the largest of the three. The large model has a small percentage of sensitive weights as indicated by the steep initial slope of the *Oracle*. In other words, the vast majority of the weights are insensitive to faults, which is not surprising given that the model has

many more weight bits. The ECON-T large NN has 61 344 weight bits compared to ECON-T medium (12 720 weight bits) and the ECON-T small (10 240 weight bits).

CIFAR-10 is a different classification problem with a different error measure. Thus, the results are not as easily comparable as the three ECON-T NNs. Overall, *CIFAR-10* is the largest model with 459 520 weight bits. The fairly steep initial slope of the *Oracle* indicates that most of the sensitivity resides in a small number of bits. However, there is a relatively long tail, e.g., more similar to ECON-T medium Pareto NN. The relatively large separation between the *Random* and *Oracle* indicates that the bit sensitivity is not easy to predict. *Hessian* generally performs best in determining the most sensitive bits.

In Fig. 5.8, we compare the cumulative ΔEMD and mispredicts with state-of-the-art work in fault injection: BinFI [40] and StatFI [162]. To recap Sec. 5.2, BinFI performs a binary search within each value in the NN to find the bit that is the "inflection point," wherein all the bits more significant than it are considered sensitive to faults. BinFI calls this the *silent data corruption (SDC) boundary*. While BinFI applies this technique to the NN activations, we instead apply it to the NN weights according to our fault model. Since BinFI performs a binary search to find the SDC boundary, we first plot the actual bits BinFI flips during the binary search, which we call *BinFI (Actual Bits Flipped)*. The SDC boundary implies that all the bits more significant than it are also sensitive. We plot the actual bits flipped plus these implied sensitive bits as *BinFI (Actual+Implied Bits Flipped)*. BinFI does not specify the order in which to search the NN values, so we flip them in the order they appear in the NN. StatFI introduces two fault injection methods for finding the sensitive NN weight bits: data-aware and data-unaware. StatFI statistically determines the sample size of how many bits to flip per weight bit index, e.g., the MSB or second MSB, per layer. *StatFI (Data-aware)* statically measures the changes in magnitude in each weight that occur from flipping a bit to determine the sample size per weight bit index per layer. The larger the magnitude change the higher the sample size will be and vice versa. *StatFI (Data-unaware)* does not rely on changes in magnitude and selects the same sample size per weight bit index per layer. StatFI randomly sample based on the determined sample

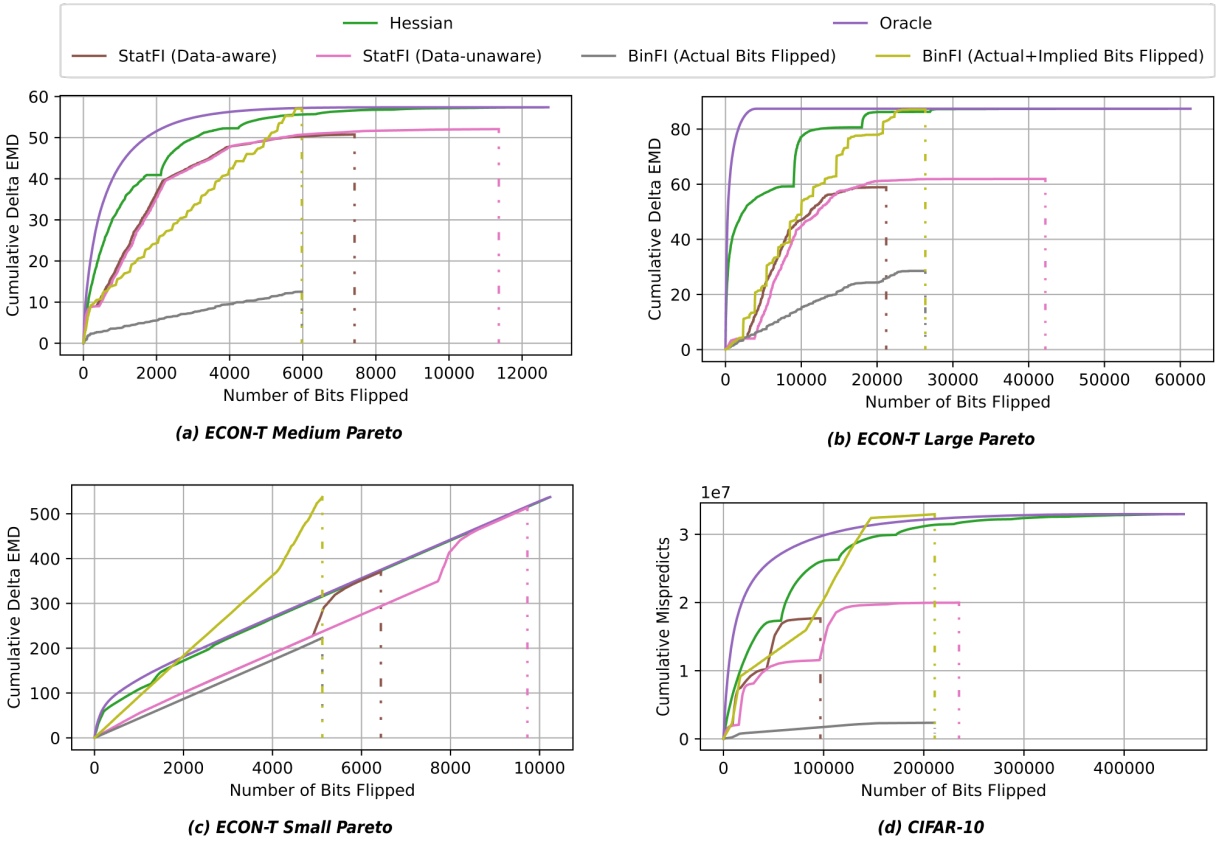


Figure 5.8. Comparing the magnitude of the error vs. the number of bits flipped for the four NN models with state-of-the-art fault injection methods BinFI [40] and StatFI [162]. The three ECON-T models use cumulative ΔEMD as the error measure and the CIFAR-10 CNN uses the cumulative number of mispredicts. In both cases, larger indicates more errors. Each model plots six ranking metrics which attempt to order the NN weight bits from those to contribute most to error to those that contribute little or nothing to the error. In particular, we compare our own Hessian metric to BinFI's and StatFI's bit flips.

size, i.e., they do not specify the order in which to flip bits. We thus order them MSB to LSB, as StatFI provides a list of bits to flip per MSB, second MSB, etc.

Let us first consider Fig. 5.8a and look at how BinFI compares with the Hessian and the Oracle on the ECON-T Medium Pareto NN. We find that *BinFI (Actual Bits Flipped)* is not very impressive, as expected. BinFI uses each of these bit flips to implicate more bits. As such, *BinFI (Actual+Implied Bits Flipped)* performs more impressively in the beginning; however, it begins to falter after a few hundred bit flips. This is expected because BinFI does not specify an order in which to search the weights, whereas the *Hessian* does. By only flipping around half of all of the NN weight bits though, *BinFI (Actual+Implied Bits Flipped)* identifies most of the sensitive bits, as indicated by the dashed vertical line that drops down after $\sim$6 000 bits flipped. It fails to identify 5% of the sensitive bits (676 out of a total of 8 080). Comparing *BinFI (Actual+Implied Bits Flipped)* to our work, the *Hessian* finds more sensitive bits much sooner in its search. In fact, for the first $\sim$5 500 bit flips, the *Hessian*'s bit flips are much more informative, as it finds the highly sensitive bits early on. Thus from Fig. 5.8a as well as Fig. 5.8b, we observe this tradeoff of the *Hessian* finding more sensitive bits earlier in the search versus the *BinFI (Actual+Implied Bits Flipped)* finding a majority of the sensitive bits earlier for the ECON-T Medium and Large Pareto models.

Fig. 5.8c and Fig. 5.8d show more complex trends. In both the ECON-T Small Pareto (Fig. 5.8c) and CIFAR-10 (Fig. 5.8d), we see that *BinFI (Actual+Implied Bits Flipped)* rises slowly, as we have seen previously, before exceeding the *oracle*. This happens because *BinFI (Actual+Implied Bits Flipped)* implies multiple bits are sensitive per bit flip whereas the *oracle* only implicates one bit based on how we plot it. Clearly, the *oracle* knows all of the sensitive bits prior to fault injection (and could reach the maximum cumulative ΔEMD/mispredicts with 0 bit flips); however, we are interested in understanding the ceiling for the*Hessian* in our *oracle*. As such, *BinFI (Actual+Implied Bits Flipped)* exceeds the best the Hessian could perform for these two models. This is likely the case because most of the bits in the ECON-T Small Pareto and CIFAR-10 models are sensitive. 100% of the ECON-T Small Pareto bits and 82.72% of the

CIFAR-10 bits are sensitive. Thus, these NNs are easier tasks for BinFI—each bit flip is highly likely to find a sensitive bit. We are not necessarily finding a needle in a haystack the way we are for ECON-T Large, where only 6.5% of the bits are sensitive. The *Hessian* is clearly better in this case (Fig. 5.8b).

However, there is a major caveat with BinFI: the only information we receive on how sensitive the model bits are is from *BinFI (Actual Bits Flipped)*. As seen in all four charts in Fig. 5.8, *BinFI (Actual Bits Flipped)* reveals very little information and has the lowest cumulative ΔEMD/mispredicts out of all of the methods. We have no way of determining cumulative ΔEMD/mispredicts unless we actually flip all the bits plotted by *BinFI (Actual+Implied Bits Flipped)*. As a result, it is difficult to determine which bits are a higher priority to protect, which may be a tradeoff worth considering in the resource-constrained environments of edge NNs. Overall, BinFI performs well when the lion's share of a model's bits are sensitive and poorly when most of a model's bits are insensitive.

StatFI performs the worst in all cases in Fig. 5.8. Its statistical sampling is ineffective at selecting the sensitive bits in a NN. In particular, *StatFI (Data-aware)* depends on large changes in magnitude from flipping a weight bit to determine the sample size per weight bit index per layer. These large changes are more likely to occur in float32 and less likely to occur when we represent our weights with $\leq$8-bit fixed point precision. We therefore observe a large gap between the *StatFI (Data-aware)* line and the Hessian line in the ECON-T Medium Pareto (Fig. 5.8a), ECON-T Large Pareto (Fig. 5.8b), and CIFAR-10 (Fig. 5.8d) models charted. We would expect either StatFI method to work well for ECON-T Large Pareto, where few of the bits are sensitive because it was designed to find the few sensitive bits with fewer fault injections; however, StatFI randomly selects the bits to sample per weight bit index per layer, which is ineffective. For the ECON-T Small Pareto model (Fig. 5.8c), where 100% of the bits are sensitive, *StatFI Data-aware* fails to identify 37% of the sensitive bits, whereas *StatFI Data-unaware* fails to identify 5% of the bits. *StatFI (Data-unaware)* samples more bits, as we see in the pink line falling down after having flipped more bits than *StatFI (Data-aware)*'s brown line falling point in

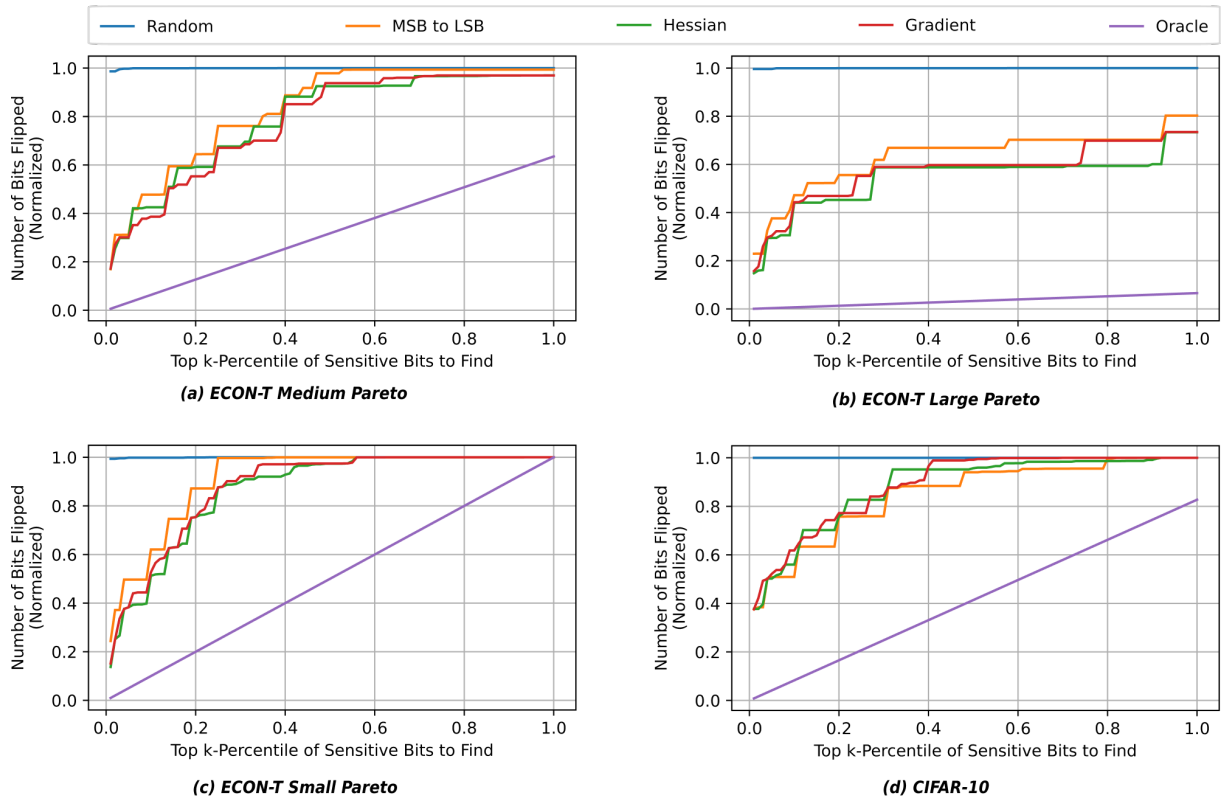


Figure 5.9. The ability of the sensitivity metrics to find the top k-percentile of the sensitive bits across the four NN models. The *oracle* provides a perfect ordering of the bits and thus provides a best-case result (lower bound).

every case; nevertheless, it fails to find many of the bits, especially for the ECON-T Large Pareto (Fig. 5.8b), missing 35% of the sensitive bits, and for CIFAR-10 (Fig. 5.8d), missing 76% of the sensitive bits. Both the *StatFI Data-aware* and *Data-unaware* methods are not sampling cleverly enough. Since the *Hessian* captures how sensitive the NN weights are, it easily outperforms both *StatFI Data-aware* and *StatFI Data-unaware*.

Fig. 5.9 provides a different analysis related to the ability of the sensitivity metrics to find the top-k percentile of the sensitive bits. The *Oracle* provides a perfect ranking with respect to the bit's sensitivity. In other words, the oracle perfectly selects the top-k percentile of the bits and provides a lower bound (best-case result) for a sensitivity metric. *Oracle* will not go to 1.0 on the y-axis when a subset of the bits are insensitive to single-bit faults. For example, only 63.5% of the bits are sensitive for the ECON-T Medium Pareto NN, only 6.55% of the bits

are sensitive for the ECON-T Large Pareto NN, 100% of the bits are sensitive for the ECON-T Small Pareto NN, and 82.72% of the bits are sensitive for the CIFAR-10 NN. *Random* provides the other extreme as the top-k bits are randomly distributed through its ranking, and thus the entire ranking must be enumerated to find the top-k bits for all but the smallest values of k.

For the ECON-T NNs, *MSB to LSB* performs the worst of the other metrics. Overall, *Hessian* is better than *Gradient* for the ECON-T Large and Small Pareto NNs. *Gradient* is overall generally lower (better) than *Hessian* for the ECON-T Medium Pareto model.

The CIFAR-10 classification task is interesting in that *MSB to LSB* performs quite well overall, while it is the worst sensitivity metric for the ECON-T NNs. The CIFAR-10 NN is a two-stack ResNet model with five convolutional layers [27]. This is fundamentally different than the ECON-T models, which have only two layers, especially for ECON-T Small which consists of only dense layers.

We then compare the top-k percentile performance of the *Hessian* and the *oracle* with BinFI [40] and StatFI [162] in Fig. 5.10. We first compare with BinFI. Let us first focus on the ECON-T medium model in Fig. 5.10a. To find the top-1 percentile of the sensitive bits, the *Hessian* only needs to flip $\sim$18% of the bits, whereas *BinFI (Actual+Implied Bits Flipped)* must flip $\sim$41% of the bits, taking longer to find the most sensitive bits. BinFI then drops to 0 after the top-15th percentile because it produces false negatives, i.e., it does not find all of the sensitive bits. The top-k percentile requirement is stringent. If a method fails to identify even a single bit in the top-k percentile, then we say this method has failed because there is no number of bits to flip according to that method that will find all of the top-k sensitive bits. We can thus infer from Fig. 5.10a that *BinFI (Actual+Implied Bits Flipped)* fails to identify bits beyond the top 15th percentile, e..g, it misses bits that are quite sensitive (such as in the 20th percentile). The *Hessian* provides both weight-level and bit-level sensitivity information to rank weight bits, whereas BinFI only provides bit-level sensitivity information per weight without any way of ranking the weights. Therefore, the *Hessian* is significantly more efficient than BinFI at finding the most sensitive bits because it captures more sensitivity information. For the ECON-T Large

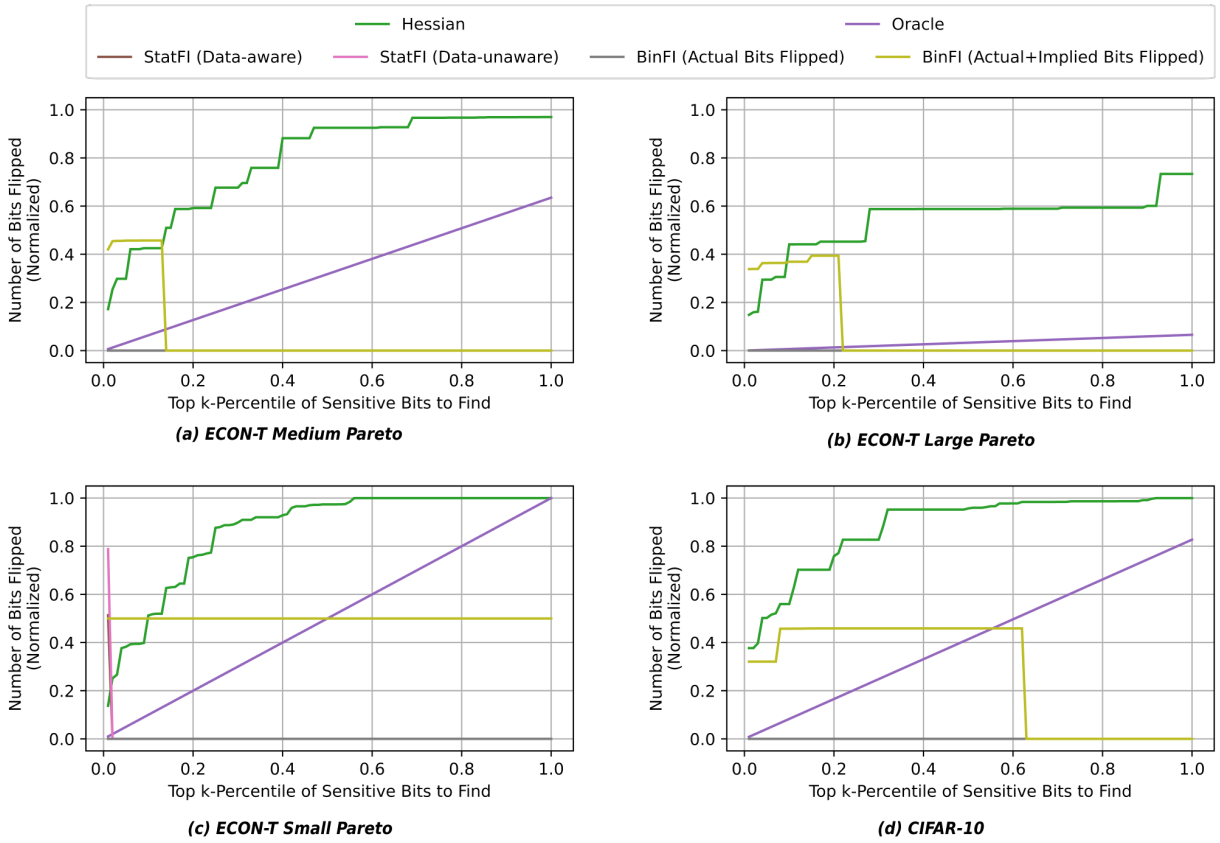


Figure 5.10. Comparing the ability of the sensitivity metrics to find the top k-percentile of the sensitive bits across the four NN models with state-of-the-art fault injection methods BinFI [40] and StatFI [162]. . The *oracle* provides a perfect ordering of the bits and thus provides a best-case result (lower bound).

Pareto (Fig. 5.10b), the Hessian outperforms *BinFI (Actual+Implied Bits Flipped)* for the top 10th percentile before *BinFI (Actual+Implied Bits Flipped)* exceeds the *Hessian* up until the top 20th percentile when it falls to 0—once again due to its failure to find sensitive bits. For ECON-T Small Pareto, the *Hessian* is the closest to the *oracle* up until the top 15th percentile when *BinFI (Actual+Implied Bits Flipped)* is the better method at finding top sensitive bits, eventually exceeding the *oracle* past the top 50th percentile. As previously discussed, this is due to *BinFI (Actual+Implied Bits Flipped)*'s implicating multiple bits as sensitive per bit flip, whereas the *oracle* only implicates one bit per bit flip. Since all the bits in ECON-T Small Pareto are sensitive, *BinFI (Actual+Implied Bits Flipped)* always succeeds in finding a sensitive bit. Moreover it only needs to flip about half of the bits to identify all of the sensitive bits. This is due to the easy nature of this task, i.e., when most if not all of the bits are sensitive. We see a similar pattern for the CIFAR-10 model where *BinFI (Actual+Implied Bits Flipped)* outperforms the *oracle* for a bit around the top 60th percentile before it falls to 0 due to its failure to identify all of the sensitive bits beyond the 60th percentile. Note that *BinFI (Actual Bits Flipped)* always lies at 0 for all four NNs because it is primarily flipping bits to implicate other bits in the model and is thus not good at flipping the most sensitive bits first.

We then compare with *StatFI (Data-aware)* and *StatFI (Data-unaware)*. For all four models, both StatFI methods fail to identify any top-k percentile sensitive bits, so all StatFI lines lie at 0, except for *StatFI (Data-unaware)* for the ECON-T Small Pareto model (Fig. 5.10c) which can find the top-1st percentile by flipping 80% of the bits before immediately falling to 0. Beyond this instance, no number of bits flipped according to StatFI will find some top-k percentile of the sensitive bits.

Next, we summarize the relationship between model size and the sensitivity of its weights. Fig. 5.11a plots the number of sensitive bits versus the total number of bits for the three ECON-T NNs. All of the bits in the ECON-T Small Pareto model are sensitive. As the model size increases, the number of sensitive bits decreases. The ECON-T Large Pareto model has only 6.55% of its bits sensitive to single-bit faults. Fig. 5.11b show the same three ECON-T models

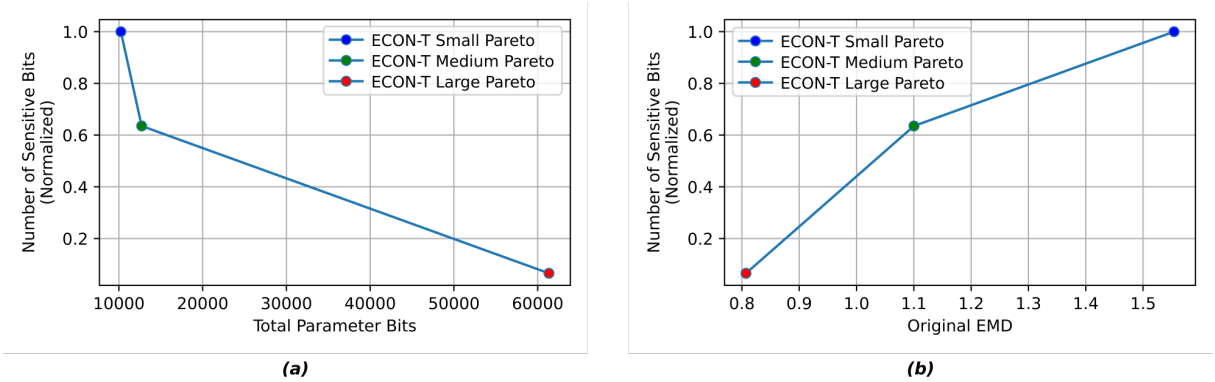
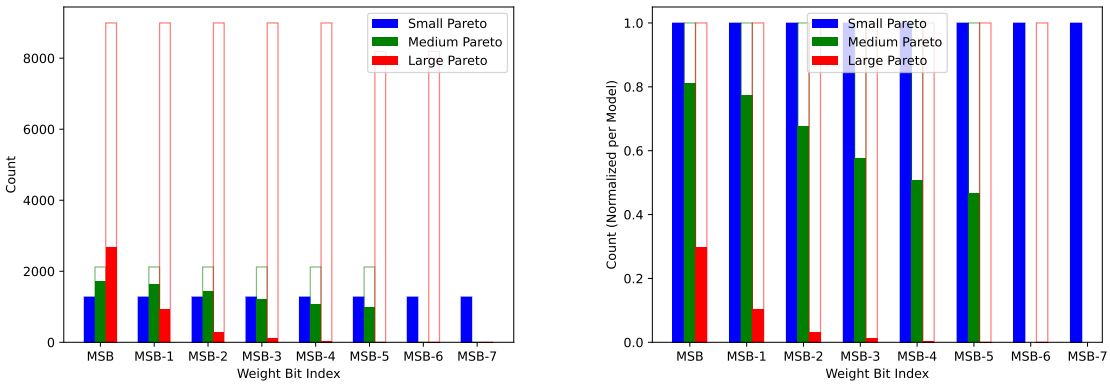


Figure 5.11. Part a) As model size increases, the percentage of sensitive bits in the model decreases. Part b) As *EMD* increases, the percentage of sensitive bits in the model increases.



(a) Total Sensitive Bit Count Distribution. **(b)** Normalized Sensitive Bit Count Distribution.

Figure 5.12. Distribution of the sensitive bits related to bit position for the three ECON-T NNs.

with respect to the EMD (error) of the non-faulty model. The ECON-T Large Pareto model has the smallest EMD (0.807), which is expected given that it is more complex. Reducing the model size increases the EMD (decreasing its performance).

Fig. 5.12 breaks out the number of sensitive bits according to their relative bit position in the weight from the MSB to the LSB. All models are quantized to a fixed-point representation such that `MSB` is a sign bit followed by 1–3 integer bits and some fractional bits remaining. Note that each model has a different quantization, with ECON-T Small Pareto having 8-bit weights, ECON-T Medium Pareto having 6-bit weights, and ECON-T Large Pareto having both 5-bit

and 7-bit weights. In the ECON-T Small Pareto NN, all the bits are sensitive; thus the sensitive bits are equally distributed across all bit indices. 63.5% of the bits are sensitive in the ECON-T Medium Pareto NN. The sensitive bits are relatively equally distributed across each bit index though more reside in the `MSB` and `MSB-1` bit indices. In the ECON-T Large Pareto NN, only a tiny fraction (6.5%) of the bits are sensitive. The sensitive bits are clustered in the first 3 MSBs out of (at most) 7 bits. Fig. 5.12b shows that when there are sensitive bits in the model, the majority of them reside in the most significant bits. Moreover, as model size increases, the percentage of sensitive bits decreases (as shown in Fig. 5.11a).

5.4.3 Multiple Bit Flip Results

We also aim to understand how the NNs respond to multiple-bit faults. Unlike the previous experiments, where we flip only one bit at a given time, we assume multiple bit flips are possible. Such scenarios are more likely in high radiation environments as is experienced by the ECON-T ASICs in the LHC. In particular, we want to understand the resilence of using protection mechanisms, e.g., using TMR on the NN weights as is done in the LHC [81].

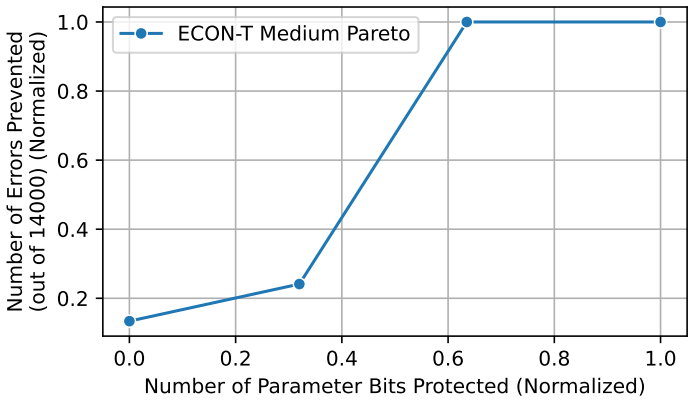


Figure 5.13. The effect of bit protection on errors for the ECON-T medium Pareto NN.

For this experiment, we assume a multi-bit flip fault model with a variable amount (0%, 32%, 63.5%, and 100%) of protection for the parameter bits. For the 100% protection level, all parameter bits are protected, which means that none of the attempted bit flips will occur. For the

63.5% protection level, we protect all of the sensitive bits identified under the single-bit flip fault model. For the 32% protection level, we protect the top 50% of the sensitive bits identified under the single-bit flip fault model. For the 0% protection level, we do not protect any parameter bits. Fig. 5.13 shows that by protecting all of the sensitive bits identified by the single-bit flip fault model, we can prevent all of the errors induced by the simultaneous multiple bit flips. Even without protection, a small percentage (13.4%) of the multi-bit flip errors are prevented.

5.5 Summary

We develop FKeras as a tool to assess the fault tolerance of edge neural networks that run inference fully on chip. FKeras provides several bit-level metrics that can quickly identify NN weight bits that are most sensitive to faults. We use FKeras to study four different NN models—three Pareto-optimal models for an autoencoder hardware in the CERN Large Hadron Collider and an edge NN that performs CIFAR-10 image classification. We show that the sensitivity of the bits varies greatly across weights and that the Hessian provides a good weight sensitivity ranking. Additionally, our results indicate that the sensitivity of different bits within a weight can vary dramatically. Even more, we found that larger, more accurate NNs have signficantly fewer sensitive bits compared with smaller, less accurate NNs. This raises some architectural codesign tradeoffs: should we design hardware for a larger, more accurate NN that requires less protection or a smaller, less accurate NN that requires full protection? How much more resilience can a larger NN provide at the expense of resource efficiency? FKeras provides valuable insights for addressing these codesign tradeoffs to design fault-tolerant, quantized NNs for hardware.

Chapter 5, in part, is a reprint of the material as it appears in ACM J. Auton. Transport. Syst. 1, 3, Article 15 (September 2024). The work in this chapter was done in collaboration with Andres Meza, Quinlan Bock, Benjamin Hawks, Javier Campos, Nhan Tran, Javier Mauricio Duarte, and Ryan Kastner. The dissertation author was the primary investigator and author of this work.

Chapter 6

PrioriFI: More Informed Fault Injection for Edge Neural Networks

In this chapter, I improve on FKeras's Hessian-based fault injection algorithm presented in the previous chapter. I present PrioriFI, which addresses the shortcomings of FKeras, primarily its assumption of bit-level monotonicity, that a more significant bit is more fault-sensitive than a less significant bit. In fact, I demonstrate that there are several exceptions to bit-level monotonicity across a variety of edge models and datasets.

6.1 Introduction

As machine learning (ML) becomes more capable, it is increasingly used in safety-critical applications, e.g., autono-mous vehicles [24], healthcare [171], and scientific fields like high-energy physics and electron microscopy [50]. In these cases, ML algorithms must operate correctly when encountering soft errors [26]. Mitigating soft errors is important in applications that operate under harsh conditions [25, 50, 61, 81, 209].

For example, high-energy physicists at the Large Hadron Collider (LHC) plan on deploying tens of thousands of endcap concentrator (ECON-T) ASICs, each running a neural network (NN) encoder that must complete inference in 25 ns within an area budget of 4 mm^2 [81]. The ECON-T ASICs operate in the High Luminosity LHC (HL-LHC), which exhibits 1000$\times$ the radiation in space (see Fig. 6.1) [25]. A particle strike can cause a bit flip in NN weights,

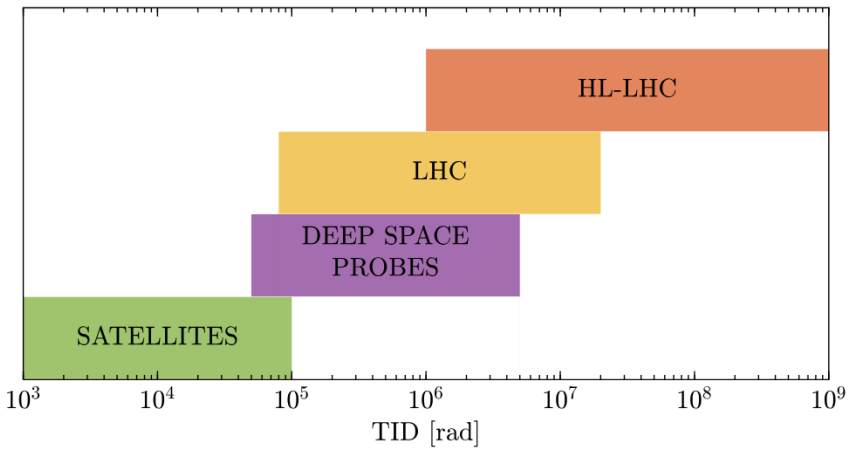


Figure 6.1. Edge NNs are deployed in high radiation environments. For example, high-energy physicists want to deploy edge NNs at the High Luminosity Large Hadron Collider (HL-LHC), which exhibits $> 1000\times$ the radiation in space when measured in Total Ionizing Dose (TID) [25]. Edge NNs deployed here will experience a fault every 15 seconds.

resulting in incorrect results. This edge NN must tolerate soft errors from particle strikes while operating within the given resource and latency constraints.

Edge NNs with tight latency and throughput constraints have unique characteristics, including heavy quantization and fully on-chip inference. Edge NNs often rely on custom hardware to meet such strict constraints and use design space exploration (DSE) to tune software and hardware to meet budgets on power, performance, and resources [27, 156, 168]. For instance, designers can use parity bits or triple modular redundancy (TMR) to protect their edge NNs—or no protection mechanisms at all if the edge NN is resilient enough for the given constraints. As a result, they need to understand how resilient their edge NN is to design the software and hardware accordingly. Previous research in NN fault tolerance often assumes that the weights and biases are stored in memory safeguarded by parity mechanisms [15, 40, 113], which is common for large NNs that retrieve weights and biases from off-chip memory [128]. Thus, previous work concentrates on faults in the outputs of NN layers, such as those in pipeline registers and arithmetic logic units. Yet, edge NNs often run on custom hardware, where designers choose how to protect their weights and biases, if at all. Since prior work has focused on large NNs under this fault model, the study of edge NN fault tolerance has been neglected [200].

Most prior work evaluates NN fault tolerance by performing fault injection (FI) campaigns. FI campaigns simulate faults in software or hardware and then run multiple inputs through the NN to evaluate its performance. This approach is computationally intensive due to the need to simulate many faulty scenarios. Earlier work has aimed to reduce the scope of the search space, with a few that target edge NNs [40, 162]. BinFI [40] performs a binary search FI campaign that relies on *bit-level monotonicity*—that a more significant bit is at least as sensitive as a less significant bit—to reduce the search space exponentially. FKeras [200] uses the Hessian and bit-level monotonicity to sort the most significant bits (MSBs) of an NN's weights by its Hessian ranking first, followed by the same ranking on the second MSBs, and so on.

However, bit-level monotonicity is not a strict rule that holds for an NN's parameters. There are many exceptions where a less significant bit is more sensitive than a more significant bit—not only within a parameter but also across the parameters of a model. For the NNs we study in this chapter, on average 15% of the NN bits violate bit-level monotonicity within a weight or bias and 80% violate monotonicity across weights or biases. These exceptions occur because faults get masked by operations that happen downstream in the NN, such as activation functions, leading to subpar FI campaign performance. BinFI suffers from falsely identifying weight or bias bits as sensitive or insensitive when bit-level monotonicity does not hold. FKeras injects faults in all of the MSBs before lesser significant bits, even though some lesser significant bits are more sensitive to faults than more significant bits. These exceptions in FKeras' Hessian ranking occur even for the least significant bits (LSBs) in the weights.

This chapter shows how to consider the exceptions in bit-level monotonicity during an FI campaign. *The key idea is to use the information gained during an FI campaign to dynamically prioritize the remaining campaign on the bits likely to be the next most sensitive, avoiding the pitfalls of assuming bit-level monotonicity.*

Our work PrioriFI[1] is a more informed FI algorithm that evaluates edge NN robustness by dynamically ranking NN bits based on their fault sensitivity. PrioriFI uses the Hessian for the

[1]PrioriFI is a portmanteau of "priority" and "FI", pronounced "priOR-uh-fy."

initial weight and bias ranking. Then, during an FI campaign, PrioriFI leverages the information gathered from each FI to prioritize subsequent FIs on the bits most likely to exhibit high fault sensitivity. PrioriFI's goal is to guide FI campaigns to inject faults on the most sensitive weight and bias bits first because the sensitivity of these parameters is variable. Many do not result in faulty behavior. Thus, a campaign should focus on injecting faults into the bits most susceptible to faults. With PrioriFI, designers can quickly evaluate different NN architectures and co-design fault-tolerant edge NNs.

When deploying NNs in high-radiation environments, the traditional power, performance, area (PPA) tradeoff space found in architecture and systems must expand to include reliability, a desirable design objective amid soft errors. This creates a PPA-reliability (PPAR) tradeoff space. PrioriFI is a hardware-software codesign tool that helps designers evaluate where an edge NN sits in the PPAR space, a key step when designing reliable systems and hardware for NNs. For example, designers can use PrioriFI to quickly ascertain what level of soft-error protection they need and whether to apply it at the hardware or system level. On the one hand, they could selectively apply a hardware mitigation technique only to sensitive NN bits, thereby reducing the area required for a given hardware design. On the other hand, designers could deploy protection mechanisms at the system level, such as a selective soft error detection protocol, potentially reducing detection latency and improving performance. Designers could also pick a model that is inherently more resilient to errors. This would improve system-level resilience, as different models have different fault sensitivity, a key result later seen in Sec. 6.4.2 (e.g., only 11% of ECON-T-L's bits are sensitive compared with 100% of ECON-T-S's bits). Designers can only make such decisions if they understand the sensitivity of their edge NNs first, and PrioriFI provides them with this data in an insightful way.

We evaluate the effectiveness of PrioriFI in incorporating fault analysis into edge NN DSE. We assess how PrioriFI performs on NNs trained on two high-energy physics datasets [61, 209] and the CIFAR-10 dataset [107]. While the scientific datasets represent applications in extreme, high fault-rate environments, the ML tasks themselves, e.g., classification, are common to many

other non-scientific, edge applications [18]. By studying these datasets, we show that PrioriFI works well on scientific edge NNs and takes a step towards generalizing to more standard edge NNs similar to CIFAR-10. For each dataset, we study three NN architectures, namely a small, medium, and large NN, representing a spread from small, less accurate NNs to large, more accurate NNs. Our results show that PrioriFI avoids many exceptions to bit-level monotonicity, identifying more sensitive bits in FI campaigns sooner.

Our primary contributions are: (1) analyzing the shortcomings of relying on bit-level monotonicity to accelerate fault injection campaigns, (2) introducing PrioriFI[2] to dynamically prioritize fault injection campaigns on the most sensitive bits first, and (3) conducting NN architecture design space explorations that incorporate fault tolerance alongside traditional optimization criteria such as performance and resource utilization.

6.2 Related Work

Identifying silent data corruptions (SDCs)—soft errors that lead to incorrect NN output [185]—is critical for fault tolerance. Three popular approaches are FI campaigns [129, 185], heuristics [44, 169, 170], and ML modeling [36, 187]. Few studies have focused on edge NNs that are quantized and operate on specialized hardware, such as FPGAs and ASICs [194, 200]. Such hardware is critical for low-power sensing applications deployed at the edge. For the few studies that apply to edge NNs, we identify their limitations stemming from dependence on bit-level monotonicity. Tab. 6.1 compares PrioriFI with prior work.

FI campaigns simulate a fault either in software [40, 126, 130, 159] or hardware [69, 159] and pass a number of inputs through the NN to assess performance. FI campaigns are computationally intensive due to the vast number of faulty scenarios that must be simulated, particularly given the thousands to millions of parameters and operations involved in an NN [185]. Researchers have developed methods to speed up FI campaigns through heuristics, statistical FI, and more [40, 69, 129].

[2]PrioriFI is open-sourced at https://github.com/KastnerRG/priorifi.

Table 6.1. Comparing prior work to PrioriFI. If the work is just an FI tool, then we note its **No False Positives/Negatives** and **Dynamic FI** abilities as not applicable (–). Dynamic FI means this method uses the information it receives during an FI campaign to guide future FIs in the campaign.

FI Tool /Method	FI Speedup Method	No False Positives /Negatives	Dynamic FI
Ares [159]	✗	–	–
TensorFI [112]	✗	–	–
PyTorchFI [126]	✗	–	–
GoldenEye [130]	✗	–	–
enpheeph [48]	GPU support	–	–
[44]	Gradient	✓	✗
Aspis [169]	Gradient	✓	✗
StatFI [162]	Statistical sampling	✗	✗
FKeras [200]	Hessian-guided FI	✓	✗
BinFI [40]	Element-wise binary search	✗	✓
PrioriFI (this work)	Priority-guided FI	✓	✓

Prior work uses heuristics to determine NN sensitivity through measurements like the gradient [44, 169] or patterns in error propagation [170]. Aspis [169] uses a first-order Taylor expansion to rank NN weights by sensitivity. While inherently less accurate than FI campaigns, heuristics are cost-effective, sacrificing accuracy to achieve faster execution.

Statistical fault injection [162], which we call "StatFI", reduces the FI search space. StatFI relies on bit flips that lead to large swings in the weight's magnitude. Based on this information, StatFI selects a subset of the bits in each bit position (e.g., MSB) in each layer to randomly flip. These large magnitude changes occur more frequently in floating point and less in fixed point, which are more common in edge NNs. Our results show that StatFI fails to identify many sensitive bits in edge NNs (as seen later in Fig. 6.8).

BinFI [40] relies on bit-level monotonicity, i.e., higher-order bits are at least as sensitive as lower-order bits, to reduce the number of faults to inject. BinFI performs a binary search

within each layer output. BinFI starts by flipping the middle bit first, leading to two outcomes: (1) if it is sensitive, BinFI assumes all bits more significant than it are sensitive, continuing its binary search on the remaining lower order bits, or (2) if it is insensitive, BinFI continues its binary search on the bits more significant than it. BinFI is effective because it reduces the search space by $O(\log(n))$ per layer output, where n is the number of bits per output. Bit-level monotonicity generally works for edge NNs, so BinFI can find most of the sensitive bits in an edge NN. But, as previously mentioned in Sec. 4.1, faults do not always behave monotonically [40]. As we discuss later in Sec. 6.3.2, there are many exceptions to bit-level monotonicity—where a lower order bit is more sensitive than a higher order bit. In fact, our results show that for the nine NNs we study, on average 80% of the NN parameter bits violate bit-level monotonicity across weights and 15% of the bits violate bit-level monotonicity within a weight. Such violations lead BinFI to incorrectly identify bits as sensitive (false positives) or insensitive (false negatives), as seen later in Tab. 6.6.

FKeras [200] provides various metrics to rank weight bits by fault sensitivity, guiding FI campaigns to target the most sensitive bits first. FKeras' best sensitivity metric uses the Hessian[3]. FKeras' Hessian ranking speeds up FI campaigns but has subpar performance. Although the Hessian is effective at ranking the weights, it does not know their binary representation, so it cannot inform the relative sensitivity of the bits. FKeras thus relies on bit-level monotonicity to sort the individual bits of the weights. FKeras sorts all of the MSBs by Hessian weight ranking, followed by the second MSBs by the same weight ranking, and so on. But, there are many instances when a second MSB is more sensitive than a MSB—both within a weight and across weights (as seen in our results Fig. 6.2). These exceptions to bit-level monotonicity explain why FKeras' bit-level ranking is not as good as it could be.

In response to these shortcomings, PrioriFI provides a bit-level ranking that combines the Hessian with information learned during an FI campaign. The key idea is that information

[3]FKeras efficiently computes the Hessian eigenvalues and eigenvectors, taking $\mathcal{O}(n)$ memory in $\mathcal{O}(n)$ time, where n is the number of NN parameters, by using techniques from randomized numerical linear algebra [208].

gained during an FI can guide the remainder of the campaign, avoiding the limitations of bit-level monotonicity. Our results show that PrioriFI identifies the most sensitive weight and bias bits with high accuracy and is thus a more effective FI algorithm (as seen later in Fig. 6.8).

6.3 PrioriFI

PrioriFI is a more informed fault injection algorithm for edge NNs. PrioriFI uses the Hessian and bit-level monotonicity to provide an initial parameter ranking. Then during an FI campaign, PrioriFI uses the sensitivity results it previously collected to guide the campaign to avoid the exceptions to bit-level monotonicity. This section defines the fault model and our sensitivity metric. Then, we describe the limitations of assuming bit-level monotonicity in an NN. Finally, we provide algorithmic details of PrioriFI.

6.3.1 Fault Model

The single-bit flip model, which simulates scenarios where only one bit flips at a time, is widely used [40, 69, 129]. This fault model can be applied to NN weights, biases, activations, and even at finer granularity [88]. Since many edge NNs execute fully on-chip and require custom hardware, our study focuses on bit flips in NN weights and biases, as designers must consider how to protect them.

Given this fault model, we want to determine which NN weight and bias bits are sensitive to faults. Some bits are more sensitive than others, meaning flipping them results in significantly worse model performance than flipping others. Some bits are not sensitive, i.e., flipping them leads to correct results. Thus, we want to measure the different levels of fault sensitivity.

NN Sensitivity Metric

We define a sensitivity metric to measure which bits are more sensitive than others.

Definition 1 (Sensitivity Metric ΔC)

$$\Delta C = \max(\overline{C}_{faulty} - \overline{C}_{faultless}, 0) \tag{6.1}$$

C is the cost of NN execution, $\overline{C}_{faulty}$ is the average cost with an FI, and $\overline{C}_{faultless}$ is the average cost without an FI. Thus, when flipped, a sensitive bit results in $\Delta C > 0$, and an insensitive bit results in $\Delta C = 0$, i.e., no cost to faulty NN execution.

We determine C based on the NN's task at hand. For example, for classification tasks, C is the number of mispredicts. Given X, the correctly classified samples from the NN's validation dataset, we have $\forall x \in X, C_{faultless}(x) = 0$, so $\overline{C}_{faultless} = 0$. In other words, a sensitive bit leads to worse model performance when flipped.

Oracle

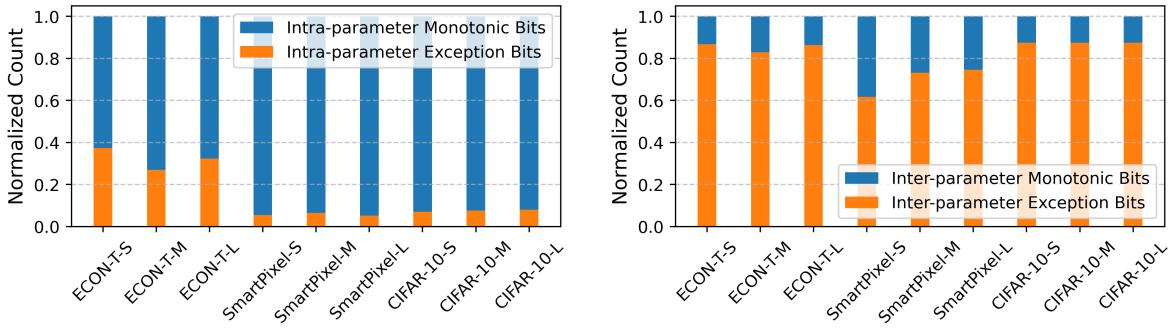
We perform an exhaustive FI campaign using the fault model and sensitivity metric to provide an *Oracle*. The Oracle knows the sensitivity metric value for every NN weight and bias bit when flipped. To generate the Oracle, we use FKeras's [200] FI capabilities to exhaustively perform single bit-flips for each NN (more details in Sec. 6.4.1). We use the Oracle to describe the limits of bit-level monotonicity (Sec. 6.3.2) and to evaluate the performance of PrioriFI (Sec. 6.4).

6.3.2 Limits of Bit-level Monotonicity

Bit-level monotonicity indicates that higher-order bits are more likely to be sensitive to faults than lower-order bits. Previous work uses bit-level monotonicity to accelerate FIs, e.g., BinFI [40] and FKeras [200]. But there are shortcomings.

Let us first define bit-level monotonicity more formally.

Definition 2 (Bit-level Monotonicity) *Bit-level monotonicity says higher-order bits are at least*



(a) Intra-parameter exceptions

(b) Inter-parameter exceptions

Figure 6.2. Distribution of intra- and inter-parameter exceptions of bit-level monotonicity. There are two ways to break bit-level monotonicity, where a lower-order bit exhibits higher sensitivity than a higher-order bit: (1) intra-parameter (within a parameter) and (2) inter-parameter (between parameters). In the nine NNs we study, on average 80% of a NN's parameter bits exhibit inter-parameter exceptions and 15% exhibit intra-parameter exceptions.

as sensitive as lower-order bits:

$$for\ x \geq y, \Delta C_x \geq \Delta C_y \tag{6.2}$$

where x is a higher-order bit and y is a lower-order bit, ΔC_x is the sensitivity of flipping bit x, and ΔC_y is the sensitivity of flipping bit y. This is based on the definition found in the BinFI paper [40].

Bit-level monotonicity is generally valid, especially within a parameter, because higher-order bits have a greater influence on the overall computation within an NN; however, this is not always true. Flipping higher-order bits causes a larger change in magnitude in an NN parameter compared with flipping a lower-order bit [40]. Prior work like StatFI [162] uses magnitude change to guide their FI campaign. Despite this intuition, there exist many instances when flipping a higher-order bit leads to a lower ΔC compared with flipping a lower-order bit, violating bit-level monotonicity [40, 129].

In fact, there are two types of exceptions to bit-level monotonicity: (1) *intra-parameter* (within a parameter in Fig. 6.2a) and (2) *inter-parameter* (between parameters in Fig. 6.2b). Based on Def. 2, an NN parameter bit violates *intra-parameter* monotonicity if it is less sensitive

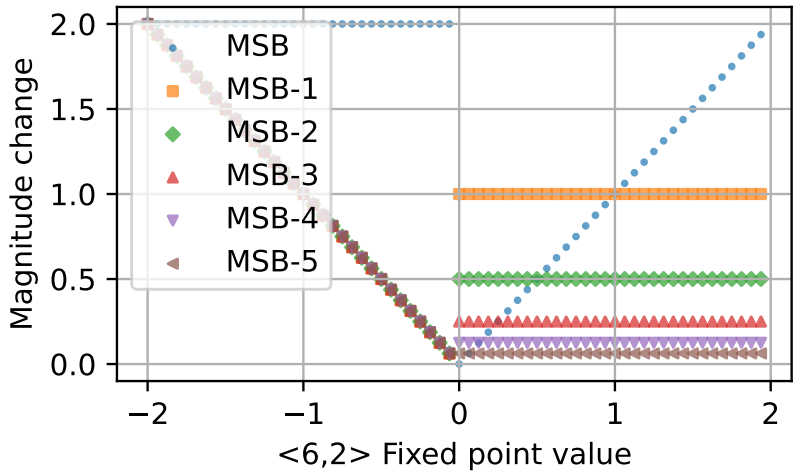


Figure 6.3. Applying ReLU to fixed-point values after a bit flip breaks bit-level monotonicity. In this example, we flip a bit for every value in the $< 6, 2 >$ fixed-point representation (6 bits total, 2 bits integer) and apply a ReLU activation function to the result, plotting the resulting change in magnitude. Note that the brown-dotted diagonal on the left represents the overlapping of all the lower-order bits (MSB-1 and lower). Without ReLU, bit flips are monotonic. With ReLU, bit flips are not strictly monotonic, as seen in prior work [40, 129].

than a lower-order bit in the same parameter. A bit violates *inter-parameter* monotonicity if it is less sensitive than a lower-order bit found in any of the other parameters. We count the number of bit-level monotonic exceptions in the nine edge models we study later in Sec. 6.4, plotting them in Fig. 6.2. These models are trained on three edge datasets: the HGCal dataset [61], the Smart Pixel dataset [209], and CIFAR-10 [107]. For each dataset, we trained a small, medium, and large model to represent a spread in model size, accuracy, and fault sensitivity. Note, the ECON-T model is trained on the HGCal dataset. More details are found in Sec. 6.4.1. As seen in Fig. 6.2, there are significantly more inter-parameter than intra-parameter exceptions. On average, 80% of the bits exhibit inter-parameter exceptions and 15% exhibit intra-parameter exceptions. This is likely due to the fact that there are many more lower-order bits across parameters compared to within a single parameter. These exceptions can occur because faults are masked by operations that happen downstream in the NN, such as activation functions, leading to non-monotonic behavior.

Algorithm 2: PRIORIFI

```
let N = NN model
let p = NN parameters
let p_ranked = hessianRanking(p)
let Bs = sortBitsByHessian(p_ranked)
Note: Bs is a list of ranked bit lists sorted by bit significance order, i.e., [[MSBs], [MSB-1s], ...]
let deltaCs = {} // store ΔCs here
let k = how many of the last ΔCs to consider
Function nextBitFlip(bitLists, deltaCs, k):
    ms = {}
    for i in len(bitLists) do
        m = medianLastK(deltaCs[i], k)
        ms[i].append(m)
    end
    a = argsort(ms, reverse=True)
    for i in len(a) do
        if bitLists[i] is not empty then
            nextBit = a[i]
            break
        end
    end
    return nextBit
// Flip first bit in each bit list
for i in len(Bs) do
    N' = flipBit(Bs[i][0], N)
    d = deltaC(N') // Measure ΔC of N'
    deltaCs[i].append(d)
    Bs[i].pop()
end
// Priority-based FI
while ∀b ∈ Bs, b is not empty do
    f = nextBitFlip (Bs, deltaCs, k)
    N' = flipBit(Bs[f][0], N)
    d = deltaC(N')
    deltaCs[i].append(d)
    Bs[f].pop()
end
```

In Fig. 6.3, we present how masking causes non-monotonic behavior in edge NNs. In this example, we select a $< 6, 2 >$ fixed-point representation, where there are 6 total bits and 2 integer bits (one of which is a sign bit). This is the data type that the ECON-T-M uses to represent its parameters; however, the trends presented here generalize to other signed fixed-point representations. For each $< 6, 2 >$ value, we flip a bit, one of MSB, MSB-1 (second MSB), etc., and then apply ReLU (Fig. 6.3), a common activation function used in NNs. We then plot the resulting magnitude change for that value.[4] With the ReLU activation function, which itself is monotonic, magnitude changes behave non-monotonically—higher-order bits may result in smaller magnitude changes compared to lower-order bits, e.g., fixed point value 0.5. We thus observe non-monotonic behavior within a fixed-point value due to masking by a ReLU.

Thus, there exist many exceptions to bit-level monotonicity, especially across parameters (inter-parameter). These exceptions can occur because faults are masked in a variety of ways as they propagate through the NN, as seen in Fig. 6.3.

6.3.3 PrioriFI Algorithm

Although bit-level monotonicity provides a first-order approximation for bit sensitivity, it has numerous exceptions, particularly across different parameters. Thus, PrioriFI uses bit-level monotonicity as a starting point and then uses methods that consider *inter-parameter* importance to avoid the many exceptions to monotonicity, especially given how prevalent inter-parameter exceptions are (see Fig. 6.2b). During an FI campaign, PrioriFI uses information gained on the fly to guide its next FI to prioritize inter-parameter exceptions. This way, it avoids the pitfalls of bit-level monotonicity, targeting the most sensitive bits first.

Alg. 2 describes the PrioriFI algorithm in detail. PrioriFI begins by ranking the bits of each significance, e.g., MSB, MSB-1, etc., using FKeras' Hessian sensitivity metric. This metric has been shown to be effective at ranking NN parameters by fault sensitivity [200]. In the initial

[4] In Fig. 6.3, the brown-dotted diagonal on the left represents the overlapping of all the lower-order bits (MSB-1 and lower).

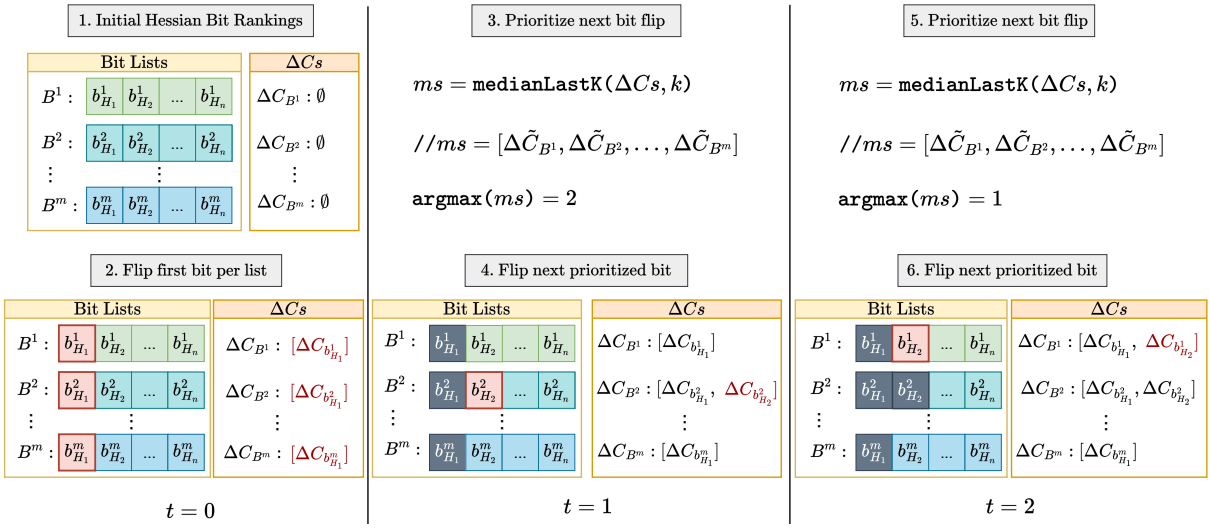


Figure 6.4. The PrioriFI algorithm in action. For an NN with n weights and biases quantized to m bits, we first rank the bits of the same significance using the Hessian from most to least sensitive. For instance, the MSBs are in list B^1, where the highest ranked MSB is $b^1_{H_1}$, where H_1 is the highest Hessian ranking. B^2 contains the second MSBs and so on till we reach B^m the LSBs. Then, we flip the first bit in each of the bit lists, i.e., the highest Hessian-ranked bits, and record the corresponding ΔCs. This ends the initial phase ($t = 0$). Next, we enter the priority-based FI phase. For the first iteration ($t = 1$), we take the median of the last k measurements for each ΔC_{B^i}, resulting in a list of medians ms, which we denote as $\Delta\tilde{C}_{B^i}$. Note the ms are sorted in bit significance order, where $\Delta\tilde{C}_{B^1}$ is first, then $\Delta\tilde{C}_{B^2}$, and so on till $\Delta\tilde{C}_{B^m}$. Then we take the argmax of the ms, i.e., $\max_i \Delta\tilde{C}_{B^i}$. Note, this is a simplified version of the `nextBitFlip()` function in Alg. 2, which ensures the argmax corresponds to the highest-ranked non-empty B^i list. In the $t = 1$ case, the argmax is 2, so we flip the next bit in the B^2 list and record its ΔC. This process continues in the next iteration ($t = 2$).

phase, PrioriFI flips the highest-ranked bit of each significance, measuring the ΔC per bit flip. Next, in the priority-based FI phase, PrioriFI takes the median of the last k ΔC measurements per bit significance. The highest median corresponds to the bit significance that the FI campaign should flip next, e.g., the next highest-ranked MSB-1. By using the ΔC measurements to guide our FI campaign, we account for inter-parameter exceptions to bit-level monotonicity, prioritizing these exceptions when they occur to find the more sensitive bits earlier. Fig. 6.4 demonstrates the first few iterations of the PrioriFI algorithm, providing a detailed explanation.

6.4 Evaluation

In this section, we describe our experimental setup and evaluate PrioriFI.

6.4.1 Experimental Setup

We demonstrate how PrioriFI efficiently performs an FI campaign on three edge datasets: the HGCal dataset [61], the Smart Pixel dataset [209], and CIFAR-10 [107]. The HGCal and Smart Pixel datasets are two high-energy datasets. Both of these datasets represent edge applications that occur in the HL-LHC, a high-radiation environment, as previously shown in Fig. 6.1. While these datasets represent applications in extreme fault-rate environments, the ML tasks themselves are common to many other non-scientific, edge applications [18].

The HGCal dataset contains sensor images from high-energy particle collision sensor data. The ECON-T autoencoder previously mentioned (Sec. 4.1) is trained on the HGCal dataset. The goal of the ECON-T is to compress this data to a lower-dimension latent space (and then later decode it offline for further processing) [81]. We only evaluate the fault tolerance of the encoder (and not the decoder) because it is the only part of the NN exposed to radiation. The Smart Pixel dataset contains vectors representing 13 key features of high-energy particle clusters from sensor data. The goal of an edge NN trained on Smart Pixel data is to classify the clusters into three classes of high-momentum and low-momentum particles. CIFAR-10 is a popular image classification dataset.

For each dataset, we study three differently sized edge NN architectures, in particular, a small, medium, and large NN. Each model is quantized to a low-bitwidth (ranging from 3 to 8 bits) fixed-point representation due to resource constraints at the edge. This represents a spread of models that researchers often consider when they perform design space exploration to find an optimal edge model for their given constraints, such as model performance, latency, and resources [27, 156, 168]. In our experiments, the three models per dataset represent a tradeoff between model accuracy and size. The ECON-T autoencoders feature convolutional

and dense layers, the SmartPixel NNs are fully-connected, and the CIFAR-10-S and -M NNs are convolutional whereas CIFAR-10-L is a ResNet8 [61]. Fig. 6.5 provides detailed layer-by-layer information on the layer type, quantization, and kernel size of each edge NN we study in this chapter. All NNs were quantized via quantization-aware training using QKeras [80]. The ECON-T models are based on the models presented in the FKeras paper [200]. The SmartPixel models are based on the models presented in the SmartPixel paper [209]. The CIFAR-10 models are based on the hls4ml-FINN submission to the MLPerf Tiny Benchmark [27].

We use FKeras [200] to perform our FI campaigns on the weights and biases of each model. We ran our experiments on an AMD Ryzen Threadripper PRO 7985WX 64-Cores CPU and an NVIDIA RTX 4500 Ada Generation GPU. We first generate the Oracles for each model. For HGCal, we run each FI on 20 000 validation inputs. We evaluate ECON-T NN sensitivity using a ΔC where C is Earth Mover's Distance (EMD) [160]. EMD is the reconstruction loss between the input and the decoded output, measuring the distance between two probability distributions. An EMD of 0 indicates a lossless reconstruction. Since our ECON-T NNs are lossy, we define $\overline{C}_{faultless} = \overline{\text{EMD}}$, i.e., the average EMD of the model under non-faulty conditions. For Smart Pixel and CIFAR-10, we run each FI on a subset of the 11 113 and 10 000 validation inputs, respectively, namely the inputs that each model classifies correctly. We evaluate Smart Pixel and CIFAR-10 NN sensitivity using ΔMispredicts. Tab. 6.2 describes the baseline model performance, the total number of weight and bias bits, and the number of FI validation samples for each edge NN.

6.4.2 More informed fault injection

We evaluate the efficiency of PrioriFI with respect to its ability to guide an FI campaign. Our evaluation is conducted on the nine edge NNs (Tab. 6.2). We compare PrioriFI's FI performance against the following FI algorithms:

1. *Random* produces a ranking where all bits are included in a random order.

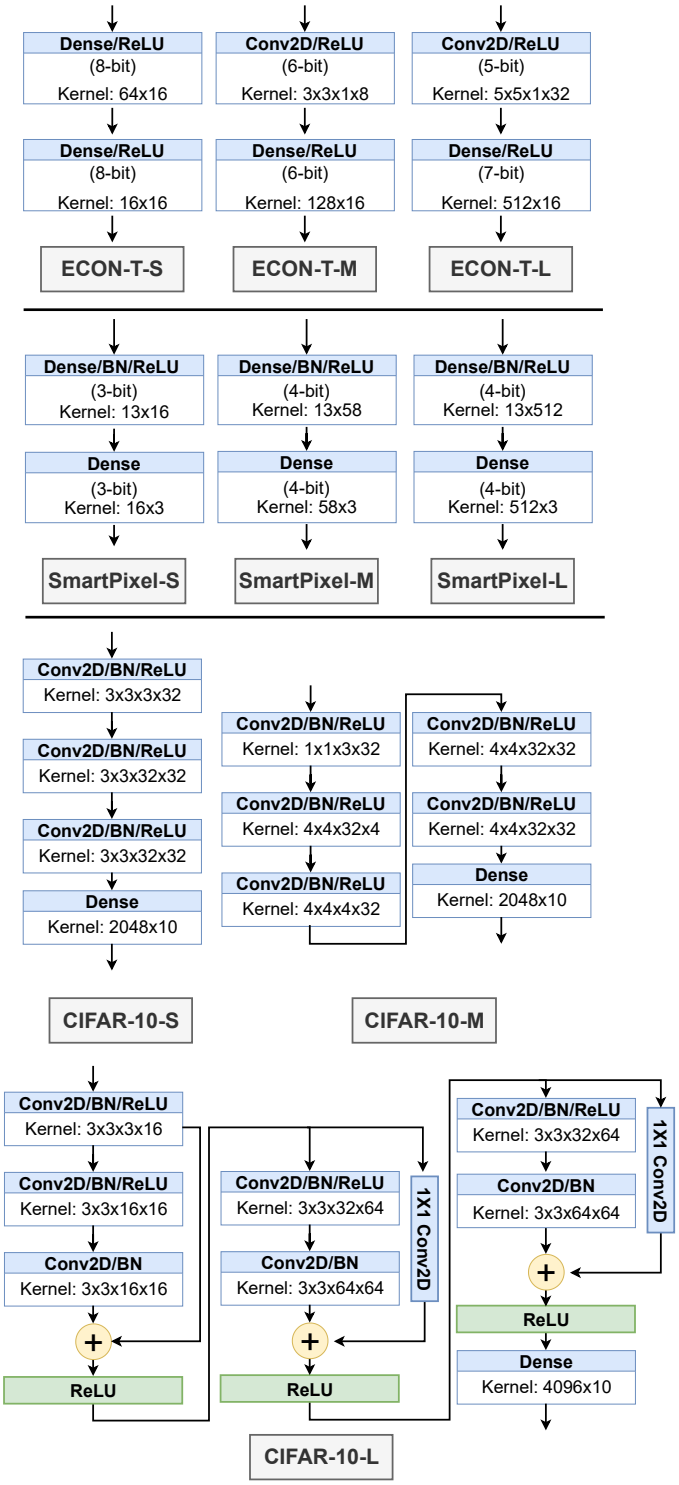


Figure 6.5. Edge NNs studied. Note that all CIFAR-10 models have been quantized to 8-bit fixed point.

Table 6.2. Experimental setup. For CIFAR-10 and SmartPixel, the **# FI samples** is the number of correctly classified samples out of 10 000 and 11 113 total validation samples, respectively. Note, the ECON-T model performance is measured by **EMD** and the SmartPixel and CIFAR-10 model performances are measured by classification accuracy (**Acc. (%)**).

NN	EMD	Acc. (%)	Total weight & bias bits	# of FI samples
ECON-T-S	1.55	-	10 496	20 000
ECON-T-M	1.10	-	12 864	20 000
ECON-T-L	0.81	-	61 616	20 000
SmartPixel-S	-	70.5	825	7 835
SmartPixel-M	-	72.9	3 956	8 098
SmartPixel-L	-	74.2	34 828	8 244
CIFAR-10-S	-	77.3	319 056	7 727
CIFAR-10-M	-	83.1	460 656	8 313
CIFAR-10-L	-	86.2	944 208	8 616

2. *MSB-to-LSB* produces a ranking where all bits are ordered according to their bit significance such that all MSBs come before all MSB-1s, etc.

3. *BinFI* produces a ranking based on a binary FI algorithm [40]. BinFI was originally designed to target the activations, but we adapt it to target the weights and biases. This approach has three variants that are described later in this section.

4. *StatFI* produces a ranking based on a statistical FI algorithm [162]. This approach has Data-Aware and Data-Unaware variants that adjust the sample size used in their algorithm.

5. *Taylor* as presented in Aspis [169] introduced a ranking based on the first-order Taylor expansion to sort the weights and biases by importance. Since they do not provide a bit-level ranking, we apply an MSB-to-LSB ordering to its initial ranking.

6. *Hessian* produces a bit-level ranking by ordering bits based on a parameter-level Hessian metric and then applying an MSB-to-LSB ordering from there. This ranking was introduced in [200].

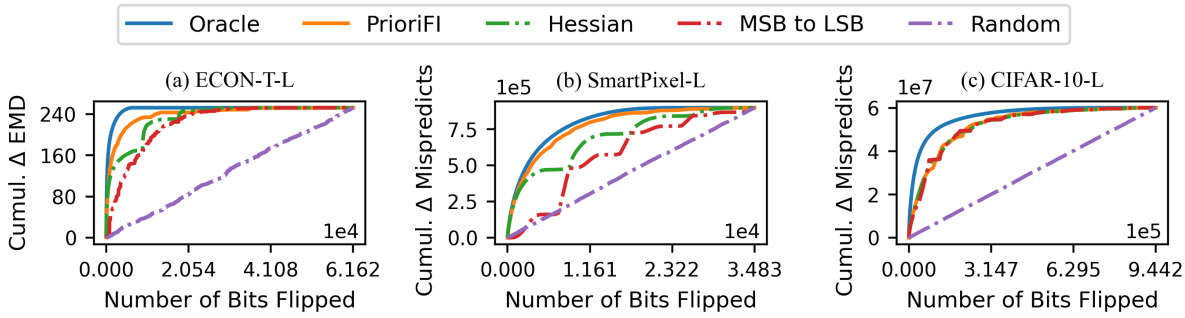


Figure 6.6. PrioriFI improves on rankings relying on bit-level monotonicity. We measure the cumulative ΔC as we flip more bits in each NN. Each model plots five rankings that attempt to order the NN weight and bias bits from those that contribute the most error to those that contribute little or no error. We see that PrioriFI adjusts for the exceptions to bit-level monotonicity found in the MSB-to-LSB and Hessian rankings, finding more sensitive bits faster. Note that the MSB-to-LSB ranking has few exceptions to bit-level monotonicity for CIFAR-10, so there is less room for improvement with PrioriFI.

7. *Oracle* produces a perfect ranking of the bits as described in Sec. 6.3.1.

Based on our Oracles, we found a different percentage of sensitive bits per NN per dataset. For the ECON-T NNs, 100%, 82%, and 11% of the bits were sensitive for ECON-T-S, ECON-T-M, and ECON-T-L, respectively. For the SmartPixel NNs, 86%, 79%, and 64% of the bits were sensitive for SmartPixel-S, SmartPixel-M, and SmartPixel-L, respectively. For the CIFAR-10 NNs, 83%, 87%, and 76% of the bits were sensitive for CIFAR-10-S, CIFAR-10-M, and CIFAR-10-L, respectively.

Fig. 6.6 features the results from our first set of experiments, demonstrating how PrioriFI improves upon rankings that rely on bit-level monotonicity. In these experiments, we compare the performance of PrioriFI to that of the Hessian and MSB-to-LSB, which base their rankings on bit-level monotonicity. The Oracle ranking and Random ranking provide the best and worst bounds, respectively, on the performance one can expect from an FI campaign on the edge NNs under analysis. In Fig. 6.6a and Fig. 6.6b, the Hessian outperforms MSB-to-LSB, and PrioriFI outperforms the Hessian. In this context, outperforming equates to requiring fewer FIs (bit flips) to identify the model bits that produce the greatest ΔC. The plateaus in the Hessian ranking appear because it fails to account for exceptions to bit-level monotonicity. PrioriFI has a smooth

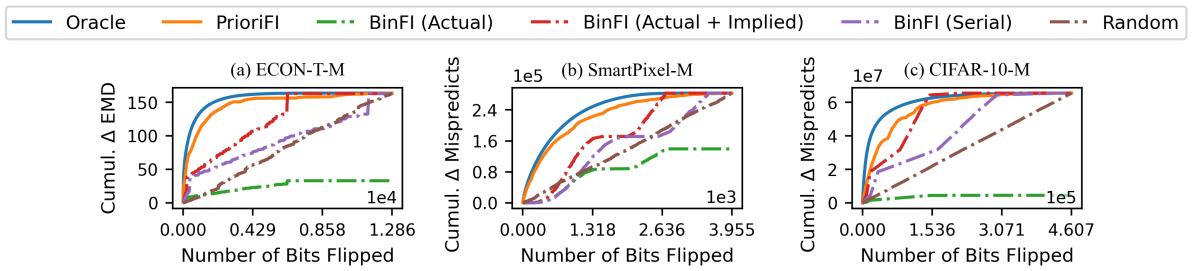


Figure 6.7. The spread of BinFI rankings. BinFI reduces the number of FIs in a campaign by implying bits to be sensitive. *BinFI (Actual)* are the actual bits that BinFI flips, representing the cumulative ΔC it actually learns. *BinFI (Actual + Implied)* includes the actual and implied sensitive bits that BinFI learns, representing the cumulative ΔC it theoretically could attain. *BinFI (Serial)* is derived from flipping the *BinFI (Actual + Implied)* bits in order, representing the cumulative ΔC it truly learns if it were to flip the implied bits too. *BinFI (Actual)* falls below the *Random* ranking because it does not flip all bits in the model.

curve without plateaus because it dynamically adjusts for these exceptions.

Fig. 6.7 showcases the results from our experiments exploring the performance differences between three variants of BinFI. *BinFI (Actual)* represents the Δ*C* that is actually measured by flipping a bit when conducting an FI campaign guided by BinFI. *BinFI (Actual + Implied)* represents the theoretical maximum performance of a BinFI approach when multiple bits are implied as being sensitive based on a single bit flip in a lower-order bit. However, the Δ*C* of the implied bits cannot be verified without actually flipping a bit, so the theoretical gains in information are not visible to the FI process—they must be assumed. *BinFI (Serial)* represents a slightly less efficient (with respect to the number of bit flips) approach that aims to remove ambiguity in the implied bits by flipping them to confirm they are indeed sensitive bits. We adopt the serial variant of BinFI in the remainder of our experiments since it fits best with the ambiguity/risk tolerance of the target applications for our NNs.

Fig. 6.8 shows PrioriFI's advantage over FI approaches from the related work. PrioriFI consistently outperforms the other approaches. Across a variety of NN architectures (the rows in Fig. 6.8) and model sizes (the columns in Fig. 6.8), PrioriFI's dynamic approach yields greater cumulative Δ*C* earlier in the FI process, as evidenced by the smoothness of its curves. The "bumps" that occur in the related work demonstrate how they fail to account for non-monotonic

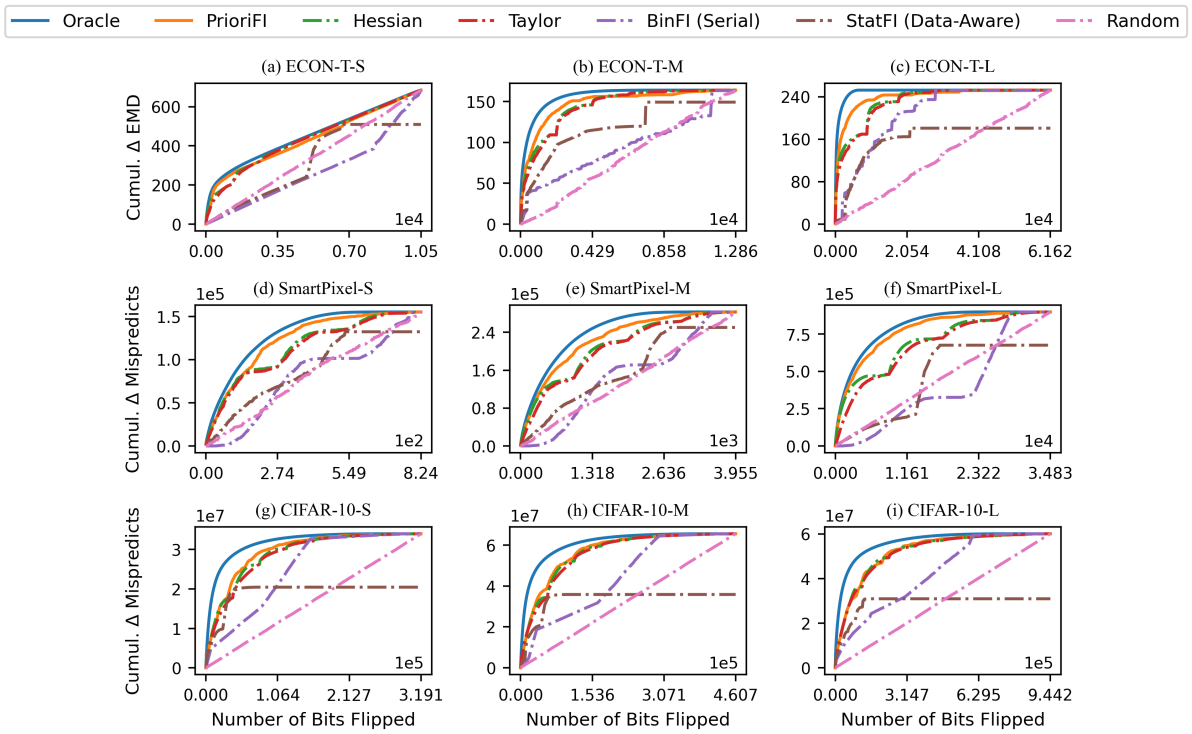


Figure 6.8. Comparing PrioriFI to related work. PrioriFI finds more sensitive bits faster than related work does. It accumulates more ΔC error in fewer bit flips, i.e., identifying the more critical sensitive bits faster.

behavior. We further quantify these results by computing the area under the curve (AUC) for each of the rankings shown in Fig. 6.8 and dividing it by the Oracle's AUC, as shown in Tab. 6.3. We use AUC because we want to quantify how close to the Oracle the other rankings are in this chart. Since the Oracle is a perfect ranking, we present its AUC as 1. PrioriFI is the closest to the Oracle ranking with the highest AUCs for seven out of the nine models evaluated. Even for ECON-T-S, PrioriFI is only 0.01 AUC worse than the Hessian.

In Tab. 6.3, PrioriFI improves on prior work such as the Hessian. The Hessian performs well in cases where bit-level monotonicity generally holds, but even in these cases, PrioriFI works nearly as well (ECON-T-S), if not the same (CIFAR-10-L). PrioriFI relies on the quality of the initial Hessian ranking and how often inter-parameter exceptions to bit-level monotonicity occur. For the ECON-T and SmartPixel models, the Hessian provides a good ranking, as seen in how close the Hessian ranking is to the Oracle ranking in the first few bit flips in Fig. 6.8. But,

Table 6.3. Quality of PrioriFI's ranking using AUC. We normalize each ranking's area under the curve (AUC) from Fig. 6.8 to the Oracle's AUC to quantify how quickly each ranking finds sensitive bits. PrioriFI improves on seven out of the nine models.

Ranking	ECON-T			SmartPixel			CIFAR-10		
	S	M	L	S	M	L	S	M	L
Oracle	1	1	1	1	1	1	1	1	1
Random	0.75	0.51	0.53	0.59	0.59	0.56	0.54	0.54	0.54
StatFI (Data-Aware)	0.70	0.76	0.64	0.69	0.66	0.57	0.61	0.55	0.52
BinFI (Serial)	0.60	0.61	0.83	0.61	0.61	0.50	0.79	0.73	0.75
Taylor	0.96	0.92	0.92	0.89	0.86	0.84	0.91	0.92	0.92
Hessian	**0.97**	0.93	0.93	0.88	0.88	0.87	0.92	0.93	**0.93**
PrioriFI	0.96	**0.95**	**0.96**	**0.95**	**0.94**	**0.97**	**0.93**	**0.94**	**0.93**

this trend is seen less in the CIFAR-10 models, as the Hessian only provides an approximation of sensitivity. We also see in Fig. 6.6c that CIFAR-10-L does not have as many bit-level monotonic exceptions, since the MSB-to-LSB ranking already works quite well. Even so, PrioriFI improves on the rankings relying on MSB-to-LSB monotonicity, such as the Taylor and Hessian. In contrast, for the ECON-T and SmartPixel models, we observe the poor performance of the MSB-to-LSB ranking as well as the many plateaus in the Hessian ranking, indicating that there are many exceptions to bit-level monotonicity.

Investigating monotonic exceptions in NNs

To better understand why the CIFAR-10 NNs do not have many monotonic exceptions, we observe that not all exceptions are equal. Although there are many inter-parameter exceptions (previously seen in Fig. 6.2b), we can imagine that a given higher-order bit is less sensitive than only a few lower-order bits in the NN, whereas another higher-order bit is less sensitive than many more lower-order bits. Thus, some parameter bits have more severe inter-parameter monotonic exceptions than others. To quantify the severity of a bit's monotonic exceptions, or lack thereof, we develop a metric called *monoscore*. It measures how monotonic or non-monotonic a bit is at an inter-parameter level. More formally, the monoscore is defined below.

Definition 3 (Monoscore) *Given a NN with n bits, the monoscore of a bit b is defined as:*

$$monoscore(b) = \frac{1}{n}\sum_{i=0}^{n} score(b, d_i) \tag{6.3}$$

where b is a given bit and d_i is the i-th bit. score(·) is defined as:

$$score(b^x, d^y) = \begin{cases} +1 & x > y, \Delta C_{b^x} \geq \Delta C_{d^y} \quad (6.4a) \\ +1 & x < y, \Delta C_{b^x} \leq \Delta C_{d^y} \quad (6.4b) \\ 0 & x = y \quad (6.4c) \\ -1 & x > y, \Delta C_{b^x} < \Delta C_{d^y} \quad (6.4d) \\ -1 & x < y, \Delta C_{b^x} > \Delta C_{d^y} \quad (6.4e) \end{cases}$$

where b^x is a bit with significance x, d^y is a bit with significance y, Eq. 6.4a and Eq. 6.4b define monotonic behavior, Eq. 6.4d and Eq. 6.4e define non-monotonic behavior, and Eq. 6.4c defines when bits b and d have equal significance, e.g., both are MSBs and are neither monotonic nor non-monotonic.

In other words, to compute the monoscore for a given bit, we first compare it with another bit in the NN. We add +1 if it behaves monotonically with the other bit, 0 if the two bits have the same bit significance, and -1 if it behaves non-monotonically with the other bit. We do this for all the bits in the NN and divide by the total number of NN bits to get an average for how monotonic this bit is relative to the rest of the NN bits. If a bit exhibits strictly monotonic behavior, its monoscore will be 1, and if it exhibits strictly non-monotonic behavior, its monoscore will be -1. A monoscore of 0 means the bit behaves monotonically with half the bits in the NN and non-monotonically with the other half.

To compute the monoscore for a whole model, or the *model monoscore*, we average all

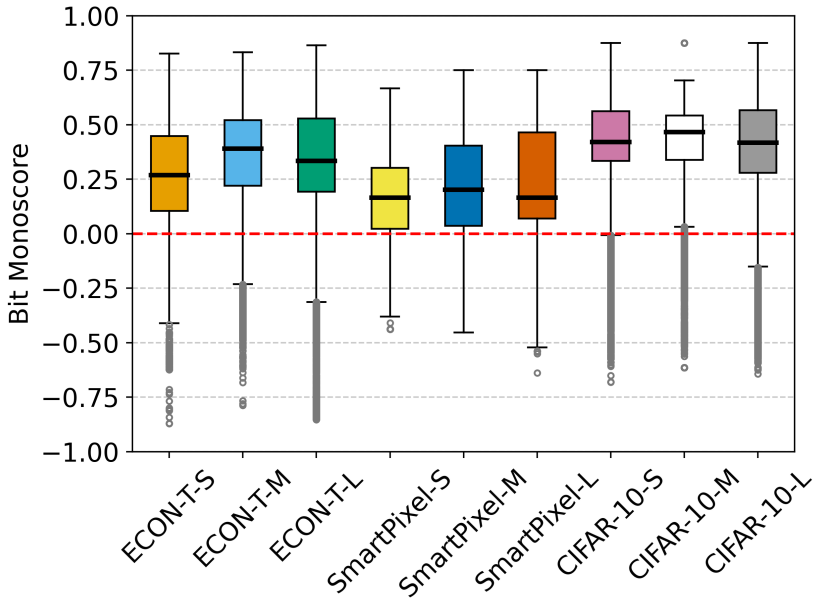


Figure 6.9. Distribution of the bit monoscores per model. None of the NNs have bits that are strictly monotonic (monoscore = 1) or strictly non-monotonic (monoscore = -1). The SmartPixel NNs have the lowest median monoscores, meaning they are the most non-monotonic compared with the other models, whereas the CIFAR-10 NNs have the highest median monoscores, indicating they are the most monotonic. PrioriFI is thus better at finding the sensitive bits in the SmartPixel NNs compared with the CIFAR-10 NNs.

the bit monoscores of an NN's bits like so:

$$\text{model monoscore} = \frac{1}{n}\sum_{i=0}^{n}\text{monoscore}(b_i) \tag{6.5}$$

where n is the total number of NN bits. Similarly to the bit monoscore, a perfectly monotonic NN has a model monoscore of 1, a perfectly non-monotonic NN has a score of -1, and a NN that is equally monotonic and non-monotonic has a score of 0.

Given the monoscore defined above, we now analyze how monotonic the nine edge NNs are. We compute the monoscore for every bit in each edge NN and plot its distribution in Fig. 6.9. In this chart, we see that none of the NNs are perfectly monotonic or non-monotonic. The SmartPixel NNs have the lowest median monoscores and lowest model monoscores (Tab. 6.4), suggesting that they are the most non-monotonic compared with the other NNs. As such,

Table 6.4. Model monoscores. The CIFAR-10 NNs are the most monotonic, and the SmartPixel NNs are the least monotonic, so PrioriFI is more effective on the SmartPixel NNs than the CIFAR-10 NNs.

Model	Model monoscore
ECON-T-S	0.26
ECON-T-M	0.35
ECON-T-L	0.32
SmartPixel-S	0.16
SmartPixel-M	0.22
SmartPixel-L	0.25
CIFAR-10-S	0.44
CIFAR-10-M	0.43
CIFAR-10-L	0.40

PrioriFI significantly improves the SmartPixel FI campaigns compared to prior work (Tab. 6.3), successfully adjusting for the exceptions to bit-level monotonicity, as seen in Fig. 6.8. The CIFAR-10 NNs have the highest median monoscores and the highest model monoscores (Tab. 6.4), so they are the most monotonic compared to the other NNs. As a result, PrioriFI is unable to significantly improve the CIFAR-10 FI campaigns because there is not a lot of non-montonic behavior to compensate for (Fig. 6.8). PrioriFI's performance on the ECON-T NNs splits the difference between SmartPixel and CIFAR-10 NNs, which corresponds to how the ECON-T monoscores (Fig. 6.9 and Tab. 6.4) similarly split the difference. Whether a model violates bit-level monotonicity a lot or not depends on each individual model's idiosyncrasies, which is hard to predict. Nevertheless, PrioriFI shines when there are many exceptions to bit-level monotonicity, especially at the inter-parameter level.

Saving time with PrioriFI

We quantify the FI time saved using PrioriFI in Fig. 6.10. In this study, we measure how much time it takes for each ranking to find the bits that contribute to 25%, 50%, 75%, and 100% of the total cumulative ΔC in the medium-sized models. PrioriFI saves FI time because it can find the most sensitive bits in less time than prior work. For instance, PrioriFI finds 50%

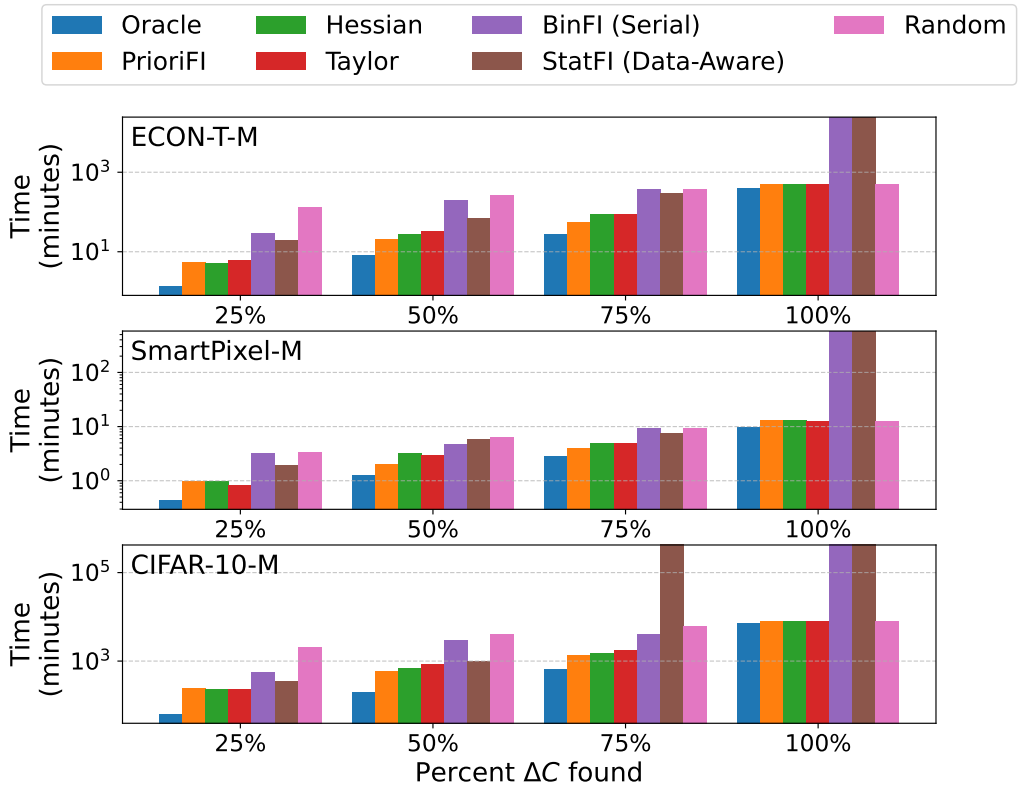


Figure 6.10. PrioriFI time savings. PrioriFI saves FI time, finding the bits contributing to 25/50/75% of the NN's sensitivity in less or comparable time compared to prior work. When the bar reaches the top of the y-axis and contains hatch marks, it means this strategy never finds that percentage of sensitive bits due to identifying false negatives.

of SmartPixel-M's cumulative ΔMispredicts 43% faster than the Hessian. In another instance, PrioriFI finds 50% of CIFAR-10-M's ΔMispredicts 14% faster than the Hessian, saving one hour and 37 minutes of time spent injecting faults. But all techniques struggle to find the "tails," spending much more time after finding 75% of the sensitive bits to find the remaining ones. Since StatFI and BinFI are unable to identify all of the sensitive bits, their bars sometimes reach the top of the chart and contain hatch marks, meaning they can never find the bits contributing a given percentage of the ΔC. This shows how StatFI and BinFI suffer from false negatives.

We also compute how much overhead is incurred from computing the Hessian and flipping the first bit in each of PrioriFI's bit lists, i.e., steps 1 and 2 in Fig. 6.4, in Tab. 6.5. Note that the initial bit flips in step 2 necessarily occur in any FI campaign that flips all of a model's

Table 6.5. Hessian and initial FI overhead. The overhead incurred from computing the Hessian and injecting the first bit in all of the bit lists (steps 1 and 2 in Fig. 6.4) is minimal compared to the time it takes to execute the full FI campaign.

Model	Hessian (sec.)	Initial FIs (sec.)	FI Campaign overhead (%)
ECON-T-S	20.2	15.8	0.17
ECON-T-M	11.7	13.9	0.09
ECON-T-L	13.5	16.5	0.02
SmartPixel-S	28.7	0.564	16
SmartPixel-M	23.6	0.748	3.2
SmartPixel-L	34.3	0.760	0.53
CIFAR-10-S	148.1	5.21	0.07
CIFAR-10-M	207.6	8.40	0.04
CIFAR-10-L	496.1	12.6	0.03

bits. We observe that the overhead PrioriFI incurs from computing the Hessian and flipping the initial bits is minimal compared with the entire FI campaign time. Most models incur less than 1 percent overhead, if not less than half a percent. Only SmartPixel-S and SmartPixel-M incur a higher percentage because there are so few bits in the model to begin with (825 and 3 956, respectively). Even so, the FI campaign itself dominates a significant portion of the time, leading to PrioriFI's significant time savings in finding the most sensitive bits first as previously shown in Fig. 6.10.

Measuring false positives/negatives

BinFI and StatFI suffer from incorrectly identifying bits as sensitive (false positives) or insensitive (false negatives) as seen in Tab. 6.6. StatFI suffers from these errors because it computes a subset of all NN model bits to sample in its FI campaign. This sample size calculation is based on large changes in magnitude. However, as shown in Sec. 6.3.2, magnitude changes from bit flips can be masked by operations that occur downstream in an NN. Moreover, these large changes in magnitude occur more often in floating point, which has a large range in the values it represents. Because fixed-point datatypes have such a small range, large swings in

Table 6.6. False Positives and False Negatives in FI algorithms. BinFI and StatFI suffer from incorrectly identifying bits as sensitive (false positives) or insensitive (false negatives). PrioriFI does not suffer from false positives or negatives. (A + I) = Actual + Implied. (D-A) = Data-Aware.

NN	FI Method	# False Positives (%)	# False Negatives (%)
ECON-T-M	BinFI (A + I)	210 (1.6%)	669 (5.2%)
	StatFI (D-A)	889 (6.9%)	4 021 (31.3%)
	PrioriFI	**0 (0%)**	**0 (0%)**
SmartPixel-M	BinFI (A + I)	28 (0.7%)	28 (0.7%)
	StatFI (D-A)	494 (12.5%)	1 021 (25.8%)
	PrioriFI	**0 (0%)**	**0 (0%)**
CIFAR10-M	BinFI (A + I)	1 109 (0.2%)	4 024 (0.9%)
	StatFI (D-A)	2 699 (0.6%)	306 445 (66.5%)
	PrioriFI	**0 (0%)**	**0 (0%)**

magnitude do not occur. Thus, StatFI (Data-Aware)'s reliance on magnitude change to determine its statistical FI sampling is flawed for the quantized representation present in edge NNs. BinFI encounters these false positive/negative errors because it relies on bit-level monotonicity to infer the sensitivity of bits. Although the rates are low, missing these amounts of sensitive bits can have a significant impact on resource constraints on edge NNs when considering how many resources can be dedicated to protecting the sensitive bits. PrioriFI flips all bits from high sensitivity to low, so it does not suffer from false positives or false negatives.

6.4.3 Discussion

Our evaluation of PrioriFI mostly focuses on vision-style NNs that are likely to be exposed to high radiation. Note, however, that the ECON-T is an autoencoder focused on data compression and the SmartPixel operates on high-energy clusters from sensor data. Given this variety of data representations in our evaluation, PrioriFI should apply well to other edge workloads beyond the ones studied in this work. Any NN that can be measured by the Hessian can be evaluated with PrioriFI; however, how well PrioriFI improves a NN's FI campaign

depends on the model's monotonicity, as seen in our model monoscore evaluation in Sec. 6.4.2.

It would be ideal to provide a tight bound on how much PrioriFI can improve a NN's FI campaign, but this is challenging, especially given the lack of monotonicity in the models (Fig. 6.9). However, our work does compare to the optimal FI performance (i.e., the Oracle), demonstrating that we are closer to it than prior state-of-the-art work (Tab. 6.3). By generating the Oracle through exhaustive FI analysis for each of the nine NNs, our experiments show how close to the Oracle PrioriFI gets on a variety of NNs in Tab. 6.3, with AUCs of 0.93 and above. But there is still room for improvement, as PrioriFI does not achieve an AUC of 1, demonstrating some remaining inefficiencies for future work to discover.

One of the key challenges in getting closer to the Oracle's performance is the variability in the monotonic behavior of a NN's bits. As discussed in Sec. 6.4.2, the bits in a NN are neither uniformly monotonic nor non-monotonic. Some bits are more (non-)monotonic than others. One thought is to use the monoscore in PrioriFI's heuristic for addressing non-monotonic behavior. The difficulty is that we can only compute a bit's monoscore after computing the full FI campaign—we need to know each bit's ΔC first. Instead, future work should seek to develop a heuristic (or some other metric) that better captures the subtleties of the monoscore and a NN's variable non-monotonic behavior than PrioriFI's current `medianLastK()` heuristic. In fact, PrioriFI could likely benefit from a better heuristic to build on or to replace `medianLastK()`.

6.5 Summary

PrioriFI is a more informed FI algorithm for edge NNs. It uses information gained during an FI campaign to dynamically determine which bit to flip next, from highest sensitivity to lowest. PrioriFI is generally effective at identifying sensitive bits faster in the nine edge NNs that we study in this chapter. With PrioriFI, designers can quickly evaluate the robustness of different NN architectures under strict latency and resource constraints, codesigning fault-tolerant edge NNs faster and in a more informed way.

Chapter 6, in part, is a reprint of the material as it appears in Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26). The work in this chapter was done in collaboration with Andres Meza, Nhan Tran, and Ryan Kastner. The dissertation author was the primary investigator and author of this work.

Chapter 7

Conclusion

In this dissertation, I have introduced several techniques for building efficient and resilient computer architectures for edge NN inference. I have taken a hardware-first approach to designing efficient architectures and a software-first approach to evaluating the resilience of NNs to faults. I defend the following thesis statement: *On-chip neural network inference introduces unique hardware-software codesign challenges and opportunities for building efficient, fault-tolerant neural network architectures.*

In Chapter 3, I introduced Tailor, a hardware-aware training algorithm that removes or shortens skip connections in NNs for more resourece-efficient inference on FPGAs. I demonstrate the effectiveness of Tailor on both on- and off-chip inference, showing that it can reduce resources for on-chip inference and latency for off-chip inference while maintaining accuracy. Tailor improves resource utilization by up to 34% for BRAMs, 13% for FFs, and 16% for LUTs for on-chip, dataflow-style architectures. Tailor increases performance by 30% and reduces memory bandwidth by 45% for a 2D processing element array architecture.

In Chapter 4, I introduced AmigoLUT, a method for scaling up LUT-based NN architectures through ensemble learning. AmigoLUT trains multiple smaller LUT-based NNs and builds an ensemble out of them to achieve higher accuracy without exponentially increasing LUTs. AmigoLUT improves the scalability of LUT-based NNs, reaching higher throughput with up to an order of magnitude fewer LUTs than the largest LUT-based NNs.

In Chapter 5, I introduced FKeras, a tool for assessing the sensitivity of NN weights to faults and guiding efficient fault injection campaigns. FKeras uses a Hessian-based sensitivity metric to determine the most sensitive bits in the NN weights and guides fault injection campaigns to consider those bits first. With FKeras, I found that different NN architectures have vastly differing resilience to faults.

In Chapter 6, I introduced PrioriFI, an improvement on FKeras that avoids the pitfalls of relying on bit-level monotonicity during a fault injection campaign, prioritizing the most sensitive bits. I demonstrate that many edge NN parameter bits do not exhibit strictly monotonic behavior, and making this assumption to speedup fault injection campaigns leads to an inefficient search for sensitive bits. PrioriFI uses past fault injection results as a heuristic to guide future fault injections in the campaign, which allows it to adapt to the non-monotonic behavior of bit-level sensitivity in NNs.

I believe the insights from my dissertation will be valuable for future research. My contributions to hardware-software codesign demonstrates how it is important to consider the hardware architecture during the training process and the NN software during the hardware design process to achieve efficient on-chip inference. Expertise of both hardware and software is necessary, and my belief is that my tools like Tailor and FKeras will help make hardware-software codesign easier for future researchers. I also believe that designing for fault tolerance is an important aspect of edge NN design, as researchers continue to deploy NNs in more safety-critical and high-radiation environments. With PrioriFI, I have made it easier to assess the fault tolerance of edge NNs and provide insights into the inefficiencies of relying on bit-level monotonicity in past work. PrioriFI is one way of avoiding this misconception. I hope researchers take this insight with them in their future work. Hardware-software codesign is more critical than ever as we continue to push the boundaries of edge NN efficiency and resilience, especially as AI becomes more integrated with humanity. I hope my dissertation eases the path for future research in this area.

Bibliography

[1] 2012. Vivado design suite user guide. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_2/ug902-vivado-high-level-synthesis.pdf

[2] 2023. Tailor. https://github.com/oliviaweng/tailor.

[3] Thea Aarrestad, Vladimir Loncar, Nicolò Ghielmetti, Maurizio Pierini, Sioni Summers, Jennifer Ngadiuba, Christoffer Petersson, Hampus Linander, Yutaro Iiyama, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Dylan Rankin, Sergo Jindariani, Kevin Pedro, Nhan Tran, Mia Liu, Edward Kreinar, Zhenbin Wu, and Duc Hoang. 2021. Fast convolutional neural networks on FPGAs with hls4ml. *Machine Learning: Science and Technology* 2, 4 (jul 2021), 045015. https://doi.org/10.1088/2632-2153/ac0ea1

[4] G. Abarajithan, Zhenghua Ma, Ravindu Munasinghe, Francesco Restuccia, and Ryan Kastner. 2026. CGRA4ML: A Hardware/Software Framework to Implement Neural Networks for Scientific Edge Computing. *ACM Trans. Reconfigurable Technol. Syst.* (March 2026). https://doi.org/10.1145/3801097 Just Accepted.

[5] Mohammad Hasan Ahmadilivani, Mahdi Taheri, Jaan Raik, Masoud Daneshtalab, and Maksim Jenihhin. 2024. A Systematic Literature Review on Hardware Reliability Assessment Methods for Deep Neural Networks. *ACM Comput. Surv.* 56, 6, Article 141 (Jan. 2024), 39 pages. https://doi.org/10.1145/3638242

[6] Qeethara Kadhim Al-Shayea. 2011. Artificial neural networks in medical diagnosis. *International Journal of Computer Science Issues* 8, 2 (2011), 150–154.

[7] Syed Asad Alam, David Gregg, Giulio Gambardella, Thomas Preusser, and Michaela Blott. 2023. On the RTL Implementation of FINN Matrix Vector Unit. *ACM Trans. Embed. Comput. Syst.* 22, 6, Article 94 (Nov. 2023), 27 pages. https://doi.org/10.1145/3547141

[8] Igor Aleksander, WV Thomas, and PA Bowden. 1984. WISARD· a radical step forward in image recognition. *Sensor review* 4, 3 (1984), 120–124.

[9] Amazon Web Services. 2019. *AWS Inferentia Architecture and the Neuron SDK*. Amazon Web Services, Inc. https://aws.amazon.com/machine-learning/inferentia/ Technical

Documentation.

[10] Marta Andronic and George A Constantinides. 2023. PolyLUT: Learning Piecewise Polynomials for Ultra-Low Latency FPGA LUT-based Inference. In *2023 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 60–68.

[11] Marta Andronic and George A Constantinides. 2024. NeuraLUT: Hiding Neural Network Density in Boolean Synthesizable Functions. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 140–148.

[12] Marta Andronic and George A. Constantinides. 2025. NeuraLUT-Assemble: Hardware-Aware Assembling of Sub-Neural Networks for Efficient LUT Inference. In *2025 IEEE 33rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 208–216. https://doi.org/10.1109/FCCM62733.2025.00077

[13] Syed Muhammad Anwar, Muhammad Majid, Adnan Qayyum, Muhammad Awais, Majdi Alnowami, and Muhammad Khurram Khan. 2018. Medical image analysis using convolutional neural networks: a review. *Journal of medical systems* 42 (2018), 1–13.

[14] Luis Alberto Aranda, Alfonso Sánchez-Macián, and Juan Antonio Maestro. 2019. ACME: A Tool to Improve Configuration Memory Fault Injection in SRAM-Based FPGAs. *IEEE Access* 7 (2019), 128153–128161. https://doi.org/10.1109/ACCESS.2019.2939858

[15] Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, Ronald F. DeMara, Chen-Yong Cher, and Pradip Bose. 2015. Understanding the propagation of transient errors in HPC applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. Association for Computing Machinery, New York, NY, USA, Article 72, 12 pages. https://doi.org/10.1145/2807591.2807670

[16] Haim Avron and Sivan Toledo. 2011. Randomized algorithms for estimating the trace of an implicit symmetric positive semi-definite matrix. *Journal of the ACM (JACM)* 58, 2 (2011), 1–34.

[17] Alan Tendler Leibel Bacellar, Zachary Susskind, Mauricio Breternitz Jr, Eugene John, Lizy Kurian John, Priscila Machado Vieira Lima, and Felipe MG França. 2024. Differentiable Weightless Neural Networks. In *Forty-first International Conference on Machine Learning*.

[18] Colby R. Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Király, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, Urmish Thakker, Antonio Torrini, Pete Warden, Jay Cordaro, Giuseppe Di Guglielmo, Javier M. Duarte, Stephen Gibellini, Videet Parekh, Honson Tran, Nhan Tran, Wenxu Niu, and Xuesong Xu. 2021. MLPerf Tiny Benchmark. *CoRR* abs/2106.07597 (2021). arXiv:2106.07597

https://arxiv.org/abs/2106.07597

[19] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation. *arXiv:1308.3432 [cs]* (Aug. 2013). http://arxiv.org/abs/1308.3432 arXiv: 1308.3432.

[20] Y. Bengio, P. Simard, and P. Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Netw.* 5, 2 (March 1994), 157. https://doi.org/10.1109/72.279181

[21] Christoph Berganski, Felix Jentzsch, Marco Platzner, Max Kuhmichel, and Heiner Giefers. 2024. FINN-T: Compiling Custom Dataflow Accelerators for Quantized Transformers. In *2024 International Conference on Field Programmable Technology (ICFPT)*. 01–10. https://doi.org/10.1109/ICFPT64416.2024.11113391

[22] Timoteo García Bertoa, Giulio Gambardella, Nicholas J. Fraser, Michaela Blott, and John McAllister. 2023. Fault-Tolerant Neural Network Accelerators With Selective TMR. *IEEE Design & Test* 40, 2 (2023), 67–74. https://doi.org/10.1109/MDAT.2022.3174181

[23] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfigurable Technol. Syst.* 11, 3, Article 16 (Dec. 2018), 23 pages. https://doi.org/10.1145/3242897

[24] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016). arXiv:1604.07316 http://arxiv.org/abs/1604.07316

[25] Giulio Borghello. 2020. Ionizing radiation effects on 28 nm CMOS technology. *CERN report "TID effects 28 nm"* (2020).

[26] S. Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (2005), 10–16. https://doi.org/10.1109/MM.2005.110

[27] Hendrik Borras, Giuseppe Di Guglielmo, Javier Duarte, Nicolò Ghielmetti, Ben Hawks, Scott Hauck, Shih-Chieh Hsu, Ryan Kastner, Jason Liang, Andres Meza, Jules Muhizi, Tai Nguyen, Rushil Roy, Nhan Tran, Yaman Umuroglu, Olivia Weng, Aidan Yokuda, and Michaela Blott. 2022. Open-source FPGA-ML codesign for the MLPerf Tiny Benchmark. (2022). arXiv:2206.11791 [cs.LG] https://arxiv.org/abs/2206.11791

[28] Leo Breiman. 1996. Bagging predictors. *Machine learning* 24 (1996), 123–140.

[29] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. 33 (2020), 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[30] Randal E. Bryant and David R. O'Hallaron. 2011. *Computer systems: a programmer's perspective* (2nd ed ed.). Prentice Hall, Boston. OCLC: ocn457156657.

[31] Simon Burton, Lydia Gauerhof, and Christian Heinzemann. 2017. Making the case for safety of machine learning in highly automated driving. In *Computer Safety, Reliability, and Security: SAFECOMP 2017 Workshops, ASSURE, DECSoS, SASSUR, TELERISE, and TIPS, Trento, Italy, September 12, 2017, Proceedings 36*. Springer, 5–16.

[32] Javier Campos, Zhen Dong, Javier Duarte, Amir Gholami, Michael W Mahoney, Jovan Mitrevski, and Nhan Tran. 2023. End-to-end codesign of Hessian-aware quantized neural networks for FPGAs and ASICs. *arXiv preprint arXiv:2304.06745* (2023).

[33] Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto, and Lenka Zdeborová. 2019. Machine learning and the physical sciences. *Reviews of Modern Physics* 91, 4 (2019), 045002.

[34] Sung-En Chang, Yanyu Li, Mengshu Sun, Runbin Shi, Hayden K-H So, Xuehai Qian, Yanzhi Wang, and Xue Lin. 2021. Mix and Match: A novel FPGA-centric deep neural network quantization framework. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 208. https://doi.org/10.1109/HPCA51647.2021.00027 arXiv:2012.04240

[35] Odysseas Chatzopoulos, George Papadimitriou, Vasileios Karakostas, and Dimitris Gizopoulos. 2024. Gem5-MARVEL: Microarchitecture-Level Resilience Analysis of Heterogeneous SoC Architectures. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 543–559. https://doi.org/10.1109/HPCA57654.2024.00047

[36] Arjun Chaudhuri, Ching-Yuan Chen, Jonti Talukdar, Siddarth Madala, Abhishek Kumar Dubey, and Krishnendu Chakrabarty. 2021. Efficient Fault-Criticality Analysis for AI Accelerators using a Neural Twin. In *2021 IEEE International Test Conference (ITC)*. 73–82. https://doi.org/10.1109/ITC50571.2021.00015

[37] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-

Learning Supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 609–622. https://doi.org/10.1109/MICRO.2014.58

[38] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[39] Zitao Chen, Guanpeng Li, and Karthik Pattabiraman. 2021. A low-cost fault corrector for deep neural networks through range restriction. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 1–13.

[40] Zitao Chen, Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2019. BinFI: an efficient fault injector for safety-critical machine learning systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 69, 23 pages. https://doi.org/10.1145/3295500.3356177

[41] Eric Cheng, Shahrzad Mirkhani, Lukasz G. Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R. Stan, Klas Lilja, Jacob A. Abraham, Pradip Bose, and Subhasish Mitra. 2016. CLEAR: Cross-Layer Exploration for Architecting Resilience - Combining hardware and software techniques to tolerate soft errors in processor cores. In *Proceedings of the 53rd Annual Design Automation Conference* (Austin, Texas) *(DAC '16)*. Association for Computing Machinery, New York, NY, USA, Article 68, 6 pages. https://doi.org/10.1145/2897937.2897996

[42] Quan Cheng, Wang Liao, Ruilin Zhang, Hao Yu, Longyang Lin, and Masanori Hashimoto. 2025. HachiFI: A Lightweight SoC Architecture-Independent Fault-Injection Framework for SEU Impact Evaluation. In *2025 Design, Automation & Test in Europe Conference (DATE)*. 1–7. https://doi.org/10.23919/DATE64628.2025.10993139

[43] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I.-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. PACT: Parameterized Clipping Activation for Quantized Neural Networks. *arXiv:1805.06085 [cs]* (July 2018). http://arxiv.org/abs/1805.06085 arXiv: 1805.06085.

[44] Wonseok Choi, Dongyeob Shin, Jongsun Park, and Swaroop Ghosh. 2019. Sensitivity based Error Resilient Techniques for Energy Efficient Deep Neural Network Accelerators. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 204, 6 pages. https://doi.org/10.1145/3316781.3317908

[45] Claudionor N. Coelho, Aki Kuusela, Shan Li, Hao Zhuang, Thea Aarrestad, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Adrian Alan Pol, and Sioni Summers.

2021. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Mach. Intell.* 3 (2021), 675–686. https://doi.org/10.1038/s42256-021-00356-5 arXiv:2006.10159 [physics.ins-det]

[46] Leo Coic. 2023. Single Event Effects.. In *Radiation and its Effects on Microelectronics and Photonics Technologies Workshop*.

[47] The CMS collaboration. 2017. The phase-2 upgrade of the CMS endcap calorimeter. *CMS Technical Design Report CERN-LHCC-2017-023. CMS-TDR-019, CERN* (2017).

[48] Alessio Colucci, Andreas Steininger, and Muhammad Shafique. 2022. enpheeph: A Fault Injection Framework for Spiking and Compressed Deep Neural Networks. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 5155–5162. https://doi.org/10.1109/IROS47612.2022.9982181

[49] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[50] Allison McCarn Deiana, Nhan Tran, Joshua Agar, Michaela Blott, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Scott Hauck, Mia Liu, Mark S. Neubauer, Jennifer Ngadiuba, Seda Ogrenci-Memik, Maurizio Pierini, Thea Aarrestad, Steffen Bähr, Jürgen Becker, Anne-Sophie Berthold, Richard J. Bonventre, Tomás E. Müller Bravo, Markus Diefenthaler, Zhen Dong, Nick Fritzsche, Amir Gholami, Ekaterina Govorkova, Dongning Guo, Kyle J. Hazelwood, Christian Herwig, Babar Khan, Sehoon Kim, Thomas Klijnsma, Yaling Liu, Kin Ho Lo, Tri Nguyen, Gianantonio Pezzullo, Seyedramin Rasoulinezhad, Ryan A. Rivera, Kate Scholberg, Justin Selig, Sougata Sen, Dmitri Strukov, William Tang, Savannah Thais, Kai Lukas Unger, Ricardo Vilalta, Belina von Krosigk, Shen Wang, and Thomas K. Warburton. 2022. Applications and Techniques for Fast Machine Learning in Science. *Frontiers in Big Data* Volume 5 - 2022 (2022). https://doi.org/10.3389/fdata.2022.787421

[51] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. 248. https://doi.org/10.1109/CVPR.2009.5206848

[52] Xiaohan Ding, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. 2021. RepVGG: Making VGG-style convnets great again. In *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 13733. https://doi.org/10.1109/CVPR46437.2021.01352 arXiv:2101.03697

[53] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245* (2021).

[54] Xibin Dong, Zhiwen Yu, Wenming Cao, Yifan Shi, and Qianli Ma. 2020. A survey on ensemble learning. *Frontiers of Computer Science* 14 (2020), 241–258.

[55] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. *Advances in neural information processing systems* 33 (2020), 18518–18529.

[56] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2019. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 293–302.

[57] Fernando Fernandes dos Santos, Caio Lunardi, Daniel Oliveira, Fabiano Libano, and Paolo Rech. 2019. Reliability evaluation of mixed-precision architectures. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 238–249.

[58] Petros Drineas and Michael W Mahoney. 2018. Lectures on randomized numerical linear algebra. *The Mathematics of Data* 25, 1 (2018).

[59] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: shifting vision processing closer to the sensor. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 92–104. https://doi.org/10.1145/2872887.2750389

[60] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* 13, 07 (2018), P07027.

[61] Javier Duarte, Nhan Tran, Ben Hawks, Christian Herwig, Jules Muhizi, Shvetank Prakash, and Vijay Janapa Reddi. 2022. FastML Science Benchmarks: Accelerating Real-Time Scientific Edge Machine Learning. *arXiv preprint arXiv:2207.07958* (2022).

[62] Christer Ericson. 2005. Chapter 11 - Numerical Robustness. In *Real-Time Collision Detection*, Christer Ericson (Ed.). Morgan Kaufmann, San Francisco, 427–463. https://doi.org/10.1016/B978-1-55860-732-3.50016-4

[63] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P. Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, Dylan Rankin, Manuel Blanco Valentin, Josiah Hester, Yingyi Luo, John Mamish, Seda Orgrenci-Memik, Thea Aarrestad, Hamza Javed, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, Javier Duarte, Scott Hauck, Shih-Chieh Hsu, Jennifer Ngadiuba, Mia Liu, Duc Hoang, Edward Kreinar, and Zhenbin Wu. 2021. hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices. arXiv:2103.05579 [cs.LG] https://arxiv.org/abs/2103.05579

[64] Angela Fan, Pierre Stock, Benjamin Graham, Edouard Grave, Remi Gribonval, Herve Jegou, and Armand Joulin. 2021. Training with Quantization Noise for Extreme Model Compression. *arXiv:2004.07320 [cs, stat]* (Feb. 2021). http://arxiv.org/abs/2004.07320 arXiv: 2004.07320.

[65] Leopoldo Lusquino Filho, Felipe MG França, and Priscila MV Lima. 2023. WiSARD-based Ensemble Learning. (2023).

[66] Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, Olivia Wu, Gurdeepak Grewal, Harish Aepala, Bhasker Jakka, Bob Dreyer, Adam Hutchin, Utku Diril, Krishnakumar Nair, Ehsan K. Aredestani, Martin Schatz, Yuchen Hao, Rakesh Komuravelli, Kunming Ho, Sameer Abu Asal, Joe Shajrawi, Kevin Quinn, Nagesh Sreedhara, Pankaj Kansal, Willie Wei, Dheepak Jayaraman, Linda Cheng, Pritam Chopda, Eric Wang, Ajay Bikumandla, Arun Karthik Sengottuvel, Krishna Thottempudi, Ashwin Narasimha, Brian Dodds, Cao Gao, Jiyuan Zhang, Mohammed Al-Sanabani, Ana Zehtabioskuie, Jordan Fix, Hangchen Yu, Richard Li, Kaustubh Gondkar, Jack Montgomery, Mike Tsai, Saritha Dwarakapuram, Sanjay Desai, Nili Avidan, Poorvaja Ramani, Karthik Narayanan, Ajit Mathews, Sethu Gopal, Maxim Naumov, Vijay Rao, Krishna Noru, Harikrishna Reddy, Prahlad Venkatapuram, and Alexis Bjorlin. 2023. MTIA: First Generation Silicon Targeting Meta's Recommendation Systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 80, 13 pages. https://doi.org/10.1145/3579371.3589348

[67] Yoav Freund and Robert E Schapire. 1996. Experiments with a new boosting algorithm. In *icml*, Vol. 96. Citeseer, 148–156.

[68] Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig. 2017. *Deep learning with INT8 optimization on Xilinx devices*. White Paper WP486. https://docs.xilinx.com/v/u/en-US/wp486-deep-learning-int8

[69] Giulio Gambardella, Johannes Kappauf, Michaela Blott, Christoph Doehring, Martin Kumm, Peter Zipf, and Kees Vissers. 2019. Efficient Error-Tolerant Quantized Neural Network Accelerators. In *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 1–6. https://doi.org/10.1109/DFT.2019.8875314

[70] Rubén García Alía, Markus Brugger, Francesco Cerutti, Salvatore Danzeca, Alfredo Ferrari, Simone Gilardoni, Yacine Kadi, Maria Kastriotou, Anton Lechner, Corinna Martinella, Oliver Stein, Yves Thurel, Andrea Tsinganis, and Slawosz Uznanski. 2018. LHC and HL-LHC: Present and Future Radiation Environment in the High-Luminosity Collision Points and RHA Implications. *IEEE Transactions on Nuclear Science* 65, 1 (2018), 448–456. https://doi.org/10.1109/TNS.2017.2776107

[71] Jonas Gava, Alex Hanneman, Geancarlo Abich, Rafael Garibotti, Sergio Cuenca-Asensi, Rodrigo Possamai Bastos, Ricardo Reis, and Luciano Ost. 2023. A Lightweight Mitigation Technique for Resource- Constrained Devices Executing DNN Inference Models Under Neutron Radiation. *IEEE Transactions on Nuclear Science* 70, 8 (2023), 1625–1633. https://doi.org/10.1109/TNS.2023.3262448

[72] Jonas Gava, Ricardo Reis, and Luciano Ost. 2021. RAT: A Lightweight Architecture Independent System-Level Soft Error Mitigation Technique. In *VLSI-SoC: Design Trends*, Andrea Calimera, Pierre-Emmanuel Gaillardon, Kunal Korgaonkar, Shahar Kvatinsky, and Ricardo Reis (Eds.). Springer International Publishing, Cham, 235–253.

[73] Johannes Geier and Daniel Mueller-Gritschneder. 2023. vRTLmod: An LLVM based Open-source Tool to Enable Fault Injection in Verilator RTL Simulations. In *Proceedings of the 20th ACM International Conference on Computing Frontiers* (Bologna, Italy) *(CF '23)*. Association for Computing Machinery, New York, NY, USA, 387–388. https://doi.org/10.1145/3587135.3591435

[74] Nicolò Ghielmetti, Vladimir Loncar, Maurizio Pierini, Marcel Roed, Sioni Summers, Thea Aarrestad, Christoffer Petersson, Hampus Linander, Jennifer Ngadiuba, Kelvin Lin, and Philip Harris. 2022. Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml. *Machine Learning: Science and Technology* 3, 4 (nov 2022), 045011. https://doi.org/10.1088/2632-2153/ac9cb5

[75] Luigi Ghionda, Riccardo Tedeschi, Yvan Tortorella, Arpan Suravi Prasad, Davide Rossi, Luca Benini, and Francesco Conti. 2025. HMR-NEureka: Hybrid Modular Redundancy DNN Acceleration in Heterogeneous RISC-V SoCs. In *2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Vol. 1. 1–6. https://doi.org/10.1109/ISVLSI65124.2025.11130209

[76] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630* (2021).

[77] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. *arXiv:2103.13630 [cs]* (June 2021). http://arxiv.org/abs/2103.13630 arXiv: 2103.13630.

[78] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proc. 13th International Conference on Artificial Intelligence and Statistics*, Yee Whye Teh and Mike Titterington (Eds.), Vol. 9. 249. https://proceedings.mlr.press/v9/glorot10a.html

[79] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep Sparse Rectifier Neural Networks. In *Proc. 14th International Conference on Artificial Intelligence and Statistics*,

Geoffrey Gordon, David Dunson, and Miroslav Dudík (Eds.), Vol. 15. 315. http://proceedings.mlr.press/v15/glorot11a.html

[80] Google. 2019. qkeras. https://github.com/google/qkeras Accessed 2025-08-21.

[81] Giuseppe Di Guglielmo, Farah Fahim, Christian Herwig, Manuel Blanco Valentin, Javier Duarte, Cristian Gingu, Philip Harris, James Hirschauer, Martin Kwok, Vladimir Loncar, Yingyi Luo, Llovizna Miranda, Jennifer Ngadiuba, Daniel Noonan, Seda Ogrenci-Memik, Maurizio Pierini, Sioni Summers, and Nhan Tran. 2021. A Reconfigurable Neural Network ASIC for Detector Front-End Data Compression at the HL-LHC. *IEEE Transactions on Nuclear Science* 68, 8 (2021), 2179–2186. https://doi.org/10.1109/TNS.2021.3087100

[82] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International conference on machine learning*. PMLR, 1737–1746.

[83] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[84] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. 2009. Multi-class adaboost. *Statistics and its Interface* 2, 3 (2009), 349–360.

[85] Benjamin Hawks, Jason Weitz, Dmitri Demler, Karla Tame-Narvaez, Dennis Plotnikov, Mohammad Mehdi Rahimifar, Hamza Ezzaoui Rahali, Audrey C. Therrien, Donovan Sproule, Elham E Khoda, Keegan A. Smith, Russell Marroquin, Giuseppe Di Guglielmo, Nhan Tran, Javier Duarte, and Vladimir Loncar. 2025. wa-hls4ml: A Benchmark and Surrogate Models for hls4ml Resource and Latency Estimation. (2025). arXiv:2511.05615 [cs.LG] https://arxiv.org/abs/2511.05615

[86] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770. https://doi.org/10.1109/CVPR.2016.90 arXiv:1512.03385

[87] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity Mappings in Deep Residual Networks. In *ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). 630. https://doi.org/10.1007/978-3-319-46493-0_38 arXiv:1603.05027

[88] Yi He, Prasanna Balaprakash, and Yanjing Li. 2020. FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 270–281. https://doi.org/10.1109/MICRO50266.2020.00033

[89] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural

Network. (2015). arXiv:1503.02531

[90] Duc Hoang, Aarush Gupta, and Philip C Harris. 2026. KANELÉ: Kolmogorov–Arnold Networks for Efficient LUT-based Evaluation. In *Proceedings of the 2026 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (USA) *(FPGA '26)*. Association for Computing Machinery, New York, NY, USA, 44–55. https://doi.org/10.1145/3748173.3779202

[91] Le-Ha Hoang, Muhammad Abdullah Hanif, and Muhammad Shafique. 2020. Ft-clipact: Resilience analysis of deep neural networks and improving their fault tolerance using clipped activation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1241–1246.

[92] Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. 2021. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 9–16.

[93] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proc. IEEE conference on computer vision and pattern recognition*. 4700. https://doi.org/10.1109/CVPR.2017.243 arXiv:1608.06993

[94] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. 2021. Accurate Post Training Quantization With Small Calibration Sets. In *Proceedings of the 38th International Conference on Machine Learning*. 10.

[95] IEEE 2008. *Intermittent faults and effects on reliability of integrated circuits*. IEEE.

[96] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proc. IEEE conference on computer vision and pattern recognition*. 2704. https://doi.org/10.1109/CVPR.2018.00286

[97] Felix Jentzsch, Yaman Umuroglu, Alessandro Pappalardo, Michaela Blott, and Marco Platzner. 2022. RadioML Meets FINN: Enabling Future RF Applications With FPGA Streaming Architectures. *IEEE Micro* 42, 6 (2022), 125–133. https://doi.org/10.1109/MM.2022.3202091

[98] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew,

Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 1–12. https://doi.org/10.1145/3140659.3080246

[99] Linus Jungemann, Bjarne Wintermann, Heinrich Riebler, and Christian Plessl. 2025. FINN-HPC: Closing the Gap for Energy-Efficient Neural Network Inference on FPGAs in HPC. In *Proceedings of the 15th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART '25)*. Association for Computing Machinery, New York, NY, USA, 103–116. https://doi.org/10.1145/3728179.3728189

[100] Leonardo Rezende Juracy, Rafael Garibotti, and Fernando Gehm Moraes. 2023. From CNN to DNN Hardware Accelerators: A Survey on Design, Exploration, Simulation, and Frameworks. *Foundations and Trends in Electronic Design Automation* 13, 4 (06 2023), 270–344. https://doi.org/10.1561/1000000060 arXiv:https://www.emerald.com/fteda/article-pdf/13/4/270/10902322/1000000060en.pdf

[101] Endri Kaja, Nicolas Ojeda Leon, Michael Werner, Bogdan Andrei-Tabacaru, Keerthikumara Devarajegowda, and Wolfgang Ecker. 2021. Extending Verilator to Enable Fault Simulation. In *MBMV 2021; 24th Workshop*. 1–6.

[102] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. 2018. Parallel programming for FPGAs. *arXiv preprint arXiv:1805.03648* (2018).

[103] Shashwat Khandelwal, Jakoba Petri-Koenig, Thomas B Preußer, Michaela Blott, and Shreejith Shanker. 2025. FINN-GL: Generalized Mixed-Precision Extensions for FPGA-Accelerated LSTMs. *arXiv preprint arXiv:2506.20810* (2025).

[104] Navid Khoshavi, Connor Broyles, Yu Bi, and Arman Roohi. 2020. Fiji-FIN: A Fault Injection Framework on Quantized Neural Network Inference Accelerator. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 1139–1144. https://doi.org/10.1109/ICMLA51294.2020.00183

[105] Navid Khoshavi, Arman Roohi, Connor Broyles, Saman Sargolzaei, Yu Bi, and David Z Pan. 2020. Shieldenn: Online accelerated framework for fault-tolerant deep neural network architectures. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[106] Samuel Kriman, Stanislav Beliaev, Boris Ginsburg, Jocelyn Huang, Oleksii Kuchaiev, Vitaly Lavrukhin, Ryan Leary, Jason Li, and Yang Zhang. 2020. QuartzNet: Deep automatic speech recognition with 1D time-channel separable convolutions. In *2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 6124. https://doi.org/10.1109/ICASSP40776.2020.9053889

[107] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

[108] J. Laurent, C. Deleuze, F. Pebay-Peyroula, and V. Beroulle. 2021. Bridging the Gap between RTL and Software Fault Injection. *J. Emerg. Technol. Comput. Syst.* 17, 3, Article 38 (May 2021), 24 pages. https://doi.org/10.1145/3446214

[109] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324. https://doi.org/10.1109/5.726791

[110] Régis Leveugle, A Calvez, Paolo Maistri, and Pierre Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 502–506.

[111] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W Keckler. 2017. 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*. 1–12.

[112] Guanpeng Li, Karthik Pattabiraman, and Nathan DeBardeleben. 2018. TensorFI: A Configurable Fault Injector for TensorFlow Applications. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 313–320. https://doi.org/10.1109/ISSREW.2018.00024

[113] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling Soft-Error Propagation in Programs. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 27–38. https://doi.org/10.1109/DSN.2018.00016

[114] Guilin Li, Junlei Zhang, Yunhe Wang, Chuanjian Liu, Matthias Tan, Yunfeng Lin, Wei Zhang, Jiashi Feng, and Tong Zhang. 2020. Residual distillation: Towards portable deep neural networks without shortcuts. *Advances in Neural Information Processing Systems* 33 (2020), 8935. https://proceedings.neurips.cc/paper/2020/file/657b96f0592803e25a4f07166fff289a-Paper.pdf

[115] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. Visualizing

the Loss Landscape of Neural Nets. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2018/file/a41b3bb3e6b050b6c9067c67f663b915-Paper.pdf

[116] Tuo Li, Jude Angelo Ambrose, Roshan Ragel, and Sri Parameswaran. 2016. Processor Design for Soft Errors: Challenges and State of the Art. *ACM Comput. Surv.* 49, 3, Article 57 (Nov. 2016), 44 pages. https://doi.org/10.1145/2996357

[117] Fabiano Libano, Brittany Wilson, J Anderson, Michael J Wirthlin, Carlo Cazzaniga, Christopher Frost, and Paolo Rech. 2018. Selective hardening for neural networks in FPGAs. *IEEE Transactions on Nuclear Science* 66, 1 (2018), 216–222.

[118] Hongzhou Lin and Stefanie Jegelka. 2018. Resnet with one-neuron hidden layers is a universal approximator. *Advances in neural information processing systems* 31 (2018).

[119] Binglei Lou, Richard Rademacher, David Boland, and Philip HW Leong. 2024. PolyLUT-Add: FPGA-based LUT Inference with Wide Inputs. *arXiv preprint arXiv:2406.04910* (2024).

[120] Haiquan Lu, Xiaotian Liu, Yefan Zhou, Qunli Li, Kurt Keutzer, Michael W. Mahoney, Yujun Yan, Huanrui Yang, and Yaoqing Yang. 2024. Sharpness-diversity tradeoff: improving flat ensembles with SharpBalance. arXiv:2407.12996 [stat.ML] https://arxiv.org/abs/2407.12996

[121] Dongning Ma, Fred Lin, Alban Desmaison, Joel Coburn, Daniel Moore, Sriram Sankar, and Xun Jiao. 2024. Dr. DNA: Combating Silent Data Corruptions in Deep Learning using Distribution of Neuron Activations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 239–252. https://doi.org/10.1145/3620666.3651349

[122] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet v2: Practical guidelines for efficient CNN architecture design. In *Proc. European conference on computer vision (ECCV)*. 116. https://doi.org/10.1109/ASPCON49795.2020.9276669 arXiv:1807.11164

[123] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2018. Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA. *IEEE Trans Very Large Scale Integr. VLSI Syst.* 26, 7 (2018), 1354. https://doi.org/10.1109/TVLSI.2018.2815603

[124] Y. Ma, M. Kim, Y. Cao, S. Vrudhula, and J. Seo. 2017. End-to-end scalable FPGA accelerator for deep residual networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1. https://doi.org/10.1109/ISCAS.2017.8050344

[125] Yifang Ma, Zhenyu Wang, Hong Yang, and Lin Yang. 2020. Artificial intelligence applications in the development of autonomous vehicles: A survey. *IEEE/CAA Journal of Automatica Sinica* 7, 2 (2020), 315–329.

[126] Abdulrahman Mahmoud, Neeraj Aggarwal, Alex Nobbe, Jose Rodrigo Sanchez Vicarte, Sarita V. Adve, Christopher W. Fletcher, Iuri Frosio, and Siva Kumar Sastry Hari. 2020. PyTorchFI: A Runtime Perturbation Tool for DNNs. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 25–31. https://doi.org/10.1109/DSN-W50199.2020.00014

[127] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W. Fletcher, Sarita V. Adve, Charbel Sakr, Naresh Shanbhag, Pavlo Molchanov, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. 2020. HarDNN: Feature Map Vulnerability Evaluation in CNNs. arXiv:2002.09786 [cs.LG] https://arxiv.org/abs/2002.09786

[128] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. 2018. Optimizing Software-Directed Instruction Replication for GPU Error Detection. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 842–854. https://doi.org/10.1109/SC.2018.00070

[129] Abdulrahman Mahmoud, Siva Kumar Sastry Hari, Christopher W. Fletcher, Sarita V. Adve, Charbel Sakr, Naresh Shanbhag, Pavlo Molchanov, Michael B. Sullivan, Timothy Tsai, and Stephen W. Keckler. 2021. Optimizing Selective Protection for CNN Resilience. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 127–138. https://doi.org/10.1109/ISSRE52982.2021.00025

[130] Abdulrahman Mahmoud, Thierry Tambe, Tarek Aloui, David Brooks, and Gu-Yeon Wei. 2022. GoldenEye: A Platform for Evaluating Emerging Numerical Data Formats in DNN Accelerators. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 206–214. https://doi.org/10.1109/DSN53405.2022.00031

[131] Michael W. Mahoney. 2011. Randomized Algorithms for Matrices and Data. *Foundations and Trends in Machine Learning* 3, 2 (11 2011), 123–224. https://doi.org/10.1561/2200000035 arXiv:https://www.emerald.com/ftmal/article-pdf/3/2/123/11134672/2200000035en.pdf

[132] Pierre Maillard, Yanran P. Chen, Jeff Barton, and Martin L. Voogel. 2021. Single Event Latchup (SEL) and Single Event Upset (SEU) Evaluation of Xilinx 7nm Versal™ ACAP programmable logic (PL). In *2021 IEEE Radiation Effects Data Workshop (REDW)*. 1–6. https://doi.org/10.1109/NSREC45046.2021.9679343

[133] Igor D.S. Miranda, Aman Arora, Zachary Susskind, Luis A.Q. Villon, Rafael F. Katopodis, Diego L.C. Dutra, Leandro S. De Araújo, Priscila M.V. Lima, Felipe M.G. França, Lizy K. John, and Mauricio Breternitz. 2022. LogicWiSARD: Memoryless Synthesis of

Weightless Neural Networks. In *2022 IEEE 33rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 19–26. https://doi.org/10.1109/ASAP54787.2022.00014

[134] Seyed Iman Mirzadeh, Mehrdad Farajtabar, Ang Li, N. Levine, A. Matsukawa, and H. Ghasemzadeh. 2020. Improved Knowledge Distillation via Teacher Assistant. In *AAAI*, Vol. 34. 5191. https://doi.org/10.1609/aaai.v34i04.5963 arXiv:1902.03393

[135] Sparsh Mittal. 2020. A survey on modeling and improving reliability of DNN algorithms and accelerators. *Journal of Systems Architecture* 104 (2020), 101689.

[136] Ricardo Pio Monti, Sina Tootoonian, and Robin Cao. 2018. Avoiding Degradation in Deep Feed-Forward Networks by Phasing Out Skip-Connections. *Artificial Neural Networks and Machine Learning (ICANN)* 11141 (2018). https://doi.org/10.1007/978-3-030-01424-7_44

[137] Bert Moons, Koen Goetschalckx, Nick Van Berckelaer, and Marian Verhelst. 2017. Minimum Energy Quantized Neural Networks. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 1921. https://doi.org/10.1109/ACSSC.2017.8335699 arXiv:1711.00215 [cs.NE]

[138] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* 29–40. https://doi.org/10.1109/MICRO.2003.1253181

[139] Markus Nagel and Rana Ali Amjad. 2020. Up or Down? Adaptive Rounding for Post-Training Quantization. In *Proceedings of the 37th International Conference on Machine Learning*. 10.

[140] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M. Bronstein, and Avi Mendelson. 2020. Loss Aware Post-training Quantization. *arXiv:1911.07190 [cs]* (March 2020). http://arxiv.org/abs/1911.07190 arXiv:1911.07190.

[141] Vinod Nair and Geoffrey E. Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proc. 27th International Conference on Machine Learning*. 807. https://icml.cc/Conferences/2010/papers/432.pdf

[142] Niranjhana Narayanan, Zitao Chen, Bo Fang, Guanpeng Li, Karthik Pattabiraman, and Nathan Debardeleben. 2022. Fault Injection for TensorFlow Applications. *IEEE Transactions on Dependable and Secure Computing* (2022).

[143] Mahdi Nazemi, Arash Fayyazi, Amirhossein Esmaili, Atharva Khare, Soheil Nazar

Shahsavani, and Massoud Pedram. 2021. NullaNet Tiny: Ultra-low-latency DNN inference through fixed-function combinational logic. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 266–267.

[144] Mahdi Nazemi, Ghasem Pasandi, and Massoud Pedram. 2018. Nullanet: Training deep neural networks for reduced-memory-access inference. *arXiv preprint arXiv:1807.08716* (2018).

[145] Mohamed A Neggaz, Ihsen Alouani, Smail Niar, and Fadi Kurdahi. 2019. Are cnns reliable enough for critical applications? an exploratory study. *IEEE Design & Test* 37, 2 (2019), 76–83.

[146] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. 2011. Reading Digits in Natural Images with Unsupervised Feature Learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning* (2011).

[147] Jennifer Ngadiuba, Vladimir Loncar, Maurizio Pierini, Sioni Summers, Giuseppe Di Guglielmo, Javier Duarte, Philip Harris, Dylan Rankin, Sergo Jindariani, Mia Liu, Kevin Pedro, Nhan Tran, Edward Kreinar, Sheila Sagear, Zhenbin Wu, and Duc Hoang. 2020. Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml. *Machine Learning: Science and Technology* 2, 1 (dec 2020), 015001. https://doi.org/10.1088/2632-2153/aba042

[148] OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Lo-

gan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. GPT-4 Technical Report. (2024). arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[149] Elbruz Ozen and Alex Orailoglu. 2020. Boosting bit-error resilience of DNN accelerators through median feature selection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3250–3262.

[150] Elbruz Ozen and Alex Orailoglu. 2022. Architecting Decentralization and Customizability in DNN Accelerators for Hardware Defect Adaptation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 3934–3945.

[151] Sinno Jialin Pan and Qiang Yang. 2009. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng.* 22, 10 (2009), 1345. https://doi.org/10.1109/TKDE.2009.191

[152] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. LibriSpeech: An ASR corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5206. https://doi.org/10.1109/ICASSP.2015.7178964

[153] Alessandro Pappalardo. 2023. *Xilinx/brevitas*. https://doi.org/10.5281/zenodo.3333552

[154] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Vol. 32. 8024. arXiv:1912.01703 http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[155] Philipp Petersen and Felix Voigtlaender. 2020. Equivalence of approximation by convolutional neural networks and fully-connected networks. *Proc. Amer. Math. Soc.* 148, 4 (2020), 1567–1581.

[156] Andy D. Pimentel. 2017. Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration. *IEEE Design & Test* 34, 1 (2017), 77–90. https://doi.org/10.1109/MDAT.2016.2626445

[157] Antonio Porsia, Giacomo Perlo, Annachiara Ruospo, and Ernesto Sanchez. 2025. On the Resilience of INT8 Quantized Neural Networks on Low-Power RISC-V Devices. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 119–122. https://doi.org/10.1109/DSN-W65791.2025.00049

[158] Trishna Rajkumar and Johnny Öberg. 2024. Guided Fault Injection Strategy for Rapid Critical Bit Detection in Radiation-Prone SRAM-FPGA. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. https://doi.org/10.23919/DATE58400.2024.10546801

[159] Brandon Reagen, Udit Gupta, Lillian Pentecost, Paul Whatmough, Sae Kyu Lee, Niamh Mulholland, David Brooks, and Gu-Yeon Wei. 2018. Ares: a framework for quantifying the resilience of deep neural networks. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 17, 6 pages. https://doi.org/10.1145/3195970.3195997

[160] Yossi Rubner, Carlo Tomasi, and Leonidas J. Guibas. 2000. The Earth Mover's Distance as a Metric for Image Retrieval. *Int. J. Comput. Vis.* 40 (2000), 99. https://doi.org/10.1023/A:1026543900054

[161] Annachiara Ruospo, Angelo Balaara, Alberto Bosio, and Ernesto Sanchez. 2020. A Pipelined Multi-Level Fault Injector for Deep Neural Networks. In *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 1–6. https://doi.org/10.1109/DFT50435.2020.9250866

[162] A. Ruospo, G. Gavarini, C. de Sio, J. Guerrero, L. Sterpone, M. Sonza Reorda, E. Sanchez, R. Mariani, J. Aribido, and J. Athavale. 2023. Assessing Convolutional Neural Networks Reliability through Statistical Fault Injections. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. https://doi.org/10.23919/DATE56975.2023.10136998

[163] Annachiara Ruospo, Matteo Sonza Reorda, Riccardo Mariani, and Ernesto Sanchez. 2025. An Effective Iterative Statistical Fault Injection Methodology for Deep Neural Networks. *IEEE Trans. Comput.* 74, 7 (2025), 2431–2444. https://doi.org/10.1109/TC.2025.3566863

[164] Vladimir Rybalkin, Alessandro Pappalardo, Muhammad Mohsin Ghaffar, Giulio Gambardella, Norbert Wehn, and Michaela Blott. 2018. FINN-L: Library Extensions and Design Trade-Off Analysis for Variable Precision LSTM Networks on FPGAs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 89–897. https://doi.org/10.1109/FPL.2018.00024

[165] Omer Sagi and Lior Rokach. 2018. Ensemble learning: A survey. *Wiley interdisciplinary reviews: data mining and knowledge discovery* 8, 4 (2018), e1249.

[166] Behzad Salami, Osman S. Unsal, and Adrian Cristal Kestelman. 2018. On the Resilience of RTL NN Accelerators: Fault Characterization and Mitigation. In *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 322–329. https://doi.org/10.1109/CAHPC.2018.8645906

[167] Bharat Bhusan Sau and Vineeth N. Balasubramanian. 2016. Deep Model Compression: Distilling Knowledge from Noisy Teachers. (2016). arXiv:1610.09650

[168] Benjamin Carrion Schafer and Zi Wang. 2020. High-Level Synthesis Design Space Exploration: Past, Present, and Future. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2628–2639. https://doi.org/10.1109/TCAD.2019.2943570

[169] Anna Schmedding, Lishan Yang, Adwait Jog, and Evgenia Smirni. 2024. Aspis: Lightweight Neural Network Protection Against Soft Errors. In *2024 IEEE 35th International Symposium on Software Reliability Engineering (ISSRE)*. 248–259. https://doi.org/10.1109/ISSRE62328.2024.00036

[170] Christoph Schorn, Andre Guntoro, and Gerd Ascheid. 2018. Accurate neuron resilience prediction for a flexible reliability management in neural network accelerators. In *2018*

Design, Automation & Test in Europe Conference & Exhibition (DATE). 979–984. https://doi.org/10.23919/DATE.2018.8342151

[171] Nida Shahid, Tim Rappon, and Whitney Berta. 2019. Applications of artificial neural networks in health care organizational decision-making: A scoping review. *PLOS ONE* 14, 2 (02 2019), 1–22. https://doi.org/10.1371/journal.pone.0212356

[172] Jordi Silvestre-Ryan and Ian Holmes. 2021. Pair consensus decoding improves accuracy of neural network basecallers for nanopore sequencing. *Genome Biol.* 22 (2021), 38. https://doi.org/10.1186/s13059-020-02255-1

[173] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556

[174] David Stutz, Nandhini Chandramoorthy, Matthias Hein, and Bernt Schiele. 2021. Bit error robustness for energy-efficient dnn accelerators. *Proceedings of Machine Learning and Systems* 3 (2021), 569–598.

[175] Chang Sun, Zhiqiang Que, Vladimir Loncar, Wayne Luk, and Maria Spiropulu. 2026. da4ml: Distributed Arithmetic for Real-time Neural Networks on FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 19, 1, Article 13 (March 2026), 27 pages. https://doi.org/10.1145/3777387

[176] Zachary Susskind, Aman Arora, Igor D. S. Miranda, Alan T. L. Bacellar, Luis A. Q. Villon, Rafael F. Katopodis, Leandro S. de Araújo, Diego L. C. Dutra, Priscila M. V. Lima, Felipe M. G. França, Mauricio Breternitz Jr., and Lizy K. John. 2023. ULEEN: A Novel Architecture for Ultra-low-energy Edge Neural Networks. *ACM Trans. Archit. Code Optim.* 20, 4, Article 61 (Dec. 2023), 24 pages. https://doi.org/10.1145/3629522

[177] Zachary Susskind, Aman Arora, Igor D. S. Miranda, Luis A. Q. Villon, Rafael F. Katopodis, Leandro S. de Araújo, Diego L. C. Dutra, Priscila M. V. Lima, Felipe M. G. França, Mauricio Breternitz, and Lizy K. John. 2023. Weightless Neural Networks for Efficient Edge Inference. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques* (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 279–290. https://doi.org/10.1145/3559009.3569680

[178] Zachary Susskind, Alan TL Bacellar, Aman Arora, Luis AQ Villon, Renan Mendanha, Leandro Santiago de Araújo, Diego Leonel Cadette Dutra, Priscila MV Lima, Felipe MG França, Igor DS Miranda, Mauricio Breternitz Jr, and Lizy K John. 2022. Pruning weightless neural networks. *ESANN 2022 proceedings* (2022).

[179] Emil Talpes, Debjit Das Sarma, Ganesh Venkataramanan, Peter Bannon, Bill McGee,

Benjamin Floering, Ankit Jalote, Christopher Hsiong, Sahil Arora, Atchyuth Gorti, and Gagandeep S. Sachdev. 2020. Compute Solution for Tesla's Full Self-Driving Computer. *IEEE Micro* 40, 2 (2020), 25–35. https://doi.org/10.1109/MM.2020.2975764

[180] Lucas Antunes Tambara, Jorge Tonfat, André Santos, Fernanda Lima Kastensmidt, Nilberto H. Medina, Nemitala Added, Vitor A. P. Aguiar, Fernando Aguirre, and Marcilei A. G. Silveira. 2017. Analyzing Reliability and Performance Trade-Offs of HLS-Based Designs in SRAM-Based FPGAs Under Soft Errors. *IEEE Transactions on Nuclear Science* 64, 2 (2017), 874–881. https://doi.org/10.1109/TNS.2017.2648978

[181] Antti Tarvainen and Harri Valpola. 2017. Mean Teachers Are Better Role Models: Weight-Averaged Consistency Targets Improve Semi-Supervised Deep Learning Results. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. 1195. arXiv:1703.01780 https://proceedings.neurips.cc/paper/2017/file/68053af2923e00204c3ca7c6a3150cf7-Paper.pdf

[182] {The CMS Collaboration}. 2008. The CMS experiment at the CERN LHC. *Journal of Instrumentation* 3, August 2008 (14 Aug. 2008), 1–334. https://doi.org/10.1088/1748-0221/3/08/S08004

[183] Ryan Theisen, Hyunsuk Kim, Yaoqing Yang, Liam Hodgkinson, and Michael W Mahoney. 2024. When are ensembles really effective? *Advances in Neural Information Processing Systems* 36 (2024).

[184] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[185] Cesar Torres-Huitzil and Bernard Girau. 2017. Fault and Error Tolerance in Neural Networks: A Review. *IEEE Access* 5 (2017), 17322–17341. https://doi.org/10.1109/ACCESS.2017.2742698

[186] Marcello Traiola, Angeliki Kritikakou, and Olivier Sentieys. 2023. harDNNing: a machine-learning-based framework for fault tolerance assessment and protection of DNNs. In *ETS 2023-IEEE European Test Symposium*.

[187] Marcello Traiola, Angeliki Kritikakou, and Olivier Sentieys. 2023. A machine-learning-guided framework for fault-tolerant DNNs. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–2. https://doi.org/10.23919/DATE56975.2023.10137033

[188] Ilya Tuzov, David de Andrés, Juan-Carlos Ruiz, and Carles Hernández. 2023. BAFFI: a bit-accurate fault injector for improved dependability assessment of FPGA prototypes.

In *2023 Design, Automation & Test in Europe Conference & Exhibition*. 1–6. https://doi.org/10.23919/DATE56975.2023.10137300

[189] Shashanka Ubaru, Jie Chen, and Yousef Saad. 2017. Fast estimation of tr(f(A)) via stochastic Lanczos quadrature. *SIAM J. Matrix Anal. Appl.* 38, 4 (2017), 1075–1099.

[190] Yaman Umuroglu, Yash Akhauri, Nicholas James Fraser, and Michaela Blott. 2020. LogicNets: Co-designed neural networks and circuits for extreme-throughput applications. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 291–297.

[191] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*. 65–74.

[192] Andreas Veit, Michael J Wilber, and Serge Belongie. 2016. Residual networks behave like ensembles of relatively shallow networks. *Advances in Neural Information Processing Systems* 29 (2016). arXiv:1605.06431 https://proceedings.neurips.cc/paper/2016/file/37bc2f75bf1bcfe8450a1a41c200364c-Paper.pdf

[193] Shashikiran Venkatesha and Ranjani Parthasarathi. 2024. Survey on Redundancy Based-Fault tolerance methods for Processors and Hardware accelerators - Trends in Quantum Computing, Heterogeneous Systems and Reliability. *ACM Comput. Surv.* 56, 11, Article 275 (June 2024), 76 pages. https://doi.org/10.1145/3663672

[194] Zishen Wan, Aqeel Anwar, Abdulrahman Mahmoud, Tianyu Jia, Yu-Shun Hsiao, Vijay Janapa Reddi, and Arijit Raychowdhury. 2022. FRL-FI: Transient Fault Analysis for Federated Reinforcement Learning-Based Navigation Systems. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 430–435. https://doi.org/10.23919/DATE54114.2022.9774562

[195] Erwei Wang, James J Davis, Peter YK Cheung, and George A Constantinides. 2019. LUTNet: Rethinking inference in FPGA soft logic. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 26–34.

[196] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-aware automated quantization with mixed precision. In *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 8612. https://doi.org/10.1109/CVPR.2019.00881 arXiv:1811.08886

[197] Yumou Wei, Ryan F Forelli, Chris Hansen, Jeffrey P Levesque, Nhan Tran, Joshua C Agar, Giuseppe Di Guglielmo, Michael E Mauel, and Gerald A Navratil. 2023. *Low latency*

optical-based mode tracking with machine learning deployed on FPGAs on a tokamak. Technical Report. Fermi National Accelerator Laboratory (FNAL), Batavia, IL (United States).

[198] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *J. Big Data* 3, 1 (2016), 1. https://doi.org/10.1186/s40537-016-0043-6

[199] Olivia Weng, Marta Andronic, Danial Zuberi, Jiaqing Chen, Caleb Geniesse, George A. Constantinides, Nhan Tran, Nicholas J. Fraser, Javier Mauricio Duarte, and Ryan Kastner. 2025. Greater than the Sum of its LUTs: Scaling Up LUT-based Neural Networks with AmigoLUT. In *Proceedings of the 2025 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '25)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/3706628.3708874

[200] Olivia Weng, Andres Meza, Quinlan Bock, Benjamin Hawks, Javier Campos, Nhan Tran, Javier Mauricio Duarte, and Ryan Kastner. 2024. FKeras: A Sensitivity Analysis Tool for Edge Neural Networks. *ACM J. Auton. Transport. Syst.* 1, 3, Article 15 (July 2024), 27 pages. https://doi.org/10.1145/3665334

[201] Olivia Weng, Andres Meza, Nhan Tran, and Ryan Kastner. 2026. PrioriFI: More Informed Fault Injection for Edge Neural Networks. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Pittsburg, PA, USA) *(ASPLOS '26)*. Association for Computing Machinery, New York, NY, USA, 15 pages. https://doi.org/10.1145/3779212.3790204

[202] Olivia Weng, Alexander Redding, Nhan Tran, Javier Mauricio Duarte, and Ryan Kastner. 2024. Architectural implications of neural network inference for high data-rate, low-latency scientific applications. *arXiv preprint arXiv:2403.08980* (2024).

[203] Tim Whitaker and Darrell Whitley. 2022. Prune and tune ensembles: low-cost ensemble learning with sparse independent subnetworks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 8638–8646.

[204] Yaoqing Yang, Liam Hodgkinson, Ryan Theisen, Joe Zou, Joseph E Gonzalez, Kannan Ramchandran, and Michael W Mahoney. 2021. Taxonomizing local versus global structure in neural network loss landscapes. *Advances in Neural Information Processing Systems* 34 (2021), 18722–18733.

[205] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, and Kurt Keutzer. 2019. Synetgy: Algorithm-hardware Co-design for ConvNet Accelerators on Embedded FPGAs. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing

Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/3289602.3293902

[206] Yongquan Yang, Haijun Lv, and Ning Chen. 2023. A survey on ensemble learning under the era of deep learning. *Artificial Intelligence Review* 56, 6 (2023), 5545–5589.

[207] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, and Kurt Keutzer. 2021. HAWQ-V3: Dyadic Neural Network Quantization. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 11875–11886. https://proceedings.mlr.press/v139/yao21a.html

[208] Zhewei Yao, Amir Gholami, Kurt Keutzer, and Michael W. Mahoney. 2020. PyHessian: Neural Networks Through the Lens of the Hessian. In *2020 IEEE International Conference on Big Data (Big Data)*. 581–590. https://doi.org/10.1109/BigData50022.2020.9378171

[209] Jieun Yoo, Jennet Dickinson, Morris Swartz, Giuseppe Di Guglielmo, Alice Bean, Douglas Berry, Manuel Blanco Valentin, Karri DiPetrillo, Farah Fahim, Lindsey Gray, James Hirschauer, Shruti R Kulkarni, Ron Lipton, Petar Maksimovic, Corrinne Mills, Mark S Neubauer, Benjamin Parpillon, Gauri Pradhan, Chinar Syal, Nhan Tran, Dahai Wen, and Aaron Young. 2024. Smart pixel sensors: towards on-sensor filtering of pixel clusters with deep learning. *Machine Learning: Science and Technology* 5, 3 (aug 2024), 035047. https://doi.org/10.1088/2632-2153/ad6a00

[210] Yi Yuan and Derek Chiou. 2025. Accelerating Fault Injection for Validating Processor RTL Implementations. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design* (Newark Liberty International Airport Marriott, New York, NY, USA) *(ICCAD '24)*. Association for Computing Machinery, New York, NY, USA, Article 97, 9 pages. https://doi.org/10.1145/3676536.3676759

[211] Sergey Zagoruyko and Nikos Komodakis. 2017. DiracNets: Training Very Deep Neural Networks Without Skip-Connections. (2017). arXiv:1706.00388

[212] Sergey Zagoruyko and Nikos Komodakis. 2018. DiracNets: Training Very Deep Neural Networks Without Skip-Connections. (2018). arXiv:1706.00388

[213] Ussama Zahid, Giulio Gambardella, Nicholas J Fraser, Michaela Blott, and Kees Vissers. 2020. FAT: Training neural networks for reliable inference under hardware faults. In *2020 IEEE International Test Conference (ITC)*. IEEE, 1–10.

[214] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design* (Austin, Texas) *(ICCAD '16)*. Association for Computing Machinery, New York, NY, USA, Article

12, 8 pages. https://doi.org/10.1145/2966986.2967011

[215] Yangchao Zhang, Hiroaki Itsuji, Takumi Uezono, Tadanobu Toba, and Masanori Hashimoto. 2022. Estimating vulnerability of all model parameters in dnn with a small number of fault injections. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 60–63.

[216] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2018. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv:1606.06160 [cs]* (Feb. 2018). http://arxiv.org/abs/1606.06160 arXiv: 1606.06160.

[217] Zhi-Hua Zhou. 2012. *Ensemble methods: foundations and algorithms*. CRC press.

[218] Shilin Zhu, Xin Dong, and Hao Su. 2019. Binary ensemble neural network: More bits per network or more networks per bit?. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 4923–4932.

[219] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43. https://doi.org/10.1109/JPROC.2020.3004555