A VISION FOR EDGE AI: ROBUST BINARIZED NEURAL NETWORKS ON EMERGING RESOURCE-CONSTRAINED HARDWARE

Dissertation

zur Erlangung des Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN

der Technischen Universität Dortmund
an der Fakultät Informatik

von

Mikail Yayla

Dortmund
2024

Tag der mündlichen Prüfung:	28.02.2024
Dekan:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter:	Prof. Dr. Jian-Jia Chen
	Prof. Dr.-Ing. Jürgen Teich

ABSTRACT

The recent success of neural networks (NNs) has brought benefits to numerous fields, while progressively pervading all aspects of our life. However, to achieve high accuracy, NNs use a massive number of parameters. Therefore, two main bottlenecks can be identified for realizing systems that execute NNs: The memory subsystem and the processing units.

On the NN model side, Binarized NNs (BNNs), offer high efficiency in both memory and computations at small accuracy cost compared to conventional integer or floating-point NNs, while still retaining the universal approximation property. The binary nature of the weights and inputs brings three major benefits: (1) Reduction of data movements between memories and computing units, (2) replacing the multiplications with bitwise XNOR and accumulations with popcount operations, and (3) high error tolerance, i.e. robustness to perturbations in weights, inputs, and intermediate data.

On the hardware side, the principles of approximate computing are applied on the memory and the processing units, where efficiency is traded for quality of the operation's result. In approximate memory, the supply voltage and access latency parameters of the memory are configured, to achieve lower energy consumption and faster access at the cost of reliability in form of bit errors. When NNs are executed using approximate computing units, configurable approximate circuits can be finely tuned to trade resource usage for computing errors, especially when analog computing is employed, where signals are represented approximately. All in all, BNNs synergize excellently with approximate memory and computing due to their high robustness.

To acquire BNN models with high accuracy, they have to be trained with large resource cost. The training of BNNs is by orders of magnitude more resource-intensive than inference. An efficient approach is to train BNNs on dedicated low-power accelerators in an on-chip setting, such as FPGAs or ASICs. In addition to energy-efficiency of on-chip training, privacy issues and data transfer overheads are eliminated, since the data does not need to be transferred to the cloud for training. For these reasons, resource-efficient models, such as BNNs, should not only be executed but also be trained on the edge.

Vision of this Dissertation: This dissertation proposes a vision for highly resource-efficient future intelligent systems that are comprised of robust BNNs operating with approximate memory and approximate computing units, while being able to be trained on the edge. The studies conducted within the scope of this dissertation are summarized in the following.

BNN Robustness Optimization: The classical approach for increasing the robustness of NNs is injecting bit flips from the error model during training. However, the drawbacks are that it degrades accuracy and adds high overhead. Achieving robustness in NNs without bit flip injection would be of great benefit for the robustness optimization of NNs. BNNs are composed of simple structures that enable exploration of robustness metrics based on margins. We present formal proofs that quantify the maximum number of bit flips that can be tolerated in neurons, which allows us to propose the modified hinge loss (MHL). The MHL trains BNNs for robustness without bit flip injections and enables them to tolerate higher bit error rates than with bit flip training, thus lowering the requirements on approximate memories and computing units.

FeFET-based Approximate Memory for BNNs: First, we explore Ferroelectric FET (FeFET), a promising emerging memory, as on-chip memory for BNNs and show that changes in memory temperature during runtime causes unacceptable accuracy drop. We propose two countermeasures for temperature-tolerance across the entire range of operating temperature: (1) Training BNNs for bit error tolerance by injecting bit flips and (2) using a post-training bit error rate assignment algorithm. Secondly, we investigate the use of FeFET-based XNOR gates for logic-in-memory (LiM). We show that the latency of the XNOR gates, which constitutes a major bottleneck, can be significantly reduced by exploiting the BNN robustness.

HW/SW Codesign for Efficient BNN Acceleration: First, we propose the Local Thresholding Approximation (LTA), which significantly increases the efficiency of the interface circuit in analog-based BNN crossbar accelerators. The LTA reduces the area, energy, and latency of the BNN HW significantly compared to the state of the art. Secondly, we optimize another analog computing scheme for BNNs, i.e. Integrate-and-Fire (IF) Spiking Neural Networks (SNNs). To achieve high inference accuracy in IF-SNNs, the analog HW needs to represent current-based MAC levels as spike times, for which a large membrane capacitor is required. To alleviate this, we propose a HW/SW Codesign method, called CapMin, for capacitor size minimization in analog computing IF-SNNs. To increase the computation's tolerance to process variation, we propose CapMin-V, which trades capacitor size for protection.

Resource-Efficient Training of BNNs: BNN training suffers from high memory usage, making the design of on-chip BNN training accelerators a challenge. The Binary optimizer (Bop) is one of the most memory-efficient training procedures for BNNs but still uses momentum values encoded as Floating Point (FP), leading to high memory usage. We propose methods for memory-efficient training of BNNs on the edge. To this end, we theoretically investigate the impact of arbitrary encodings on training information loss. Based on this, we develop an algorithm to find memory-efficient FP encodings, reducing memory usage of BNN training significantly compared to using 32-bit FP encoding.

PUBLICATIONS

The following conference and journal publications are included in parts or in extended versions in this dissertation:

[Bus+21] Sebastian Buschjäger, Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Christian Hakert, Katharina Morik, Rodion Novkin, Lukas Pfahler, and Mikail Yayla. "Margin-Maximization in Binarized Neural Networks for Optimizing Bit Error Tolerance." In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2021.

[Yay+22a] Mikail Yayla, Sebastian Buschjäger, Aniket Gupta, Jian-Jia Chen, Jörg Henkel, Katharina Morik, Kuan-Hsun Chen, and Hussam Amrouch. "FeFET-Based Binarized Neural Networks Under Temperature-Dependent Bit Errors." In: *IEEE Transactions on Computers* (2022).

[Yay+22b] Mikail Yayla, Simon Thomann, Sebastian Buschjäger, Katharina Morik, Jian-Jia Chen, and Hussam Amrouch. "Reliable Binarized Neural Networks on Unreliable Beyond Von-Neumann Architecture." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2022).

[YC22] Mikail Yayla and Jian-Jia Chen. "Memory-Efficient Training of Binarized Neural Networks on the Edge." In: *Design Automation Conference (DAC)*. 2022.

[Yay+23c] Mikail Yayla, Fabio Frustaci, Fanny Spagnolo, Jian-Jia Chen, and Hussam Amrouch. "Unlocking Efficiency in BNNs: Global by Local Thresholding for Analog-based HW Accelerators." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2023).

[Yay+23b] Mikail Yayla, Simon Thomann, Ming-Liang Wei, Chia-Lin Yang, Jian-Jia Chen, and Hussam Amrouch. "HW/SW Codesign for Robust and Efficient Binarized SNNs by Capacitor Minimization." In: *arXiv:2309.02111* (2023).

In my research, I also contributed to the following conference and journal publications, which are outside of the scope of this dissertation and not included:

[Hak+19] Christian Hakert, Mikail Yayla, Kuan-Hsun Chen, Georg von der Brüggen, Jian-Jia Chen, Sebastian Buschjäger, Katharina Morik, Paul R. Genssler, Lars Bauer, Hussam Amrouch, et al. "Stack Usage Analysis for Efficient Wear Leveling in Non-Volatile Main Memory Systems." In: *Workshop on Machine Learning for CAD (MLCAD)*. 2019.

[Wei+21a] Ming-Liang Wei, Mikail Yayla, Shu-Yin Ho, Jian-Jia Chen, Chia-Lin Yang, and Hussam Amrouch. "Binarized SNNs: Efficient and Error-Resilient Spiking Neural Networks through Binarization." In: *International Conference On Computer Aided Design (ICCAD)*. 2021.

[Yay+22c] Mikail Yayla, Zahra Valipour Dehnoo, Mojtaba Masoudinejad, and Jian-Jia Chen. "TREAM: A Tool for Evaluating Error Resilience of Tree-Based Models Using Approximate Memory." In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2022.

[Dav+23] Abhilasha Dave, Fabio Frustaci, Fanny Spagnolo, Mikail Yayla, Jian-Jia Chen, and Hussam Amrouch. "HW/SW Codesign for Approximation-Aware Binary Neural Networks." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2023).

[Sab+23] Muhammad Sabih, Mikail Yayla, Frank Hannig, Jürgen Teich, and Jian-Jia Chen. "Robust and Tiny Binary Neural Networks using Gradient-based Explainability Methods." In: *Workshop on Machine Learning and Systems (EuroMLSys)*. 2023.

[Moh+23] Vahidreza Mohaghaddas, Hammam Kattan, Tim Buecher, Mikail Yayla, Jian-Jia Chen, and Hussam Amrouch. "Temperature-Aware Memory Mapping and Active Cooling of Neural Processing Units." In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2023.

[Yay+23a] Mikail Yayla, Cecilia Latotzke, Robert Huber, Somar Iskif, Tobias Gemmeke, and Jian-Jia Chen. "DAEBI: A Tool for Data Flow and Architecture Explorations of Binary Neural Network Accelerators." In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2023.

[Wei+23] Ming-Liang Wei, Mikail Yayla, Shu-Yin Ho, Jian-Jia Chen, Hussam Amrouch, and Chia-Lin Yang. "Impact of Non-volatile Memory Cells on Spiking Neural Network Annealing Machine with In-situ Synapse Processing." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2023).

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to my advisor Prof. Dr. Jian-Jia Chen for his invaluable support during my years, the countless opportunities he provided, and the freedom and trust to explore my own ideas. I will always remember my time as a Ph.D researcher and will draw on the lessons I learned under his supervision for my entire life.

I would also like to thank Prof. Dr.-Ing. Hussam Amrouch. I very much appreciate all the help and guidance he has given me and I wish him the best for his new path. I also want to thank Ass.-Prof. Dr.-Ing. Kuan-Hsun Chen for motivating me to start a career in research in the early days.

A thank you also to Prof. Dr.-Ing Jürgen Teich for taking the time to review this dissertation, and Prof. Dr. Jens Teubner and Prof. Dr. Stefan Harmeling for agreeing to serve on the doctoral committee.

To the members our DAES Group and our close collaborators who are now in TU Munich, now Lamarr members, at Karlsruhe Institute of Technology, at the National Taiwan University, at the University of Calabria, and at RWTH Aachen: I very much enjoyed working with you, thank you for all the productive and fun times.

Last but not least, I want to thank my family. Especially my wife Wei, who was was always willing to provide her support. And my son Elias for brightening up the days.

CONTENTS

1 INTRODUCTION

Neural networks (NNs) have profoundly changed our lives in various aspects and continue to do so with countless emerging breakthroughs. They surpass traditional algorithms and human performance in many challenges. Due to their exceptional capability, they have been applied successfully in various fields, such as object detection and tracking, image and speech recognition, natural language processing, in control tasks exemplified by autonomous driving or industrial automation, recommendation systems, in medicine, and many more.

Nevertheless, NNs are highly resource demanding, which makes it a challenge to deploy them efficiently. A goal worth pursuing is to execute NNs on the edge, i.e. instead of using powerful servers, the computations are performed close to the sensors or other forms of data sources, with reduced energy, area, and latency. This is particularly useful in scenarios where private data needs to be processed efficiently, such as in autonomous driving, object detection or tracking, speech recognition, and authentication. To achieve this goal, a promising type of NNs are Binarized Neural Networks (BNNs), in which the weights and inputs are binarized, leading to outstanding efficiency in area and energy, enabling inference with low latency, while also offering high robustness against errors.

***Vision of this Dissertation**: This dissertation proposes a vision for efficient future intelligent systems that are comprised of robust BNNs operating with approximate memory and approximate computing units, while being able to be trained on the edge.*

The motivation for the vision is given in Sec. 1.1, the challenges and goals are in Sec. 1.2, and the contributions of this dissertation are in Sec. 1.3.

1.1 MOTIVATION

RESOURCE DEMAND OF NNS

To achieve high accuracy, modern and high-performing NN models use a massive number of parameters and compute an immensely large amount of MAC operations during their operation. Therefore, two main bottlenecks can be identified for realizing system that execute NNs: The memory subsystem and the processing units. NNs need to move data from the memory to the computation units, which necessitates data movement across the entire memory hierarchy, in which traditionally large, slow memory is used on the lower end (e.g. DRAM and SSD), and small, fast memory on the higher end (e.g. caches and registers). The data movement uses a large portion of the total system energy, for which methods regarding estimations are proposed

in [Yan+17]. The study reports that a major portion of the system energy is consumed for data movements for weight, layer input, and layer output data, which makes efficient NN execution a challenge. To alleviate this, several approaches to reduce the data movement are presented in [Sze+17], exploiting the reuse of data to avoid the data movements, emphasizing again the importance of the computation units. Once the data has been moved close to or into the computation units, floating-point or high-bit precision MAC operations need to be computed. Although MAC operations can be processed efficiently in modern hardware in CPUs and GPUs, the required amount of MAC operations in NNs still makes the efficient processing challenging. These problems regarding data movement and processing, are expected to be exacerbated in the future, as NNs are increasing in size to solve increasingly complex problems.

NNS ON THE EDGE

Yet, many use cases require efficient and intelligent decisions, which necessitate the inference to be performed on edge devices to reduce resources and to increase privacy. However, edge devices provide only limited resources in energy, area, and execution latency, posing a profound challenge for the design of efficient yet capable intelligent systems on such devices. Several approaches have been proposed to enable the efficient processing of NNs on the edge, e.g. regarding data flow in [Sze+17], while pruning [Lia+21], compression [MGD20], and quantization [Gho+21; Kul+22] also have shown promising results. Specifically, when using the extreme form of quantization, i.e. binarization to $\{0, 1\}$, the operations of NNs also become extremely efficient.

BINARIZED NEURAL NETWORKS

Binarized NNs (BNNs), introduced in 2016 [Hub+16; Ras+16; KS16], offer high efficiency in both memory and computations at small accuracy cost compared to conventional integer or floating-point based NNs. On top of that, BNNs are also universal approximators, meaning they can (under some theoretical restrictions) learn any task [Wan+18; DLS18; Spa+19]. Since their inception, BNNs have been applied in various domains. Examples for applications are real-time image classification [C+22; PE23], high speed object recognition and human activity recognition [Zha+23b; Luo+23], autonomous vehicle control [WHA18], text classification [Shr+20], in wildfire detection [CP23], and in agricultural settings [Hua21]. Furthermore, frameworks for deploying BNNs on various types of hardware have been published, namely the FINN a framework to deploy BNNs on FPGAs [Umu+17], BMXNet focuses on general purpose CPUs and GPUs [Bet+18], whereas [Ban+21] (Larq framework) and [Zha+19a] (DaBNN framework) focus on mobile platforms with ARM processors.

The binarization in BNNs leads to major benefits to the system compared to multi-bit NNs. Due to the binarization, the amount of data that needs to be stored compared

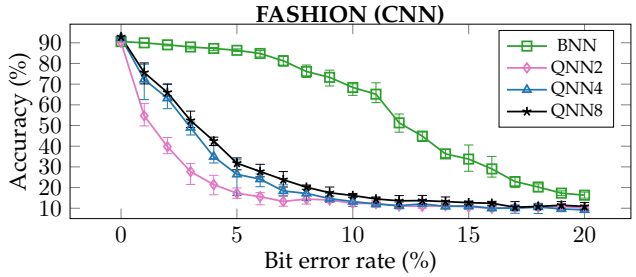


Figure 1.1: Comparison of error tolerance between BNNs and QNNs. No method for improvement is applied for BNNs or QNNs, that is why QNNs with 2 bit may be less robust than with more precision (the study in [Stu+23] proposes methods which makes lower-precision QNNs more error tolerant than higher-precision QNNs). See Ch. 3 for the experiment settings.

to floating-point NNs is approximately 32× less. This also leads to reduced data movements between the memory subsystem and the computing units. In addition, binary multiplications can be performed by bitwise XNOR between the weights and the inputs, while accumulations can be performed with popcount operations. This leads to 7× improvement in latency [Hub+16] compared to floating-point NNs.

ERROR TOLERANCE OF BNNS

BNNs have exceptionally high intrinsic error tolerance [Hir+19b], i.e. they are naturally error tolerant when classical training methods are employed. In Fig. 1.1, we show the drastic difference regarding error tolerance between binarized and multi-bit quantized NNs (QNN), where no countermeasures (e.g. robustness training) are applied.

In fact, the classical approach to achieve robustness in NNs and BNNs is the robustness training, i.e. the approach to inject bit flips during training. However, this has significant disadvantages. First, injecting bit flips during training can significantly degrade accuracy. The higher the bit error rate (BER) during training, the more significant the accuracy degradation. Another disadvantage is the additional overhead. For every bit of the error-prone data, a decision has to be made whether to inject a bit flip, which adds numerous additional steps in the NN training.

Achieving bit error tolerance in NNs without bit flip injection and conquering the above disadvantages, would be a breakthrough for the area of error tolerance. However, the principles of bit error tolerance in NNs need to be well understood first. To the best of our knowledge, before this dissertation work, the theoretical foundations of NN bit error tolerance have not been reported. Since BNNs have simpler structures, it allows for easier exploration of bit tolerance properties. When the properties of error tolerance are known and understood, they can better be exploited for the design of efficient systems that run BNNs by using approximations in their stored data and

computations. In summary, *BNNs have outstanding error tolerance, which needs to be understood better for more effective optimization*.

EXPLOITING THE ERROR TOLERANCE OF BNNS: MEMORY AND COMPUTATION

To reduce the resource cost in traditional NNs, the principles of approximate computing are applied on the memory and the processing units, where efficiency is traded for the quality of the operations' result [Mit16], exploiting the error tolerance of NNs [TG17].

In approximate memory, the supply voltage and access latency parameters of the memory are configured, to achieve lower energy consumption and faster access at the cost of reliability in form of bit errors [Kop+19]. This is especially favorable in combination with emerging non-volatile memory (NVM) [Bou+17], which use almost no energy when remained unused and use little energy when used. Furthermore, when operations are performed near memory or in-memory, such as in logic-in-memory, less data needs to be loaded into processing units.

When NNs are executed using approximate computing units, configurable approximate circuits, i.e. MAC units, can be finely tuned to trade resource usage for computing errors, or computations can be skipped for efficiency [Arm+22]. Especially analog-computing based hardware (which has been shown to be highly efficient [Chi+16; Sha+16a]) has an excellent synergy with BNNs, since (1) the signal levels and computations in the analog domain are inherently approximate, (2) the number of signal levels in BNNs that need to be represented in the analog domain is lower compared to higher-precision cases, and (3) high efficiency is achieved due to computations based on Kirchhoff's and Ohm's laws instead of employing costly digital logic.

The capability of BNNs to be optimized for error tolerance and the opportunity to exploit during system design has not received much attention in the literature before the work of this dissertation. BNNs with the high degree of error tolerance combined with approximate memory or computing approaches would enable the design of highly efficient intelligent systems, leading to breakthroughs in efficient edge intelligence. In summary, *BNNs synergize excellently with approximate memory and computing due to their high robustness*.

EFFICIENT TRAINING OF BNNS

Acquiring BNN models that have high accuracy is also highly challenging. One method is to to use established NN architectures and simulate the binarization of their operations during the training [Hub+16], while another method is to employ neural architecture search (NAS) [Gou+21] where search algorithms are used to find suitable binarized NN structures. In any case, similar to traditional NNs, the training of BNNs is by orders of magnitude more resource-intensive than inference,

as expensive and energy-hungry computation resources, such as modern GPUs with large memories are required, raising serious concerns regarding sustainability due to the carbon footprint of the training procedure [SGM20]. An efficient approach is to train BNNs on dedicated low-power accelerators in an on-chip setting, such as FPGAs or ASICs. In addition to energy-efficiency of on-chip training, privacy issues and data transfer overheads are eliminated, since the data does not need to be transferred to the cloud for training. For these reasons, *BNNs should not only be executed but also be trained on the edge.*

1.2 CONTEXT, CHALLENGES, AND GOALS

To realize the vision of this dissertation, we first identify its challenges. Then, we define goals that aim to overcome the challenges. The vision consists of three parts, which are: (1) Obtaining error tolerant BNNs, (2) BNNs using approximate memory and computation units, (3) training BNNs on the edge. Each part has its unique challenges and goals, which we discuss in the following.

Obtaining Error Tolerant BNNs without Reliance on Error Models

As explained in Sec. 1.1, before the work of this dissertation, the only method to optimize NNs for error tolerance was applying the error model during the training. However, the two disadvantages of this method are the accuracy degradation and the additional overhead of the error model application.

To overcome the two disadvantages, NN error tolerance should be achievable without applying the error model during training. Specifically, NNs that have general error tolerance should be obtainable effectively, without accuracy degradation and without additional overheads. However, this is highly challenging, because of the "black box" nature of the error tolerance in NNs; it is a property that is not well understood. Therefore, formal and understandable techniques that uncover the underlying principles of error tolerance on NNs need to be examined. This way, methods can be found for the optimization of error tolerance without the aforementioned disadvantages.

In this dissertation, we first plan to achieve error tolerance in NNs without applying the error model during training. In other words, *our first goal is to obtain NNs that have general error tolerance, without depending on any error model*.

In Ch. 4, we study the underlying principles of error tolerance in BNNs, which have a simpler structure, enabling the effective exploration of formal error tolerance metrics. In particular, we propose neuron and output layer metrics which describe the error tolerance of BNNs in a closed form. Then we optimize the BNNs based on these metrics and achieve higher error tolerance compared to classical methods. With BNNs that have increased error tolerance, the requirements of approximate memory and computing can be lowered, i.e. the result quality can be further reduced for achieving

higher system efficiency. Next, we discuss how the achieved error tolerance can be exploited in systems consisting of approximate memory and computing units.

Error Tolerant BNNs Using Approximate Memory and Computation Units

Despite the synergy, designing systems that use approximate memory and computing units in combination with error tolerant BNNs have not received much attention in the literature before the work of this dissertation. The reasons are that hardware-level experts usually do not have in-depth knowledge of NNs or BNNs, and the knowledge regarding the outstanding error tolerance of BNNs may be difficult to acquire, while software-level experts are usually not familiar with the details of the hardware and use simple assumptions to focus on their field. To overcome this challenge, both hardware and software expertise regarding BNNs is required in order to enable efficient HW/SW Codesign, where the SW and HW components of the BNN system are considered together in the design phase to find efficient synergistic solutions.

Considering together the BNN SW, i.e. increased error tolerance of BNNs using methods from Ch. 4, and the BNN HW, i.e approximate memory and computing units, it enables us to develop novel techniques that aggressively exploit the error tolerance for system efficiency. In these techniques, there are knobs on the software level (e.g. accuracy and error tolerance of BNNs) and on the hardware level (e.g. the energy, area, and latency of memory or computing units), which need to be identified and formalized such that the tradeoff regarding approximations and result quality can be explored effectively. After identifying the knobs, it needs to be known how to tune them in order to achieve as much efficiency as possible, while keeping the accuracy of BNNs high.

In this dissertation, our second goal is to consider the BNN SW and HW together to identify and tune the knobs of BNN systems jointly on the SW and HW level. Specifically, *our second goal is to exploit the error tolerance of BNNs for the design of efficient systems that employ approximate memory and computing units*.

We first identify the knobs and then propose methods for how to tune them in Ch. 5 for BNNs using approximate memory and in Ch. 6 for BNNs using approximate computing units. In Ch. 5 we focus on the emerging and highly promising FeFET memory and its tradeoffs when operating it as an approximate memory. In Ch. 6 we explore the design of approximate computing units that operate based on the principles of the analog instead of the classical digital domain.

Training BNNs on the Edge

In Sec. 1.1 we explained that BNN training is orders of magnitudes more resource intensive than BNN inference. The reason is that in the backward pass, prediction errors have to be backpropagated for gradients of parameters, which requires the computation of floating-point MAC operations for classical NNs and also BNNs.

Furthermore, in the training, certain kinds of training data needs to be intermediary stored, such as activations for constructing the backward pass graph and optimizer data to acquire updates to parameters that approach the optimum reasonably well.

Despite these challenges, the advantages of training on the edge, which are mainly efficiency and privacy, drive us to bring closer the vision of edge training for BNNs. However, directly aiming towards full on-chip training is very ambitious. A good way to optimize any system is to focus on the most resource-consuming bottleneck first. In NN and also BNN training, this is the memory.

In this dissertation, our third goal is to bring closer the vision of training BNNs on the edge. Specifically, *our third goal is to improve the memory-efficiency of the BNN training*.

In Ch. 7, we aim to increase the memory efficiency of the BNN training by proposing a method to acquire custom memory-efficient floating-point encodings for the optimizer data. We show that the optimizer data needs by far the largest amount of memory out of the data that needs to be stored during the BNN training. We then define memory-efficient FP encodings built upon on a proof-based metric that measures the number of dropped gradient updates due to the encoding employed.

1.3 CONTRIBUTIONS OF THIS WORK

This dissertation explores a vision for highly resource-efficient future intelligent systems, which run robust BNNs with approximate memory and approximate computing units, while being able to be trained on the edge. The list of contributions below describes the studies conducted to explore this vision.

- In Ch. 4 we present how to train BNNs for error tolerance without an error model by exploring the underlying principles of error tolerance. We first perform a theoretical exploration of BNNs error tolerance on the hidden-layer neuron level and then on the output-layer-level. We provide formal proofs to quantify the maximum number of bit flips that can be tolerated. With the proposed margin-based metrics and the well-known hinge loss for maximum margin classification in support vector machines (SVMs), we then construct a modified hinge loss (MHL) to train BNNs for robustness without any bit flip injections. Our results indicate that the MHL enables BNNs to tolerate higher bit error rates than with bit flip training. To the best of our knowledge, this is the first work that explores the underlying principles of BNN error tolerance in a theoretical manner and connects the error tolerance optimization of BNNs with margin-maximization.

- In Ch. 5, two studies are discussed, in which we focus on using FeFET memory as the approximate memory for the system executing the BNNs. In the first study (Sec. 5.2), we consider BNNs as on-chip memory and reveal the temperature-dependent bit error model of FeFET memories. We show that BNN accuracy drops to unacceptable levels under the errors. We explore two countermeasures:

(a) Training BNNs for bit error tolerance by injecting bit flips, and (b) applying a bit error rate assignment algorithm (BERA) which operates in a layer-wise manner and does not inject bit flips during training. In our experiments, the BNNs effectively tolerate temperature-dependent bit errors for the entire range of operating temperature for both methods. In the second study (Sec. 5.3), we consider FeFET-based XNOR gates as Logic-in-Memory (LiM) for BNNs, in which the FeFET-based XNOR LiM gates are the latency bottleneck. We investigate the probability of error in FeFET-based XNOR LiM, demonstrating the tradeoff between speed and reliability. Using our reliability model, we show how BNNs can be proactively trained in the presence of XNOR-induced errors towards obtaining robust BNNs at design time. Furthermore, we provide a runtime adaptation technique, that selectively trades off errors and XNOR speed for every BNN layer. Our results demonstrate that exploiting the tradeoff, significantly higher LiM latency can be achieved compared to the baseline. To the best of our knowledge, these two studies are the first to explore the tradeoff between FeFET reliability and BNN error tolerance.

- In Ch. 6, also two studies are discussed, which both exploit the error tolerance of BNNs for efficient analog-computing based hardware accelerating the BNNs' workloads. In the first study (Sec. 6.1), we reduce the resource demand of the interface circuit that performs the analog-to-digital conversion. Analog-computing BNNs hardware demands a large amount of analog-to-digital converters (ADCs) and registers, resulting in expensive designs. To increase the inference efficiency, the classical approach is to divide the interface circuit into an Analog Path (AP), utilizing cheap analog comparators, and a Digital Path (DP), utilizing expensive ADCs and registers. During BNN execution, a certain path is selectively triggered. Ideally, as inference via AP is more efficient, it should be triggered as often as possible. However, unless the number of weights is very small, the AP is rarely triggered. To overcome this, we propose a novel BNN inference scheme, called Local Thresholding Approximation (LTA). It approximates the global thresholdings in BNNs by local ones. This enables the use of the AP through most of the execution, which significantly increases the interface circuit efficiency. In the second study (Sec. 6.2), we optimize another analog computing scheme for BNNs, i.e. Integrate-and-Fire (IF) Spiking Neural Networks (SNNs). To achieve high inference accuracy in IF-SNNs, the analog hardware needs to represent current-based MAC levels as spike times, for which a large membrane capacitor is required. This results in high energy use, considerable area cost, and long latency, constituting one of the major bottlenecks in analog IF-SNN implementations. To alleviate this, we propose a HW/SW Codesign method, called CapMin. CapMin minimizes the capacitor size by reducing the number of spike times needed for accurate operation of the HW, based on the absolute frequency of MAC level occurrences in the SW. To increase the computation's tolerance to process variation, we propose CapMin-V, which trades capacitor

size for protection based on the reduced capacitor size found in CapMin. CapMin achieves significant reduction in capacitor size over the state of the art, while CapMin-V achieves increased variation tolerance, requiring only a small increase in capacitor size. To the best of our knowledge, these two studies are the first to exploit the error tolerance of BNNs for the design of efficient analog-computing hardware.

- In Ch. 7, we propose methods to enable the memory-efficient training of BNNs on the edge. We first investigate the impact of arbitrary floating-point (FP) encodings. When the FP format is not properly chosen, we prove that updates of the momentum values can be lost and the quality of training is therefore dropped. With this insight, we formulate a metric to determine the number of unchanged momentum values in a training iteration due to the FP encoding. Based on the metric, we develop an algorithm to find FP encodings that are more memory-efficient than the standard FP encodings. In our experiments, the memory usage in BNN training is significantly decreased with minimal cost in accuracy. To the best of our knowledge, this is the first study that increases BNN training efficiency by finding custom FP formats for the momentum values.

1.4 AUTHOR'S CONTRIBUTION TO THIS DISSERTATION

According to §10(2) of the "Promotionsordung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011", the dissertation must include statements about the author's contributions that resulted from cooperations with others. Below the contributions and statements are listed.

- The contents of Ch. 4 have been published in [Bus+20] and [Bus+21], which were written together with Sebastian Buschjäger, Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Christian Hakert, Katharina Morik, Rodion Novkin, and Lukas Pfahler. The work in [Bus+20] is the preliminary work for [Bus+21]. The conceptual idea for these papers originated from a project meeting in the SFB876-A1 project. The main authors of these papers are Sebastian Buschjäger, Lukas Pfahler, and me. I performed the experiments and conducted a theoretical investigation of the bit error tolerance of BNNs with the help of Mario Günzel, Sebastian Buschjäger developed the machine learning framework, and Lukas Pfahler had the idea on margin-maximization. The remaining authors contributed through discussions and gave feedback during writing.
- The contents of Sec. 5.2 and the introductory part in Sec. 5.1 have been published in [Yay+22a]. It was written together with Sebastian Buschjäger, Aniket Gupta, Jian-Jia Chen, Jörg Henkel, Katharina Morik, Kuan-Hsun Chen, and Hussam Amrouch. The idea for this paper originated from discussions with Hussam Amrouch and Kuan-Hsun Chen. The device-level materials regarding modeling and simulation of FeFET have been produced by Aniket Gupta and

Hussam Amrouch, therefore the figures related to the FeFET materials and the corresponding text segments are based on their materials. Sebastian Buschjäger developed the machine learning framework and its initial implementation. I enhanced and adapted it for the novel methods in the paper and performed the experiments for evaluation. The remaining authors contributed through discussions and gave feedback during writing.

- The contents of Sec. 5.3 have been published in [Yay+22b]. It was written together with Simon Thomann, Sebastian Buschjäger, Katharina Morik, Jian-Jia Chen, and Hussam Amrouch. The idea for this paper originated from discussions with Hussam Amrouch. The device-level materials regarding modelling and simulation of the FeFET-based XNOR gates have been produced by Simon Thomann and Hussam Amrouch, therefore the figures related to the FeFET-based XNOR gates materials and the corresponding text segments are based on their materials. Sebastian Buschjäger developed the machine learning framework and its initial implementation. I enhanced and adapted it for the novel methods in the paper and performed the experiments for evaluation. The remaining authors contributed through discussions and gave feedback during writing.
- The contents of Sec. 6.1 have been published in [Yay+23c]. It was written together with Fabio Frustaci, Fanny Spagnolo, Jian-Jia Chen, and Hussam Amrouch. The idea for this paper originated from discussions with Hussam Amrouch. The area, energy, and latency estimations of the digital components (using FDSOI technology) as well as the selection of analog components for reference were contributed by Fabio Frustaci and Fanny Spagnolo. The methods were conceptualized together with Hussam Amrouch and Fabio Frustaci. I performed the experiments to evaluate the methods. The remaining authors contributed through discussions and gave feedback during writing.
- The contents of Sec. 6.2 have been published in [Yay+23b]. It was written together with Simon Thomann, Ming-Liang Wei, Chia-Lin Yang, Jian-Jia Chen, and Hussam Amrouch. The idea for this paper originated from discussions with Simon Thomann and Ming-Liang Wei. The device-level data regarding the XNOR-array was contributed by Simon Thomann and Hussam Amrouch, whereas the initial SPICE model was created by me with the help of Ming-Liang Wei and Simon Thomann. I performed the experiments to evaluate the methods. The remaining authors contributed through discussions and gave feedback during writing.
- The contents of Ch. 7 have been publised in [YC22], which were written together with Jian-Jia Chen. The conceptual idea for the paper originated from discussions with Jian-Jia Chen. I conducted theoretical analyses and performed the experiments, whereas Jian-Jia Chen contributed through discussions and gave feedback during writing. For materials that are not in [YC22]: The evaluations for the data in Fig. 7.4 were contributed by my student Robert Huber, who was working as one of my student research assistants.

2

BACKGROUND

In this chapter, we cover the background and related work of this thesis. In Sec. 2.1, we introduce the neural networks (NNs), their inference, training, and most impactful ideas for efficiency. In Sec. 2.2, we introduce BNNs and give an overview of state-of-the art methods and tools of how to construct them and deploy them efficiently. In Sec. 2.3, we introduce the different types of hardware to deploy BNNs.

2.1 NEURAL NETWORKS

Neural Networks (NNs) are universal approximators, meaning they can approximate any continuous function [Cyb89; KB20] with some theoretical limitations (which are rarely relevant in most real-life problems). Since the last decade, NNs have been applied in multiple fields and in various applications. In essence, NNs are composed of one or multiple layers, where each layer consists of a certain number of neurons. Typically, the topologies of modern NNs is deep, with many layers and a manageable number of neurons per layer, which makes the inference and training more efficient compared to shallow and wide NNs. In this dissertation, we focus on feedforward NNs, where there only exist forward connections and no backward connections.

This section is structured as follows. In Sec. 2.1.1, we introduce the notation for describing the operations of NNs. In Sec. 2.1.2, we introduce the basics of inference and training in NNs. In Sec. 2.1.3, we discuss the challenge of efficiency in NNs.

2.1.1 *Notations and Matrix Dimensions*

We introduce the following conventions for notation of NNs. We assume for a convolution layer of an NN a weight matrix $\mathbf{W}$ with dimensions $(\alpha \times \beta)$, where α is the number neurons and β the number of weights of a neuron. The activation matrix $\mathbf{A}$ has dimensions $(\gamma \times \delta)$, where $\beta = \gamma$ (i.e., matrix multiplication between $\mathbf{W}$ and $\mathbf{A}$ can be performed) and δ is the number of convolution windows, i.e., unfolded kernels in the input. Every convolution (1D, 2D, etc.) of a conventional NN can be mapped to this matrix notation.

In general, each convolution layer in an NN (fully connected, 2D convolution, other convolution types) computes its outputs by performing the matrix multiplication $\mathbf{W} \times \mathbf{A}$, resulting in an output matrix with dimensions $\alpha \times \delta$. A matrix multiplication is performed by scalar products of different combinations of rows from $\mathbf{W}$ and columns from $\mathbf{A}$. These scalar products are the MAC operations.

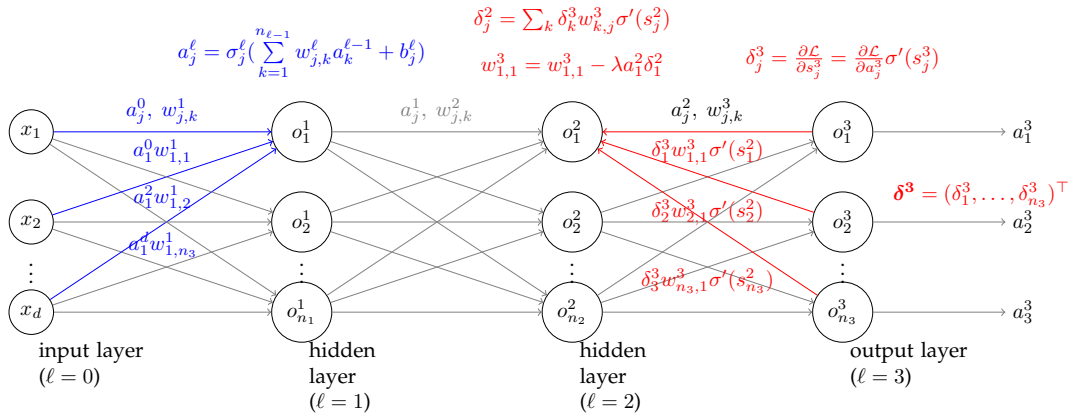


Figure 2.1: Illustration of the forward (blue) and backward (red) passes in NNs.

When convenient, we add layer indices or introduce more notation in the following parts to refer to elements in the NN in a more fine-grained way, e.g. to address the computations of single neurons and activations.

2.1.2 *Inference and Training of NNs*

In the following we use layer indices and address neurons individually in order to describe the principles of the forward propagation, also referred to as inference, and the backward propagation, also referred to as backpropagation. The materials in this section are based partially on the work in [GBC16] and [Nie15]. Specifically for the concise notations, we adopt the ideas proposed by Nielsen [Nie15].

2.1.2.1 *Inference*

NNs mainly consist of parameters and connections, which allow information to flow forward in the inference. The forward computations are illustrated in blue in Fig. 2.1 and in the following we provide a description. $w_{j,k}^\ell$ is the weight that connects the k-th neuron in layer $\ell - 1$ to the j-th neuron in layer ℓ. Usually, each neuron also has a bias term b_j^ℓ. The term $a_j^{\ell-1}$ is the activation of a neuron j in layer $\ell - 1$.

$$s_j^\ell = \sum_{k=1}^{n_{\ell-1}} w_{j,k}^\ell a_k^{\ell-1} + b_j^\ell \tag{2.1}$$

describes the inner product computations of a neuron j, which has $n_{\ell-1}$ weights. $a_j^\ell = \theta(s_j^\ell)$ describes the activation of neuron j, where θ is the activation function.

We also introduce matrix notations for a layer ℓ for simpler notations. $\mathbf{W}^\ell$ is a matrix with a row for each neuron in the layer and as many columns as weights in a neuron. $\mathbf{B}^\ell$ has the biases, there are as many biases as neurons in layer ℓ. $\mathbf{A}^{\ell-1}$ has the activations of layer $\ell - 1$. $\mathbf{S}^\ell$ is a matrix resulting from the matrix product between $\mathbf{W}^\ell$

and $\mathbf{A}^{\ell-1}$, with $\mathbf{B}^{\ell}$ added after the multiplication. After the element-wise activation has been performed on $\mathbf{S}^{\ell}$, the activation matrix $\mathbf{A}^{\ell}$ is acquired. The operation of layer can then be summarized with $\mathbf{A}^{\ell} = \theta(\mathbf{S}^{\ell}) = \theta(\mathbf{W}^{\ell}\mathbf{A}^{\ell-1} + \mathbf{B}^{\ell})$, whereas θ operates in an element-wise manner on all matrix entries.

2.1.2.2 *Training*

The objective of the training is to reduce as much as possible the loss $\mathcal{L}$, which is used to quantify the difference between the predictions of the NNs and the ground truth of the given training data. We denote the training data as $\mathbf{D} = \{(x_1, y_1), \ldots, (x_I, y_I)\}$ with the inputs x_i and the ground truth labels y_i. With $f_{\mathbf{W}}(x)$ as the NN output, the goal is to solve

$$\arg\min_{\mathbf{W}} \frac{1}{I} \sum_{(x,y)\in\mathbf{D}} \mathcal{L}(f_{\mathbf{W}}(x), y) \tag{2.2}$$

With the loss, the performance of the model can be evaluated. When the loss decreases with a higher number of training iterations, then the training is considered to be progressing successfully. Since NNs are computation graphs with differentiable components, the parameters in NNs can be optimized by computing partial derivatives. With partial derivatives of the loss with respect to the parameters, the NN model parameters can be moved towards the opposite direction of the gradient, i.e. the direction of the steepest descent, which leads to the minimization of the loss.

To train NNs, i.e., to update the weights based on the loss $\mathcal{L}$, the partial derivatives of the loss with respect to the corresponding weights need to be computed, i.e.

$$\frac{\partial \mathcal{L}}{\partial w_{j,k}^{\ell}}. \tag{2.3}$$

The backward computations are illustrated in red in Fig. 2.1 and in the following we provide explanations.

To calculate Eq. (2.3), several computation steps with the chain rule need to be performed first. For convenience, we define the term $\delta_j^{\ell} = \frac{\partial \mathcal{L}}{\partial s_j^{\ell}}$. To compute δ_j^{ℓ}, the derivative needs to be computed by the chain rule. For starting the backward propagation, the last layer $\ell = L$ has to be considered. There, $\delta_j^L = \frac{\partial \mathcal{L}}{\partial a_j^L}\frac{\partial a_j^L}{\partial s_j^L} = \frac{\partial \mathcal{L}}{\partial a_j^L}\theta'(s_j^L)$, the entries of which can be denoted in a vector $\boldsymbol{\delta}^{\boldsymbol{L}}$ for the error terms in layer L. The term $\frac{\partial \mathcal{L}}{\partial a_j^L}$ can be obtained analytically when the loss function is known. The other error terms for the remaining $L-1$ layers need to be computed as well. Note that a neuron with product s_j^{ℓ} influences all neurons in layer $\ell+1$. Therefore, to calculate the error, the sum of the partial derivatives of each neuron influenced by s_j^{ℓ} needs to be computed, which are all the neurons $s_k^{\ell+1}$. This is $\delta_j^{\ell} = \sum_k \frac{\partial \mathcal{L}}{\partial s_k^{\ell+1}}\frac{\partial s_k^{\ell+1}}{\partial s_j^{\ell}}$. Since $\delta_j^{\ell+1} = \frac{\partial \mathcal{L}}{\partial s_j^{\ell+1}}$ (per definition) and $\frac{\partial s_k^{\ell+1}}{\partial s_j^{\ell}} = \frac{\partial s_k^{\ell+1}}{\partial a_j^{\ell}}\frac{\partial a_j^{\ell}}{\partial s_j^{\ell}} = w_{k,j}^{\ell+1}\theta'(s_j^{\ell})$ (the index k is fixed, j

is the neuron index, therefore the $w_{k,j}^{\ell+1}$ is the derivative), it can also be written as $\delta_j^\ell = \sum_k w_{k,j}^{\ell+1} \delta_k^{\ell+1} \theta'(s_j^\ell)$. Note that $w_{k,j}^{\ell+1} \delta_k^{l+1}$ is the matrix operation $(\boldsymbol{W}^{\ell+1})^\top \delta^{\ell+1}$. The weight gradients are then obtained by $\frac{\partial \mathcal{L}}{\partial w_{j,k}^\ell} = a^{\ell-1} \delta_j^\ell$. The gradients for the bias values b_j are obtained similarly using the term $\frac{\partial \mathcal{L}}{\partial b_j^\ell} = \delta_j^\ell$, since deriving with respect to the bias results in a "1" for the bias, and zero for every other term, therefore only the partial derivatives in δ_j^ℓ that lead to b_j^ℓ are needed.

In Alg. 1, we present the typical learning procedure for NNs. We omit indices for simplicity of presentation. The inputs to the algorithm are specified on the top. In Line 1, the weights and momentum values (e.g. exponential moving averages of the weights, explained in the description of Line 5 below) are initialized, for which different techniques exist, such as random initialization or more sophisticated approaches [Bou+22]. One training iteration over the entire training data is referred to as an epoch. The number of specified epochs is processed in Line 2. As explained above, the general idea of the training is to move the NN model parameters towards the opposite direction of the gradient to minimize the loss. In practice, instead of performing this for the entire training data in one step, stochastic gradient descent (SGD) is employed to save memory and computation resources. In SGD, the gradient is approximated by using a subset, i.e. batch of the training data. Therefore, data is sampled from the training set in form of random batches of a certain size. In each epoch, the NN approximately learns with as many training samples as there are in the training set. Based on each batch, a gradient is computed in Line 4. Subsequently, the gradients are processed using a method R and then the weights are updated. The parameter γ is typically reduced by a certain factor after each epoch, it is also referred to as the learning rate.

Training with an R that is the identity function may lead to unstable or no convergence in the training. Therefore, different types of optimizers to process the gradients exist. One example is the Adam optimizer [KB14], which uses, among other techniques, exponential moving averages of the gradient updates over the training steps. It is currently the standard optimizer in a wide range of applications.

2.1.3 *Efficient Neural Networks*

In [Sze+17], an overview of methods for the efficient processing of NNs is given. Additional possibilities are pruning [Lia+21], compression [MGD20], and quantization [Gho+21; Kul+22]. Specifically, one of the most common, intuitive, and easily applicable methods for achieving efficiency is quantization. In quantization, certain elements in a set (may be infinite) are represented by a smaller set (finite) of elements. This concept underpins any kind of data processing, most notably when signals from the real world (typically analog) need to be processed by a computer, where the data needs to be quantized to enable useful processing with the signals and to increase efficiency in storage and computation. Quantization always incurs some information

Algorithm 1: Training Algorithm for NNs

Input: NN model f_W, weights $\mathbf{W}$, loss $\mathcal{L}$, training data (X_{train}, y_{train}), adaptivity rate γ, batch size B_s, epochs E, batches B

1 Initialize $\mathbf{W}$
2 **for** *each epoch* $e = 0, \ldots, E$ **do**
3 **for** *each batch* $b = 0, \ldots, B$ **do**
 `// Compute gradients`
4 $\mathbf{G} \leftarrow \frac{1}{B_s} \frac{\partial \mathcal{L}}{\partial \mathbf{W}} \sum_{(x,y) \in batch} \mathcal{L}(f_{\mathbf{W}}(x), y)$
 `// Update weights`
5 $\mathbf{W} \leftarrow \mathbf{W} + \gamma R(\mathbf{G})$

loss, but the information that is "quantized away" may not be important or could even be neglected [OS89].

In NNs, the information loss due to quantization does not lead to high accuracy drops. In many cases, the quantization has no noticeable impact on the accuracy at all. The quantization of NNs has been proposed in many works, e.g. [CBD15a; Sze+17; Jac+] and has since then been used in many other following works, studies, and in practice, in both inference [Gho+21] and training [CBD15b; Ban+18; Kal+19; Sun+20; Wan+21]. The quantization has many benefits for the efficiency of the NN system. The memory needed to store the values is reduced when, for example instead of 32 bit, 8 bit can be stored instead. Due to the reduced number of bits, the costs of data transfers are (e.g. reads and writes to memory, moving data across the memory hierarchy) reduced as well. Furthermore, the computations are simplified. Instead of floating point or high-bit integer logic, simpler logic can be employed for higher efficiency.

2.2 BINARIZED NEURAL NETWORKS

In BNNs, which were introduced in 2016 in [Hub+16; Ras+16; KS16], the most extreme form of quantization is employed: Binarization. The weights and activations in BNNs are binarized, which enables inference with small area usage, low latency, and low energy, enabling the inference on highly resource-constrained edge devices.

Since their inception, the use of BNNs has been explored for various applications where real-time, low-latency response and resource efficiency is critical. They have been applied in real-time image classification [C+22], for recognizing objects that are moving at a high speed [Zha+23b], to control autonomous vehicles [WHA18], and efficient traffic sign recognition [PE23]. Other application that require real-time inference and have been explored with BNNs are text classification [Shr+20] and human activity recognition [Luo+23]. Furthermore, they have been used in the wild for wildfire detection [CP23] and agricultural settings [Hua21], where resources are scarce and efficient processing is necessary.

Similar to traditional NNs, BNNs are also universal approximators. The findings for traditional NNs [Cyb89] have been extended to quantized and binarized convolutional NNs as well [Wan+18; DLS18; Spa+19]. Furthermore, the work in [Yay+21] has closed the gap to fully connected BNNs and shows that two-hidden layer fully connected BNNs possess the universal approximation capability. These studies prove that BNNs possess universal approximation property and therefore can learn any task (under some theoretical restrictions on the task which usually are not relevant in practice).

Tools and frameworks have also already been built to enable efficient BNN inference on various hardware platforms. In [Umu+17] the framework FINN has been proposed. Its purpose is to map the computations of BNNs to the resources of FPGAs. It promises easy usage and efficient acceleration for BNN inference. In [Bet+18], the BMXNet framework is proposed, which is built on top of the well-known MXNet library [Che+15]. The framework enables the creation of BNNs and provides functionality to produce C/C++-code for BNNs to deploy them efficiently on edge devices. In [Ban+21], the Larq compute engine (LCE) is proposed. It is a framework which enables training, benchmarking, and deploying BNNs in a joint manner. Its main focus are mobile devices with 64-bit ARM processors, which are included in the Raspberri Pi and various Android phones. The work in [Zha+19a] also focuses in ARM devices and promises high efficiency by employing for example bit-packing and memory layout optimization.

Note that there is a plethora of studies on BNNs since their introduction and a systematic surveys on BNNs have been conducted in [SL19; Zha+21; Say+23; YA23]. In the following, we give an introduction to the basics of BNN, while also summarizing the ideas of the most impactful studies that are related to this dissertation.

Specifically, in Sec. 2.2.1, we disscuss the BNN inference. In Sec. 2.2.2, we introduce how BNNs are trained. In Sec. 2.2.3, we cover how to binarize the inputs of BNNs. In Sec. 2.2.4, we discuss the available methods to obtain BNNs. Finally, in Sec. 2.2.5, we discuss the error tolerance optimization of BNNs.

2.2.1 *BNN Inference*

Since the weights and activations in BNNs are binarized to $\{\pm 1\}$, the multiplications can be performed by performing XNOR. This enables the use of simple logic for the BNN computations. Therefore, in practice, the binarization to $\{0, 1\}$ is used for efficient logic-based execution of the multiplication and accumulation (MAC) operation of neurons. In this case, the multiplication, summation, and activation can be computed with

$$2 * \text{popcount}(\text{XNOR}(\mathbf{W}, \mathbf{A})) - \#bits > \mathbf{T}, \tag{2.4}$$

where $\text{XNOR}(\mathbf{W}, \mathbf{A})$ computes the XNOR of the rows in $\mathbf{W}$ with the columns in $\mathbf{A}$ (analogue to matrix multiplication), popcount counts the number of set bits in the XNOR result, $\#bits$ is the number of bits of the XNOR operands, and $\mathbf{T}$ is a vector of learnable threshold parameters, with one entry for each neuron. Note that the result

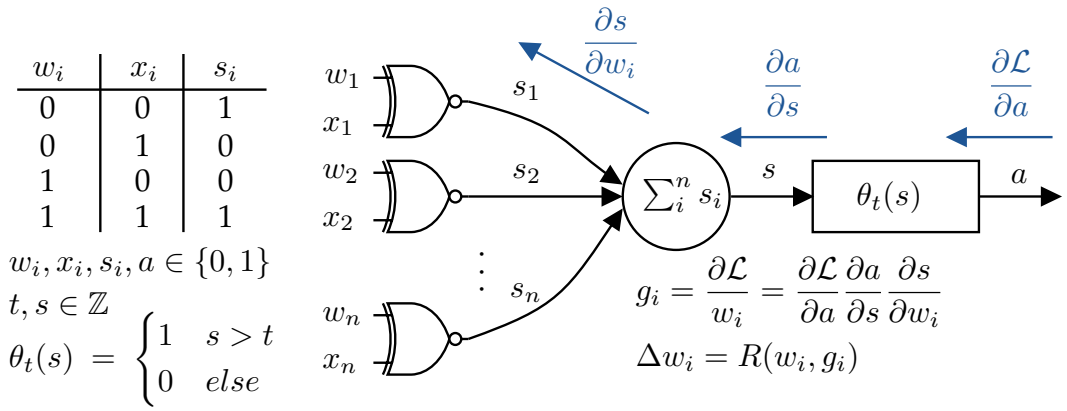


Figure 2.2: Overview of BNN inference and training in one neuron with binarized weights (w_i) and inputs (x_i). On the left is the table for performing the logical XNOR. After the XNOR (gates) and popcount operation (sum of products), the popcount result (s) is compared against a threshold (t) to produce a binary output (a). The blue arrows indicate the backward pass operations in training, where the chain rule is applied to compute the gradient g_i, which is processed in the function R to acquire the weight update Δw_i.

of popcount is multiplied by two and then *#bits* is subtracted, by which the result of popcount is transformed into a value that would have resulted if the computations were performed using the binarization $\{\pm 1\}$. Instead of transforming the result of popcount, the threshold can also be adapted instead. The thresholds are computed with the batch normalization parameters (BN), i.e., $T = \mu - \frac{\theta}{\psi}\eta$, where each neuron has a mean μ and a standard deviation θ over the result of the left side of Eq. (2.4), and ψ and η are learnable parameters. For further details about the BN please refer to Sec. 2.2.2 or [Hub+16; SBN19]. Finally, the comparisons against the thresholds produce again binary values.

2.2.2 *Training BNNs*

The simulation of BNNs for inference and training in order to create models is shown in Alg. 2. The algorithm loops over all data points and all layers in the BNN. In Line 3, the (full-precision) weights are binarized to $\{-1, +1\}$. Due to this, the matrix product can be computed with the already existing MAC libraries in high-level machine learning framework such as PyTorch or Tensorflow. Then, the $\mathbf{S}^\ell$ are computed (biases are all set to zero here). Afterwards, the BN function is applied. Finally, the output of the BN is binarized to $\{-1, +1\}$, which serve as the binarized input activations for the subsequent layer computations.

Algorithm 2: Algorithm for the Simulation of BNN Inference for Training

Input: Weights $\mathbf{W}$, training data (X_{train}, y_{train}),
1 **for** *each data point* $(x, y) \in (X_{train}, y_{train})$ **do**
2 **for** *each layer* $\ell \in 1, \ldots, L$ **do**
3 $\mathbf{W}^{\ell}_{\mathbf{bin}} \leftarrow Binarize(\mathbf{W}^{\ell})$
   `// Note: Input` x `is` A^0
4 $\mathbf{S}^{\ell} \leftarrow \mathbf{W}^{\ell}_{\mathbf{bin}} \mathbf{A}^{\ell-1}$
5 $\mathbf{A}^{\ell} \leftarrow Binarize(BatchNorm(\mathbf{S}^{\ell}))$

2.2.2.1 *Key Concepts for Training BNNs*

In the inference of BNNs, the BN is applied as a simple thresholding. When training BNNs, i.e. when executing the backward pass, it cannot be regarded as thresholding and needs to be considered as a transformation. The BN layer is typically applied after the convolution layers and is defined for each neuron j on layer ℓ as the transformation

$$BatchNorm(s_j^{\ell}) = \hat{s}_j^{\ell} = \psi_j \left(\frac{s_j^{\ell} - \mu_j^{\ell}}{\sqrt{\sigma_j^{\ell} + \epsilon}}\right) + \eta_j, \tag{2.5}$$

where ψ_j is a scaling factor and η_j is for translation. The nominator centers the s_j^{ℓ} around zero while the denominator normalizes it, where ϵ is used for numerical stability. The term μ_j^{ℓ} and σ_j^{ℓ} are the mean and the variance of the s_j^{ℓ} over the batch size. Due to the differentiable subcomponents of the BN, the derivatives can be calculated in a straightforward manner [IS15]. Note that training BNNs without BN leads to poor prediction accuracy. Specifically, the study in [SBN19] argues that BNN training without the BN leads to exploding gradients and therefore infeasible training, which makes the BN layer necessary for successful BNN training.

Another important issue in BNN training is that the activation function is the binary threshold function (Line 5 in Alg. 2), for which the derivative is zero for almost all inputs. Since nonzero derivatives are necessary for successful training, the straight-through-estimator (STE) is used instead [BLC13]. In essence, the STE just passes the gradients backwards, as if no activation function exists, and clips the values between ± 1. Although this is only an approximation, it leads to good progress in the training procedure.

2.2.3 *Stochastic Input Binarization in BNNs*

The inputs of any NN are usually encoded as floating point or quantized values (i.e., non-binarized). This makes the execution of the first layer (unlike other hidden layers) unsuitable to be performed with XNOR-based operations. To overcome this problem, stochastic input binarization has been proposed [Hir+19c]. It employs

Algorithm 3: First layer computations of a BNN with stochastic input binarization

Input: Input $\mathbf{A^0}$, repetitions nr_{pres}, scaled thresholds $\tilde{\mathbf{T}}$
Output: $\mathbf{A^1}$

1 $reps = nr_{pres}$
2 $\mathbf{A_b^0} = binarize_{stoch}(\mathbf{A^0})$
3 $\mathbf{A^1} = \mathbf{W^1 A_b^0}$
4 **while** $reps > 0$ **do**
5 $\quad \mathbf{A_b^0} = binarize_{stoch}(\mathbf{A^0})$
6 $\quad \mathbf{A^1} = \mathbf{A^1} + \mathbf{W^1 A_b^0}$
7 $\quad reps = reps - 1$
8 $\mathbf{A^1} = \mathbf{A^1} > \tilde{\mathbf{T}}$

stochastically binarized inputs, which are achieved by normalizing the input values to values between 0 and 1. These values are interpreted as probabilities and are then stochastically rounded to "0" or "1". For controlling the loss of information due to the stochastic input binarization, the number of input presentations can be tuned. With this parameter, it is possible to configure the number of times the input is sampled and processed by the first layer. In case the number of repetitions of training is not sufficient, the stochastic binarization may lead to accuracy degradation. However, with sufficient repetitions, the output of the first convolutional layer with stochastically binarized inputs can become arbitrarily close to the output of a convolutional layer with normalized input. The rules of computation for stochastic computing in the first layer are shown in Alg. 3. X^0 represents the input, nr_{pres} the number of input presentations, and $\tilde{T}$ is acquired by multiplying T by nr_{pres}. In Line 2, the input is stochastically binarized, and in Line 3, matrix product is computed. In the subsequent steps, these two operations are repeated based on the number of repetitions specified. When *reps* is depleted, a binarized output is acquired after thresholding in Line 8. During BNN training, the stochastic binarization is analogously applied.

2.2.4 *How to Obtain BNNs*

One method to obtain BNNs is to use standard off-the-shelf NN architectures, such as VGG, ResNet, MobileNet, Yolo, etc. [SZ14; He+16; How+17; Red+15]. Then, the weights and activations of these models are binarized. Note that for achieving high accuracy, binarization-aware training has to be performed, i.e. the binarization has to be simulated during the training process, as it is performed in Alg. 2 and [Jac+; Wen21]. If the binarization is performed after training, i.e. in a post-quantization manner, then the model becomes useless since the accuracy drop becomes too high. Another important component, as mentioned in Sec. 2.2.2.1, are the BN layers, which are reported to be necessary for successful BNN training [Hub+16; SBN19]. Furthermore,

components may need to be replaced as well, such as binary activation functions instead of other activation function such as those related to ReLU. Although the prediction accuracy of the BNN may be a few percentage points lower than the traditional NNs, usually high accuracy is still achieved. Different methods exist to increase the accuracy loss from binarization, such as knowledge distillation [Gou+21], where the learned knowledge of larger or multi-bit models are distilled into a smaller or binary NNs.

Another way is to acquire BNN architectures is by employing neural architecture search (NAS) [Gou+21]. In NAS, a certain search space is defined, and then a search algorithm (such as genetic algorithm) is used to find BNN topologies and their subcomponents, from which efficient BNN architectures are constructed automatically [KSC20; BMT20; Wu+20]. However, NAS is typically costly, as the search process is highly resource intensive.

2.2.4.1 *Improvements of the BNN Training Procedure*

Several studies have investigated how the BNN training procedure can be improved. In [Bet+19], several approaches from the training of high-precision NNs are evaluated for BNNs. Their conclusion is that many of the common methods are not suitable for NNs. Instead, the training methods should focus on a information flow during training without bottlenecks. This study in [Xu+21] also claims that forcing updates for weights that are barely updated in the training process increases accuracy. One other notable example to achieve this is by increasing the number of shortcut connections. In [Ali+18] several pointers for best practices regarding BNN training are given based in empirical evaluations.

The study in [Hel+19] argues that the classical approach to train BNNs using full-precision weights (i.e. latent weights) and binarizing them during the forward pass in conjunction the Adam optimizer is not suitable for BNNs and leads to suboptimal performance. They argue that a special optimizer should be used for BNNs, which does not rely on binarizing the latent weights and the Adam optimizer. Therefore, they propose the binary optimizer (Bop), which interprets the latent weights as inertia and performs binary flipping decisions based on them for training. They show successful training results with higher accuracy than the previous methods.

Following a different branch, in [He+20], the BNN training procedure is modified to decrease the quantization error between the latent weights and the binarized weights. The study in [Lin+20] expands on this and proposes to additionally employ the angular bias to reduce the quantization error between the latent and binarized weights. To put it in simple terms, for optimization they rotate the latent weights with a rotation matrix into binary weights. They report that their methods maximize the information gain during training, leading to higher training accuracy compared to the previous approaches. In [BT19], scaling factors for the outputs if binary matrix multiplications are proposed which are reported to improve training performance.

The study in [Mar+20] goes further by using full-precision activations during BNN training and in [Tu+22] adaptive per-layer binary sets are proposed.

It has recently also been shown that NNs can be optimized to alleviate catastrophic forgetting, i.e., they are capable of remembering previously learned tasks, even after they have been optimized for another task, see [Lab+21].

2.2.5 *Error Tolerance of BNNs*

In NNs, error tolerance is the ability to achieve high prediction accuracy despite errors in the weights, inputs, activations, and other data that NNs use [TG17]. The errors are in the form of bit flips that change the stored or processed information. The cause of the errors may be noise (e.g. interference from the environment, particle strikes in space), temperature (e.g. in high temperature environments), or even purposefully created errors due to aggressive control of energy supplies and timing with the aim of efficient operation at the cost of errors.

The error tolerance of classical NNs has been evaluated by various studies. The survey in [TG17] provides a comprehensive overview of the recent and further back work about fault and error tolerant NNs. For BNNs, a survey has summarized the tolerance to single or multi bit upsets, which happen in in extreme conditions such as in space or hot environments [KBB20].

The error tolerance of BNNs to more general errors, however, was first evaluated only in 2019 [Hir+19b]. BNNs can achieve much higher error-tolerance compared to traditional NNs, while having the ability to perform training with errors, also referred to as error-aware training. Traditional NNs use floating-point (e.g., 32 bits) or integer values (e.g., 8 bits) to represent the NN parameters (i.e., weights, activations, inputs, etc.). In such a case, the position of the occurred error (i.e., the bit flip in the value) matters. In floating-point NNs, one bit error in one weight can cause the prediction of the NN to become useless [Kop+19]. This typically occurs when a bit flip in the exponent of the floating point representation occurs leading to an error with an unacceptable magnitude. On the other hand, in BNNs, a flip of one bit in a binary weight or binary input causes a change of the computation result by merely 1 (with binarization to $\{0, 1\}$). Additionally, the output of every neuron in the hidden layers is binarized, which has a saturating effect. Traditional NNs that for example use ReLU or related activation function typically do not have this effect.

To make NNs and BNNs bit error-tolerant, the classical method is bit flip injections in the binarized values during the forward pass, as proposed in [Hir+19b]. The idea is simple: To make BNNs robust against certain types of bit errors, exactly these errors are simulated during training time. To this end, during each forward pass, a random bit flip mask is generated and applied to the binary weights. More formally, let $\mathbf{M}$ denote a random bit flip mask with entries ± 1. It is of the same size as $\mathbf{W}$. The matrix $\mathbf{M}$ is multiplied component-wise to the weight matrix $\mathbf{W}$, i.e. the matrix operations are denoted as $(B(\mathbf{W}) \cdot \mathbf{M}) \times \mathbf{X}$, were $B(\mathbf{W})$ binarizes $\mathbf{W}$, and $\mathbf{X}$ are the inputs in ± 1.

Note that there is not as much related work on the error tolerance of BNNs compared to traditional NNs. Therefore, this thesis focuses as well on the error tolerance analysis of BNNs. Specifically, how to exploit the error tolerance of BNNs for designing efficient hardware is one of the key topics in this thesis.

2.3 HARDWARE SYSTEMS FOR BNNS

As explained above, BNNs are extremely resource-efficient and hardware friendly. In BNNs, the memory needed to store the parameters is significantly reduced compared to traditional NNs. When compared to 32 bit floating-point or 8 bit integer-based NNs, the memory reduction is also close to $32\times$ or $8\times$ respectively. This reduces the communication overhead in the memory hierarchy, leading to fast data transfers, while providing more ways to exploit the principles of temporal and spacial locality.

With the use of binary weights and inputs for computation, the costly MAC operations are performed with simple XNOR and popcount operations. This enables the use of specialized instructions for the operations, such as bitwise XNOR and using intrinsic popcount instructions in off-the-shelf CPUs or GPUs.

Hardware to accelerate the operations of BNNs can be realized in different forms. The operations can be mapped to certain CPU or GPU instructions, or special purpose accelerators such as FPGAs and ASICs can be used. A detailed comparison of hardware systems to execute BNNs on CPUs, GPUs, FPGAs, and ASICs is given in [Nur+16]. In all these systems, BNN data (mainly weights and inputs) need to move from the memory to the computing elements. The memory and the computation units are also the main resource bottleneck in NN and BNN systems. Although the resource use is reduced in BNNs due to the binarization, still a lot of data needs to be moved and processed in binary form. For highly resource-constrained and timing sensitive edge application, it is still necessary to find and evaluate approaches to further reduce the cost of resources for BNN systems.

To reduce the resource use in the memory and the computing units, approximations can be employed. In approximate computing, the quality of the result is traded with the efforts expended. Although the quality of the results in NNs and BNNs is high, achieving perfect accuracy of (e.g. "100%") is a challenge, therefore reasonable accuracy results (which may be way below "100%") are accepted and also sufficient for many applications. Therefore, in NNs and BNNs, approximations can be applied to trade the result's quality with the efficiency of the system.

In this thesis, we consider that that the BNN hardware system has approximate memory and approximate computing units. Therefore, we separate this section into two parts: Approximate memories used for BNNs in Sec. 2.3.1 and approximate computation units used in BNN hardware systems in Sec. 2.3.2.

2.3.1 *BNNs with Emerging Approximate Memory: Technologies and Techniques*

Approximate memories are realized by tuning the memory parameters, such as the memory supply voltage and latency parameters, at the cost of bit errors. Although an approximate memory can achieve lower power consumption and faster access, it can cause very high bit error rates, which may degrade computing accuracy in NNs significantly. This trade-off has been explored for a variety of state-of-the-art and emerging memory technologies, i.e. for non-volatile and volatile memories.

In this section we review studies that mainly exploit the reduced requirements of NNs on non-volatile memory technologies in Sec. 2.3.1.1, i.e. RRAM, STT-RAM or MRAM, FeFET and CTF, and the volatile memory technologies SRAM and DRAM in Sec. 2.3.1.2.

2.3.1.1 *Non-volatile Memories (NVMs)*

BNNs synergize outstandingly with NVM. As NVM cells only consume energy when they are used (i.e. when the states switch), while their energy use is near zero during idle times. Due to their non-volatility, they are significantly more energy saving compared to traditional volatile memories such as SRAMs and DRAMs, which have high static power. Furthermore, NVMs are typically feature a high density that allows accommodating much more data within the same area footprint compared to traditional memories. Using NVMs for BNNs has the potential to considerably reduce the overall system energy. An overview of all classical and emerging NVMs is given in [Bou+17]. Here, we consider the NVMs RRAM, MRAM or STT-RAM, and FeFET, since they have been considered for the use with BNNs.

RRAM: The closest works are about error tolerant BNNs that operate with reduced requirements on the memory in order to benefit in terms of power, performance, area, lifetime, etc. The study in [Hir+19b] proposes to compute BNN operations with RRAM that features in-memory processing capabilities. They set the write energy of RRAM low and show that BNNs can tolerate the resulting errors by error tolerance training. This low energy setting also increases the RRAM cell lifetime since the low energy writes stress the cells less. In [Yu+16], RRAM is used to implement on-chip BNNs. They show that under limited bit yield, BNNs can still operate with satisfying accuracy. The study in [Sun+18] proposes a RRAM synaptic array to deploy BNNs. They investigate the accuracy impact of errors from sense amplifiers that have intrinsic offset due to process variation.

MRAM or STT-RAM: Another branch in the literature is about BNNs on STT-RAM or MRAM, which both operate based on the principles of magnetoresistance. The study in [Hir+19a] proposes deploying BNNs on MRAM with a low energy programming setting that causes relatively low error rates, no significant accuracy drop, but decreases write energy by a factor of two. The study in [TGW19] also propose operating BNNs on MRAM with reduced voltage with similar results. They test a wide range of error rates and discuss the implications of BNN bit error tolerance

on lifetime, performance, and density of MRAM. In [Pan+18], a different approach is taken for energy reduction, they investigate the benefits of multi-level cell MRAM for in-memory acceleration of BNNs.

FeFET: The studies in [Che+18; Sol+20b] explore the in-memory processing capabilities of FeFET for BNNs and compare it to other CMOS-based circuits. However, they do not consider approximate FeFET memory and to the best of our knowledge, no such study exists.

2.3.1.2 *Volatile Memories*

Traditional volatile memory that is employed in today's computing systems mainly use classical SRAM and DRAM. Due to their volatility, they lose the information once the energy is cut. SRAM memory is typically used as fast on-chip memory (e.g. in registers and caches), which not only have high cost, but also which suffer from high leakage power and large area footprint (six transistors to store a single bit). DRAM, another classical memory, typically used for larger memories such as main memory, is composed of transistors and capacitors. The latter loses charge (i.e. the stored information) over time and needs to be refreshed periodically. Nonetheless, due to their central position in the semiconductor industry and ease of accessibility, they are also considered to be used with NN systems, especially when their efficiency can be increased when applying the trade-offs of approximate memory.

SRAM: For BNN inference systems using on-chip SRAM, the studies in the literature mainly employ scaling of various device parameters. To reduce energy consumption, the SRAM voltage is scaled in [Sun+17; HLPS20]. In [Yan+18], the weight and activation memories for BNNs are separately tuned to achieve fine-grained control over the energy consumption. In [DSM20] a split SRAM is proposed for BNNs, with an efficient unreliable memory region and a reliable region.

DRAM: For DRAM, the study in [Kop+19] provides an overview over studies related to NNs that use different DRAM technologies and proposes a framework to evaluate NN accuracy for using approximate DRAM in various different settings and inference systems. Specifically, they show that DRAM parameters can be tuned such that energy and performance are optimized to achieve significant improvements, whereas the NN accuracy drop stays negligible due to the NNs' adaptations in retraining.

2.3.2 *BNN Acceleration*

All BNN hardware systems use some form of processing components to perform the BNN workloads. In the case of CPUs [Hu+18; Jaf+18; Zha+18] and GPUs [Che+20a; LS21] for deploying BNNs, traditional off-the-shelf general purpose processors are used, while in the case of FPGAs and ASICs, custom hardware for acceleration is employed. FPGAs and ASICs provide higher efficiency in energy, area, while providing lower latency than CPUs/GPUs [Nur+16]. In Fig. 2.3, we show a comparison

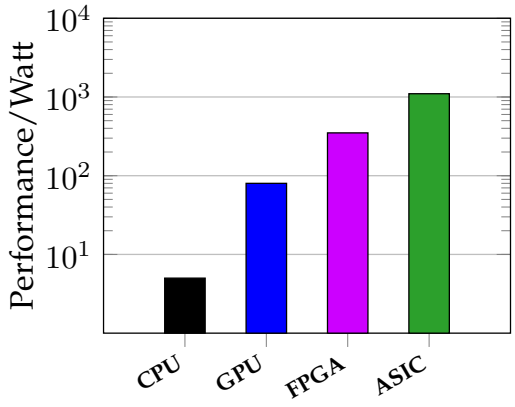


Figure 2.3: Comparison of BNN execution on different hardware: CPU, GPU, FPGA, ASIC. The data is based on the study in [Nur+16].

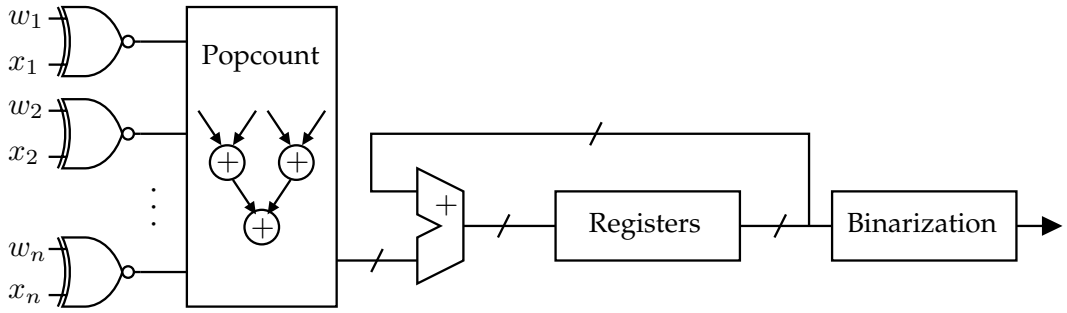


Figure 2.4: Overview of a BNN computing unit.

of BNN processing of different platforms, i.e. CPU, GPU, FPGA, and ASIC. It shows the performance per Watt relative to a baseline software implementation for CPUs, which is not optimized. The data for the figure is based on the study in [Nur+16]. We focus on ASIC acceleration in the following.

The high-level overview of a BNN accelerator computing unit is shown in Fig. 2.4. The design is inspired by the study in [Nur+16]. The binary inputs and weights, which are in form of bitstrings of length n, are loaded into the XNOR gates. The XNOR gates (representing the binary multiplication) return the result of the XNOR operations as a bitstring of length n as well. Then, the popcount unit counts the number of bits that are "1". Subsequently, the result of the popcount unit is accumulated in the registers. The binarizer returns binary value once all accumulations are completed.

For high throughput, multiple computing units of the form in Fig. 2.4 can be used in parallel. Such accelerators are organized with m computing units and n XNOR gates per computing unit, i.e., they have size $(m \times n)$, which determines the workload they can process. Accelerators of size $(m \times n)$ can further be embedded into a higher

hierarchy, i.e., multiple accelerators of size $(m \times n)$ on the same chip, or any other configuration can be used.

Different types of data flows can be used to feed the hardware with inputs and weights in Fig. 2.4. The two types of data flow methods are (1) output stationary (OS) and (2) weight stationary (WS). OS and WS are defined for general NNs in [CES16] and they can be used efficiently for BNN accelerators [Che+18]. In OS, new input activations and weights are streamed in each cycle, which necessitates only one accumulation register per computing unit. In WS, the weights are programmed into the XNOR gates once and reused as much as possible for multiple input activations, such that the number of new weight writes to the XNOR gates is minimized. This however necessitates a large amount of registers for storing intermediate results. The benefit of OS is a lower area footprint than in WS, as the partial sums are not intermediary stored but directly accumulated. The benefit of WS is the high reuse of the weights, requiring significantly less weight movements and smaller number of rewrites to the XNOR gates compared to OS. The study in [Che+18] chooses to use WS for FeFET-based XNOR gates, as writing the new weights to XNOR gates is costly and is worth paying the costs of registers for storing intermediate results.

To accelerate the operations of BNNs, classical digital circuits or the emerging analog computing paradigm can be used. In the following, we focus on BNN acceleration in the digital domain in Sec. 2.3.2.1 and in the analog domain in Sec. 2.3.2.2.

2.3.2.1 *Digital Computing Based BNN Accelerators*

Note that, to build digital hardware accelerators, they are first conceptualized then designed by creating the description of the HW and its behavior in a hardware description language (HDL), such as VHDL or Verilog. The HDL is then synthesized and evaluated in Electronic Design Automation (EDA) tools. Manual modifications on critical parts of the hardware on the circuit or physical level are sometimes required to meet constraints. The final HW designs will always be in some form of HDL or even lower level descriptions, and could be engineered or generated in different ways. More information on the steps of the hardware design process are in [WH10; HH15]

Many recent studies implement digital computing based BNNs on FPGAs and ASICs. We summarize a few examples in the following. **FPGAs**: In [Zha+17], BNN accelerators are generated by compiling C++ code Verilog with the use of High-Level Synthesis (HLS). The study [Lia+18] proposes an analytical resource aware model analysis (RAMA) for estimating the cost of a BNN accelerator design and demonstrate that using their model, highly efficient BNN accelerators can be built. In [Zha+23a] a BNN accelerator with fractional activations is used, where the information given by the binary activations are enhanced with additional binary operations, leading to high accuracy at low resource cost on FPGAs. **ASICs**: It is well-known that ASICs are significantly more resource efficient than FPGAs. This is unsurprisingly also the case for BNN hardware [Nur+16]. Therefore, specialized BNN accelerators have been designed, fabricated, and evaluated. We give a few notable examples here.

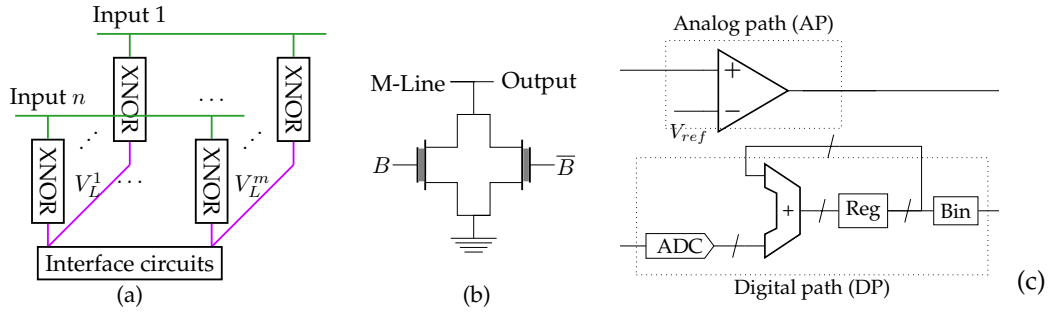


Figure 2.5: (a): Crossbar with interface circuit. A possible realization of an interface circuit is shown in (c). The voltages $V_L^1, \ldots, V_L^m$ or the currents are passed to the interface circuits. (b): Realization of an XNOR gate from FeFET transistors. (c) Interface circuit with an analog path (AP) and a digital path (DP) in [Che+18]. Reg: Registers, Bin: Digital comparator.

The first BNN ASIC accelerator has been proposed by [And+18b]. In the study it is shown how efficient the hardware performes compared to a 12-bit NN accelerator. A combinational design of BNN accelerator for low-power near-sensor processing has been designed in [RCB18], which features high energy efficiency. The study proposes an accelerator architecture for binary/ternary neural networks where the computation is performed without the need to access any external or off-chip memory [And+18a]. In [DSS20], an accelerator is proposed that uses aggressive, error-inducing voltage reduction with components to correct the occurring errors. Recently, there have also been highly specialized designs, i.e. a BNN stereo vision accelerator [Zha+23a].

2.3.2.2 *Analog Computing Based BNN Accelerators*

Note that, to design an analog circuit, a schematic needs to be drawn. Circuit simulation and verification is performed with a SPICE tool. Then many low-level steps need to be taken to realize the schematic on a chip. Special attention needs to be paid to the noise and transients, as they may impair the circuit operation. More information on the hardware design of analog circuits is in [WH10].

The operations of NNs can also be performed in the analog domain by using Ohm's and Kirchhoff's laws. It has been shown that this offers high efficiency in many metrics compared to digital systems [Chi+16; Sha+16a]. However, analog computing, especially with multi-bit NNs, have severe disadvantages. The analog states suffer from variation due to a combination of several factors such as process variation, IR drop, and leakage current [Sch+17; Che+20b; Kim+19; MN+21], which are challenging to overcome. Specifically, one the most critical issues are the imprecisions due to the aforementioned factors, therefore it is difficult to precisely differentiate between different analog states (e.g. of a current), which often have limited noise margins between them. Furthermore, the more states that are needed to be represented given a certain signal range, the more the requirement on analog computing becomes

infeasible. Consider as an example the multiplication of two four bit numbers. For each four-bit number, 2^4 different analog states are needed to represent them. When they are multiplied, the product may need up to 8 bits, which means 2^4 additional states need to be represented in the analog domain. Additonally, the results of multiple products may need to be accumulated to complete a MAC operation, potentially necessitating more analog states.

Fortunately, BNNs have exceptional synergy with analog computing. Due to the binarizations, BNNs require a much smaller number of states that need to be differentiated compared to higher-precision NNs. Since in BNNs, the result of a product is a single bit, computing the complete MAC operation using BNNs merely needs as many analog states as there are XNOR operations. For example, when using the accelerator in Fig. 2.5(a), only $n + 1$ (number of XNOR gates and one additional state for the case that all XNOR gates output "1") analog states are needed to represent the popcount result.

Furthermore, to give an example of how analog computing can improve the efficiency of computing units, consider the case where the digital computing based popcount unit is replaced with an analog computing based component. For computing the BNN workload in the digital domain, typically CMOS-based XNOR gates, popcount units, and other digital components are used. The operations of the accelerator in Fig. 2.4 can also be performed in the analog domain. Especially the popcount unit uses a large amount of resources (e.g. in [Umu+17], the performance of BNN acceleration HW is measured by XNOR–popcount operations per second). An implementation of a popcount unit for n XNOR gates needs $\frac{n}{2}$ two-bit-input adders in the first adder tree level (counting from the top). In the second level, $\frac{n}{4}$ three-bit-input adders are needed, and so on. This means, for n XNOR gates, there are in total $\log_2(n)$ levels, and in total $\sum_{i=0}^{\log_2(n)} 2^i$ adders are necessary, where with each additional level, the number of inputs and outputs in each adder gets increased by one. Furthermore, between each level of adders, pipeline registers are necessary as a buffer for the intermediate result of each level. As many registers are needed as there are adders, and the number of bits of each register is determined by the number of output pins of the adders. Note that in each column in the crossbar array, a popcount unit is required, unless the popcount units are shared, which in turn costs latency.

In analog computing based BNN accelerators, this popcount unit is completely removed. The currents that come out of the XNOR gates are just summed by employing Kirchhoff's current law, merely using the available wires that pass the current. However, note that the summed current in the analog domain needs to be converted again into the digital domain using ADCs. Depending on the used technology and implementation of the ADC and popcount unit, the resource usage of an ADC may be close to the popcount unit. In fact, in analog computing based NN accelerators, are the components that use the majority of the energy and chip area [Sha+16b]. Therefore, care should be taken when using analog-based computing and selecting ADCs, so that the analog-based circuits do not require more resources compared to digital circuits.

A high-level overview of an analog computing based crossbar for BNNs is shown in Fig. 2.5(a). The input bits (1 to n) are applied to the input lines (in green) to the XNOR gates. The XNOR gates are programmed to store the binary weights (the circuits for programming are omitted). The XNOR gates can be built from standard CMOS or other emerging technologies, such as FeFET, which promise highly efficient inference [Che+18; Sol+20b]. To give an example, FeFET-based XNOR gates are built by coupling two FeFET transistors together, as shown in Fig. 2.5(b). The binary weight is stored in a complementary manner in both transistors. Depending on the input signal B and its complement $\overline{B}$, the output will either be logic "1" or "0". In practice, the match line (M-Line) is charged to high, and when there is a match between the input and the stored value, both transistors will be off and no conducting path is formed, leading to a logic "1". Only when there is a mismatch, a conducting path is formed, which causes the voltage at the output line to drop, returning logic "0".

In the analog domain, the popcount can be computed using Kirchhoff's circuit law, i.e. by the summation of the output currents of all XNOR gate. Since in the analog domain, the popcount is computed by using Kirchhoff's circuit law (i.e. summation of the output currents of all XNOR gate), the resulting m currents (from m different computing units) are passed to the interface circuits for further processing.

The interface circuits can be built from analog comparators, ADCs, and also digital components, such as accumulators, registers, and digital comparators, see Fig. 2.5(c). Here it has two paths. The upper path (analog path) is used when the thresholding can be directly applied to the result of the popcount, i.e. when the β of the workload is smaller or equal to n. If β is larger, the lower path (digital path) is used.

Several recent studies have exploited the principles of analog computing for efficient BNN computations. In [Jeb+21], an analog computing based popcount unit is proposed, promising higher efficiency and low error probabilities compared to using Kirchhoff's circuit law in conjunction with ADCs. An SRAM-based analog computing scheme for BNNs is proposed in [Yin+20], requiring only a 3.64 bit ADC. In [Kim+19; Zha+20], BNN accelerator architectures are with RRAM and specialized methods for the compensation of the process variation, which leads to the distortion of analog computations. The study in [Sun+18] proposes methods to increase the parallelism in RRAM-based BNN accelerators. In [KLC18], a complete analog implementation of BNN neurons is built, also using RRAM as weight storage and by using switched capacitors to realize the binarized neuron operations. The studies in [Che+18; Sol+20b] propose to use FeFET-based analog computing hardware for BNNs. In [Che+18] it is shown how FeFET-based analog computing fares against RRAM, and the study in [Sol+20b] expands on that by reducing the number of ADCs by employing resource sharing. Both studies argue that FeFET-based BNN accelerators are significantly more efficient than ones based on RRAM.

3 EXPERIMENT SETUP

In this chapter we present the experiment setup for the evaluations of methods and proof-of-concepts in this thesis. In Sec. 3.1, we explain the details of the datasets. Then, in Sec. 3.2, we introduce our BNN models and how they are trained.

3.1 DATASETS

We use the following image classification datasets for evaluations. The information about the datasets are also summarized in Table 3.2

FashionMNIST: It contains 60000 training and 10000 testing images of the size 28×28 in grayscale format. The images are of clothes, categorized into 10 classes sold on Zalando. The dataset is also referred to as "Fashion" for short. More information about the dataset can be found in [Fas].

SVHN: It contains 73257 training and 26032 testing 32×32 images with 3 color channels. The images depict house numbers in 10 classes from Google Street View. More information about the dataset can be found in [Svh].

CIFAR10: It includes 50000 training and 10000 testing images of size 32×32 with 3 color channels. The images are categorized into 10 classes of common objects and animals. More information about the dataset can be found in [Cif].

Imagenette: It includes 9470 training and 3925 testing images. The images are a subset of 10 classes from the original ImageNet dataset. The image sizes in the dataset vary. For the evaluations in this thesis, the images are scaled to 64×64 (and all images have 3 color channels). More information about the dataset can be found in [Ima].

3.2 BNN MODELS

The BNN models (Table 3.2) used in this thesis are modified and binarized (weights and inputs) based on the architectures of VGG and ResNet [SZ14; He+16], adapted for the above datasets, except the fully connected model, which is a standard two-hidden layer fully connected (FC) NN. We always use binarization-aware training, i.e. we simulate the binarization during the training process. We use moderately difficult prediction tasks, with models referred to FC, VGG3, VGG7, and ResNet18. The BNNs are up-to-date, suitably sized, not overparametrized, and capable models tailored for resource-constrained inference.

Name	# Train	# Test	# Dim	# classes
FashionMNIST (FC, VGG3)	60000	10000	(1,28,28)	10
KuzushijiMNIST (VGG3)	60000	10000	(1,28,28)	10
SVHN (VGG7)	73257	26032	(3,32,32)	10
CIFAR10 (VGG7)	50000	10000	(3,32,32)	10
Imagenette (ResNet18)	9470	3925	(3,64,64)	10

Table 3.1: Datasets used for experiments.

The datasets used in this thesis are all multiclass classification tasks. To measure the performance of the BNNs regarding the train and test dataset, we use the metric referred to as accuracy. It is defined as

$$\text{Accuracy} = \frac{\text{correct classifications}}{\text{all classifications}}. \tag{3.1}$$

We report it as a percentage. To give an example, the FC BNN achieves around 89% accuracy on the test set of the Fashion dataset, meaning that 89% of its preditions are correct.

3.2.1 *BNN Layer Types*

In the following we describe the layer types that we use in our BNNs.

Fully-Connected Layer: The fully connected layer connects every neuron in the current layer with every neuron in the next layer. We write FC1024 when 1024 neurons are used to compute the output of the FC layer. FC layers have binary weights as learnable parameters.

Convolution layer: The convolutional layer computes a 2D convolution of the input with a specified number of filters, e.g. with 64 we write C64. C layers also have binary weights as parameters. In this work, we use filters of size 3×3 only.

Maxpooling: The maxpool layer downsamples the input by selecting the maximum value of the input in a given window size, with 2 we write MP2. MP has no trainable parameters. We use maxpool with window size 2×2 only in this work.

Batch Norm and Binary Activation: The batch normalization (BN) layer is used for faster and more stable training. In this work, it is always followed by the binary activation function (except for ResNets). For inference, the BN layer followed by activation can be computed by binary thresholding [SBN19]. Therefore, in BNN inference, BN layers are fused with the binary activation layer and have thresholds which are signed integers.

Skip Connection: Skip connections (or residual connections) connect the output of one layer to the input of another layer, whereas layers may be skipped. By doing this, the information from earlier layers can bypass any number of layers and can directly influence other subsequent layers. The main advantage is that it helps during

Name	Architecture
FC	In → FC2048 → FC2048 → FC10
VGG3	In → C64 → MP2 → C64 → MP2 → FC2048 → FC10
VGG7	In → C128 → C128 → MP2 → C256 → C256 → MP2 → C512 → C512 → MP2 → FC1024 → FC10
ResNet18	In → C64 → SCB64 → SCB128 → SCB256 → MP2 → SCB512 → MP4 → FC10

Table 3.2: BNNs with fully connected (FC), convolutional (C), and maxpool (MP) layers. SCB: Skip-connection block. Convolutional layers are followed by batch normalization layers, except output layers.

training, as it alleviates the vanishing gradient problem, which leads to faster convergence [He+16]. In this work, the ResNet18 architecture is equipped with a few skip connections.

3.2.2 *Training BNNs*

We train the BNNs in the configurations shown in Table 3.2 for the above mentioned image classification datasets shown in Table 3.1. For training, we develop a framework based on the PyTorch library. PyTorch does not support the BNN operations XNOR and popcount operations officially. Therefore, to simulate the usage of BNNs, we binarize the weights and activations to $\{-1, +1\}$. This way, we can still use the official MAC-libraries in PyTorch for BNN computations.

The standard loss used for multi-class classification is the cross entropy loss (CEL), which is defined as follows:

$$\mathcal{L}_{CEL}(\hat{y}, i) = -\log\left(\frac{\exp(\hat{y}_i)}{\sum_j \exp(\hat{y}_j)}\right) = \log\sum_j \exp(\hat{y}_j) - \hat{y}_i. \tag{3.2}$$

The ground truth class is i and $\hat{y}_j$ is the prediction of the BNN. We use the CEL in all experiments unless specified otherwise.

Also, unless specified otherwise, in all experiments, we run the Adam optimizer for 100 epochs for FashionMNIST and 200 epochs for CIFAR10 and Imagnette to minimize the cross entropy loss or using a different loss if specified otherwise. We use the batch sizes of 256 for Fashion, Kuzujishi, SVHN, CIFAR10, and 128 for Imagenette. In all cases we use an and an initial learning rate of 10^{-3}. To stabilize training, we exponentially decrease the learning rate every 10 or 25 epochs by 50 percent for Fashion, and every 50th epoch for CIFAR10 and Imagenette.

3.2.3 *Experiment Platform*

To run the experiments of this dissertation, we use a server with an Intel Core i7-8700K 3.70 Ghz CPU, 32 GB main memory, and two GeForce GTX 1080 8 GB GPUs. To reduce the latency of the binarization, we employ custom CUDA kernel extensions for PyTorch. For bit flip injection (e.g. in Ch. 4) and simulating approximate computation schemes (e.g. in Ch. 6), we also develop custom CUDA kernels, which, instead of the PyTorch MAC engine, use a MAC engines developed in the scope of the dissertation work.

4

ERROR TOLERANCE OPTIMIZATION OF BINARIZED NEURAL NETWORKS

To enable the exploration of this dissertation's vision, our first goal is to optimize BNNs for higher error tolerance, so that it can be exploited as much as possible with approximate memory and computing units.

The classical method to achieve bit error tolerance in BNNs and NNs is training with bit flip injections according to the error model. Bit flip injection during training, however, has disadvantages. First, recent studies have reported that injecting bit flips during training can significantly degrade accuracy. The higher the bit error rate during training, the more significant the accuracy degradation [Hir+19b; Kop+19; Bus+20]. Another disadvantage is the additional overhead [Mra+19]. During the training with bit flip injection, for every bit of the error-prone data, a decision has to be made whether to inject a bit flip. This adds numerous additional steps in the NN training.

Achieving bit error tolerance in NNs without bit flip injection, and thus, conquering the above disadvantages, would be a breakthrough for the research area of NNs using approximate computing and memory. To achieve this, the principles of bit error tolerance in NNs need to be understood well. However, to the best of our knowledge, the underlying principles of NN bit error tolerance have not received much attention before the work of this dissertation.

The key focus of this chapter is to explore methods to achieve bit error tolerance without bit flip injection in BNNs. Due to the binarization, BNNs have simple structures, due to which the error tolerance can be analyzed easier than in the case of multi-bit NNs. We analyze the bit error tolerance of BNNs and propose margin-based metrics that measure the bit error tolerance of their structural elements. Then we use the metrics for bit error tolerance optimization by transforming them to a margin maximization problem. This allows us to adopt existing methods from support vector machines (SVMs) to solve the bit error tolerance problem.

In this chapter, we first provide a margin-based bit error tolerance metric for single hidden-layer neurons in BNNs, which formally characterizes when a neuron flips its output value. We further propagate this metric to the output layer of the BNN to quantify the bit error tolerance of the output layer, which is used to quantify the ultimate impact of bit flips on the inference accuracy. Based on the margin-based output layer metric and the well-known hinge loss for maximum margin classification in SVMs, we propose a modified hinge loss (MHL) for bit error tolerance optimization of BNNs, which works without bit flip injections during training. To evaluate our methods, we perform extensive experiments to compare with the-state-of-the-art approaches, which train BNNs with cross entropy loss (CEL) and bit flip injections. The results show that applying the MHL alone (without any bit flip injections)

outperforms CEL in terms of accuracy. We further evaluate the combination of MHL and bit flip injections, which significantly improves the accuracy of BNNs at high bit error rates.

The remainder of this chapter is organized as follows. In Sec. 4.1, we define the problem of bit error tolerance in BNNs. In Sec. 4.2, we define bit error tolerance metrics on the hidden-layer-neuron and on the output-layer level. In Sec. 4.3, we construct the MHL based on the metrics to optimize for bit error tolerance. In Sec. 4.4, we present our experiment results and compare the MHL to classical methods, i.e. CEL for achieving bit error tolerance in BNNs.

4.1 PROBLEM DEFINITION

Given a set of labeled input data, the objective is to train a BNN for high accuracy and high bit error tolerance. We assume that the bit errors are transient, i.e. bit flips are injected only when reading the error-prone data and the bit flips are discarded afterwards, meaning the next time bit flips are injected, the correct data is used. In this chapter, we focus on the problem of *how to train BNNs for bit error tolerance without bit flip injections*.

To solve this problem, we explore bit error tolerance metrics, which allow us to describe how the bit error tolerance of BNNs can be conceptualized with margins. Then, we modify the hinge loss known from margin-maximization in SVMs to make it applicable to BNNs.

4.2 BIT ERROR TOLERANCE METRICS

In this section, we first introduce a margin-based neuron-level bit error tolerance metric for BNNs in Sec. 4.2.1, which we then extended to formulate a bit error tolerance metric for the output layer in Sec. 4.2.2.

4.2.1 *Neuron-Level Bit Error Tolerance*

In the following, we use a notation describing properties of neurons in fully connected layers, but our considerations also apply to neurons in convolutional layers. Let j be the index of one neuron in a BNN, and $x \in \mathbf{X}$ an input to the BNN. Let $s_{x,j} \in \mathbb{Z}$ be the pre-activation value of neuron j before applying the activation function in fully connected layers. In convolutional layers, we would use $s_{x,j,u,v} \in \mathbb{Z}$ at the location $(u, v) \in \{0, \dots, U\} \times \{0, \dots, V\}$ for a feature map with height U and width V, however, for simplicity of presentation we stick to the fully connected case. For BNNs, the pre-activation values of a neuron are computed by a weighted sum of inputs and weights (according to Eq. (2.1)) that are ± 1. Therefore, one bit flip in one weight changes the pre-activation value by 2.

Theorem 1. *Let $j \in \{0, \dots, J\}$ be the index of one neuron. Furthermore, let q be the number of bit flips induced into the weights of neuron j. The pre-activation $s_{x,j}$ of a neuron after induction of these bit flips is in the interval $[s_{x,j} - 2q, s_{x,j} + 2q]$.*

Proof. For better readability, we use s for $s_{x,j}$ for abbreviation in the proof. Since the weights are ± 1, each bit flip of one weight of neuron j changes s by 2. Inductively, this shows that q bit flips change s by up to $2q$. Hence, the pre-activation of neuron j after up to q bit flips is in $[s - 2q, s + 2q]$. □

We use this proof to first formulate a neuron-based bit error tolerance metric for hidden-layer neurons. We define the set of indices of neurons of the hidden layer by J_h. For hidden layer neurons, i.e., those with index $j \in J_h$, the pre-activation value is compared with a threshold $t_j \in \mathbb{Z}$, the comparison against which yields a binary output. Here, we consider the following activation function:

$$\theta_{t_j}(s_{x,j}): \quad s_{x,j} \mapsto \begin{cases} 1 & s_{x,j} > t_j \\ -1 & \text{else} \end{cases} \tag{4.1}$$

As long as the weights or input flips do not cause the pre-activation value to pass the threshold, the activation will not flip, in which case the neuron is bit error tolerant. Therefore, the bit error tolerance of an hidden layer neuron depends on the margin

$$M_{x,j} = |s_{x,j} - t_j| \tag{4.2}$$

between the pre-activation value and the threshold. With each bit flip $s_{x,j}$ may get closer to t_j and may finally flip the output of the activation if t_j is passed.

Corollary 1. *Let $j \in J_h$ be the index of one hidden layer neuron. If $s_{x,j} > t_j$, then $\max\left(0, \left\lfloor \frac{M_{x,j}}{2} \right\rfloor - 1\right)$ many bit flips can be tolerated. Else, $\left\lfloor \frac{M_{x,j}}{2} \right\rfloor$ can be tolerated.*

Proof. We denote s for $s_{x,j}$, t for t_j, and M for $M_{x,j}$. We analyze the two cases individually.

1) If $s - t > 0$, then the output of neuron j is $+1$. We denote by $\tilde{s}$ the value of s after up to $q := \max\left(0, \left\lfloor \frac{M_{x,j}}{2} \right\rfloor - 1\right)$ bit flips. By theorem 1, we have $\tilde{s} \in [s - 2q, s + 2q]$. We conclude that $\tilde{s} - t \geq s - 2q - t = M - 2q$ which is > 0 for q defined as above. More specifically, the output of neuron j is still $+1$.

2) On the other hand, if $s - t \leq 0$, then the output of neuron j is -1. Let $\tilde{s}$ be the pre-activation value after up to $q := \left\lfloor \frac{M_{x,j}}{2} \right\rfloor$ bit flips. Again, using theorem 1 yields $\tilde{s} \in [s - 2q, s + 2q]$. We obtain $\tilde{s} - t \leq s + 2q - t = -M + 2q \leq 0$ and the output of neuron j is still -1. □

We demand that the neuron has a bit error tolerance of at least b bit flips. We define the *bit error tolerance* $M^b_{x,j}$ of a neuron j given the input x as

$$M^b_{x,j} = \mathbf{1}\{M_{x,j} \geq b\}, \tag{4.3}$$

where **1** is the indicator function returning a "1" in case the condition is true and "0" otherwise. For the case of convolutional layers we consider that a neuron has to be robust in all convolution windows, i.e.

$$M_{x,j}^{b,conv} = \frac{1}{UV} \sum_{u=1}^{U} \sum_{v=1}^{V} \mathbf{1}\{M_{x,j,u,v} \geq b\}. \tag{4.4}$$

Again, for simplicity, we stick to the notation for the fully connected case. We then define the bit error tolerance of the entire BNN as the average error tolerance across all neurons J^{total}:

$$M_x^b = \frac{1}{J^{total}} \sum_{j=1}^{J^{total}} M_{x,j}^b \tag{4.5}$$

We determine M_x^b by evaluating the BNN on the entire dataset, where $|\mathbf{X}|$ is the number of data points in the data set $\mathbf{X}$:

$$M^b = \frac{1}{|\mathbf{X}|} \sum_{i=1}^{|\mathbf{X}|} M_x^b \tag{4.6}$$

For a set of values $b \in \{b_1, \ldots, b_B\}$ we consider M to be the tuple $M = (M^{b_1}, \ldots, M^{b_B})$.

The definition of the metric M^b in Eq. (4.6) allows optimization with respect to it. Since M^b is a count and not a differentiable function, we construct a regularizer that punishes neurons that do not fulfill a bit error tolerance of at least b. To this end, we rely on the well-known hinge function to build a convex and sub-differentiable regularizer. For a given bit error tolerance tolerance with b, we regularize each neuron j for each input example x using the hinge-function, i.e.

$$\text{Reg}_j^b(x) = \max(0, b - M_{x,j}). \tag{4.7}$$

Here, Reg reaches its minimal value when a neuron has $M_{x,j}$ of at least b. To optimize the entire BNN for bit error tolerance using Eq. (4.7), we compute the mean of all neuron regularizers. We weight the regularizer with $\lambda > 0$ and add it to the loss for minimization.

As we will see in the experiments in Sec. 4.4.2, optimizing with respect to M^b unfortunately does not increase the bit error tolerance. Note that bit flips may propagate through the hidden layers of BNNs. If the margins from Eq. (4.2) are small, weight flips might cause the neuron j to flip its output. This affects subsequent neurons for which neuron j provides inputs. A flip of the input value for a neuron affects the pre-activation value exactly as the bit error of a weight, i.e., it modifies the pre-activation value by 2. Therefore, the subsequent neurons may have to tolerate a higher number of bit errors.

A detailed analysis of the neuron-based bit error tolerance metric has been conducted in [Bus+20], showing the relation of the metric in Eq. (4.6) to the bit error

tolerance of BNN. Using this metric for optimizing bit error tolerance has been reported to be unsuccessful. We provide a summary of the analysis by presenting the negative result regarding the neuron-level error tolerance in Sec. 4.4.2. In short, based on the evaluations, we speculate that the reason for the negative result is that bit flips of neuron outputs can only affect the BNN prediction if the effect of bit flips reach the output layer and lead to a change of the predicted class. Therefore, we now shift our focus on applying the notion of margin to the output layer.

4.2.2 *Output-Layer Bit Error Tolerance*

Each neuron in the output layer has only one output value $s_{x,j}$ (for an input x, where j is the neuron index) which is one entry in the vector of all neuron predictions $\hat{y}$. There are as many values in $\hat{y}$ as there are neurons in the last layer. J^{output} is the number of output neurons in the BNN. No activation function is applied to the output value of the last-layer neurons. The index of the entry with the maximum value in $\hat{y}$ determines the class prediction, where we assume that ties are broken arbitrarily.

If bit errors lead to modifications of the output values in the output layer such that another neuron provides the highest output value, then the class prediction changes. Let $s_{x,j'}$ and $s_{x,j''}$ with j', j'' be the highest and the second highest output value of neurons in the output layer. The following corollary shows that the margin

$$m = m_{x,j'} - m_{x,j''} \tag{4.8}$$

serves as bit error tolerance metric for the output layer. We use m with a small letter to differentiate the margin for the output layer.

Corollary 2. *If $m > 0$, then the output layer of the BNN tolerates* $\max(0, \lfloor \frac{m}{2} \rfloor - 1)$ *bit flips.*

Proof. Let $q \in \{0, \ldots, \max(0, \lfloor \frac{m}{2} \rfloor - 1)\}$ be a number of bit flips. We consider any distribution of the q bit flips to weights or inputs of the output layer, i.e., $\sum_{j \in J^{output}} q_j = q$ where q_j is the number of bit flips in weights or inputs of the neuron j.

Let j' be the index of the neuron with the highest output value. Furthermore, let $j \neq j' \in J^{output}$. For better readability, we denote $s_{j'}$ for $s_{x,j'}$ and s_j for $s_{x,j}$. Furthermore, we denote by $\tilde{s}_{j'}$ and $\tilde{s}_j$ the values of $s_{j'}$ and s_j after $q_{j'}$ and q_j bit flips. By theorem 1, we know that $\tilde{s}_{j'} \in [s_{j'} - 2q_{j'}, s_{j'} + 2q_{j'}]$ and $\tilde{s}_j \in [s_j - 2q_j, s_j + 2q_j]$. We conclude

$$\begin{aligned} \tilde{s}_{j'} - \tilde{s}_j &\geq s_{j'} - 2q_{j'} - s_j - 2q_j \\ &\geq s_{j'} - s_{j''} - 2q = m - 2q > 0, \end{aligned}$$

as $q \leq \max(0, \lfloor \frac{m}{2} \rfloor - 1) < \frac{m}{2}$. Since $j \neq j' \in J^{output}$ is chosen arbitrarily and we have shown that $\tilde{s}_{j'} > \tilde{s}_j$, the output value of neuron j' is still maximal even after q bit flips in the output layer. □

4.3 MARGIN-MAXIMIZATION FOR BIT ERROR TOLERANCE OPTIMIZATION

In this section, we describe how we use the the margin-based bit error tolerance metric of the output layer and the well-known hinge loss for maximum margin classification to construct the modified hinge loss for optimizing the bit error tolerance of BNNs.

For bit error tolerance of the last layer, the margin m as introduced in Eq. (4.8) needs to be large, so that the maximum number of bit flips the output layer can tolerate is high. The margin can be directly computed by subtracting the second highest entry $\hat{y}_{c''}$ of the output vector $\hat{y}$ from the highest entry $\hat{y}_{c'}$, i.e., $m = \hat{y}_{c'} - \hat{y}_{c''}$. However, optimizing with respect to m without considering the other entries $\hat{y}_c$ of $\hat{y}$ may not exhaust the full potential of the margin between $\hat{y}_{c'}$ and the output of the other classes $\hat{y}_c$. The larger the margin between $\hat{y}_{c'}$ and $\hat{y}_c$ of other classes c, i.e. $m_c = \hat{y}_{c'} - \hat{y}_c$, the more bit errors can be tolerated in the neuron that calculates $\hat{y}_c$ and the other neurons so that the prediction does not change. To put it concisely, for a bit error tolerant output layer, $\hat{y}_{c'}$ needs to be as large as possible, while the other $\hat{y}_c$ need to be as small as possible. To achieve this, we build upon the hinge loss for maximum margin classification.

The hinge loss, as described in [Ros+03], for maximum margin classification is defined as

$$\mathcal{L}_{HL}(y, f) = \max(0, 1 - y \cdot f), \tag{4.9}$$

with the ground truth prediction $y = \pm 1$ and the prediction $f \in \mathbb{R}$. This loss becomes small if the predictions have the same sign as the predicted class and are close to 1 in magnitude. For predicted values larger than 1, the loss becomes 0. The "1" in the loss forces the classifier to maximize the margin between two class predictions. To solve optimization problems that use the hinge loss, many of the common optimizers or algorithms can be used, such as the stochastic gradient descent (SGD) strategy [Zha04].

In the case of BNNs for multi-class problems, the version of the hinge loss in Eq. (4.9) cannot be directly used. To extend the hinge loss to multiple classes, we define y_{enc} as a one-hot vector with elements in $\{-1, 1\}$, which has a $+1$ at the index with the ground truth, else -1. y_{enc} has the same number of elements as $\hat{y}$. Then the element-wise product $y_{enc} \cdot \hat{y}$ is computed. In this product, in case of correct predictions, positive predictions in the correct class will stay positive, negative predictions that should be as negative as possible become positive. In case of wrong predictions, i.e. high negative value for the correct class and high positive value for the wrong class, the values become negative. For a high penalty in the wrong case and a small penalty for the correct case, we subtract the product $y_{enc} \cdot \hat{y}$ from a parameter b, and get $(b - y_{enc} \cdot \hat{y})$. Since we do not demand higher prediction values than b, we set negative values to zero with the max function, and denote the modified hinge loss (MHL):

$$\mathcal{L}_{MHL}(\hat{y}, y_{enc}) = max\{0, (b - y_{enc} \cdot \hat{y})\}. \tag{4.10}$$

Eq. (4.10) is still a convex function like Eq. (4.9), so it can be used with the same optimizers. Here, for optimizing BNNs, the $\mathcal{L}$ in Eq. (2.2) is replaced with $\mathcal{L}_{MHL}$,

which optimizes the BNN via the mini-batch SGD strategy to minimize the difference $(b - y_{enc} \cdot \hat{y})$, as described in Sec. 2.2.2. The lower this difference, the larger the margin between $\hat{y}_{c'}$ and all the other $\hat{y}_c$. Above, we demanded exactly this property for a bit error tolerant output layer.

4.4 EXPERIMENTS

This section presents the results demonstrating the performance of BNNs optimized using the modified hinge loss (MHL) in comparison to the state-of-the-art cross entropy loss (CEL) [Hir+19b]. The experiment setup is presented in Sec. 4.4.1. We report the performance of the MHL without and with bit flip injection in Sec. 4.4.3 and Sec. 4.4.4, respectively.

4.4.1 *Experiment Setup*

We evaluate the three types of BNNs introduced in Ch. 3, namely a fully connected (FC) BNN and a small convolutional BNN (VGG3), both with the Fashion dataset, and a larger BNN with CIFAR10 (VGG7). The BNN architectures used are presented in Table 3.2 and the datasets are specified in Table 3.1. For training, we run the Adam optimizer [Hub+16] for 200 epochs for Fashion and 500 epochs for CIFAR10, with either cross entropy loss (CEL) or modified hinge loss (MHL).

To cover a wide spectrum of bit errors, for testing we use bit error rates (BERs) from 0% (no bit errors) up to 35%, with increments of either 1% for Fashion and 0.5% for CIFAR10. For training with bit flips we use different BERs, from 1% up to 30% BER, such that accuracy degradation is below 10% from the original accuracy. The bit flips are transient and symmetric. Depending on the approximate memory or computing units and their properties, accepting BERs of this extent can improve the key metrics such as energy consumption, timing parameters, production cost, and others.

As the base line to the MHL, the CEL with bit flip injections is used (proposed in [Hir+19b] and in [Kop+19]). The CEL measures the performance of classification NNs returning values that can be interpreted as class probabilities. With the ground truth class i and softmax as input, the CEL is defined in Eq. (3.2). With large differences among $\hat{y}_j$, the term $\sum_j \exp(\hat{y}_j)$ becomes approximately the highest value $\hat{y}_j*$ due to the exponential, so the term $\hat{y}_j* - \hat{y}_i$ influences the optimization most. In case of good predictions, this term will be close to zero. In case of bad predictions, i.e. large $\hat{y}_j*$ and small $\hat{y}_i$, the $\hat{y}_j*$ will be decreased and $\hat{y}_i$ increased. In these two cases, the margins between the neuron that returns $\hat{y}_i$ and neurons other than the one returning $\hat{y}_j*$ may not always be considered in the loss, because of the distortion by the exponential. However, the actual margins are considered in the MHL, which we compare to the CEL regarding bit error tolerance next.

4.4.2 *Neuron level metric*

In this subsection, our goal is to find out whether there is experimental support for the neuron-level bit error tolerance metric presented Sec. 4.2.1 and shown in Eq. (4.6). We plot the accuracy over BER for BNNs trained *without bit flip injection* (No flips) and *with bit flip injection* (Flip, p) using flip regularization, in the top row of Fig. 4.1. In the bottom row, we plot the results for the neuron-level bit error tolerance metric M^b, with $b = \{2, 4, 8, 16, 32, 64\}$.

For all cases tested, we observe that the accuracy over different BERs correlates with M^b. Note that when a BNN achieves high bit error tolerance, we also observe high M^b. Note that the difference between the M^b curves is higher for Fashion than for CIFAR10. Furthermore, the higher the BERs during training, the higher the values of the M^b curve.

Fig. 4.2 depicts the results for the direct regularization training shown in Eq. (4.7). We observe that this training method does not increase the accuracy over error rate, although the M^b values are high. Instead regularizing the training objective this way decreases accuracy at any BER. Similar curve progressions can be observed for other hyperparameter settings and the CIFAR10 dataset. For smaller regularization scalings λ, the observed curves approach the unregularized curves, however we never obtain higher accuracies at any error rate.

We conclude that the direct regularization method in the form presented Eq. (4.7) optimizing M^b is not usable for bit error tolerance optimization. While it effectively increases M^b, it does so by sacrificing accuracy thereby rendering the resulting models useless.

4.4.3 *MHL Only vs. FR*

Fig. 4.3 presents the experimental results of different BNNs with respect to the accuracy over BER, from 0% to up to 15% for the Fashion dataset, and from 0% to up to 5% for the CIFAR10 dataset. For each data set, the experiments were conducted with five BNNs different training runs using the MHL without any bit flip injections and CEL with different BERs for bit flip injections. For all BNNs trained with MHL, we employed a parameter search for b, which is shown in Eq. (4.10), testing powers of two, up to two times of the maximum value the neurons in the output layer can compute. The reason for the maximum value chosen is that the maximum output value of a neuron in the output layer is the number of neurons in the layer before the output layer. Among these configurations of b, the best one was chosen.

We observe that BNNs trained with the MHL without bit flip injections have better accuracy over BER than the BNNs trained with CEL under bit flip injections, i.e., in Fig. 4.3 for Fashion FC and CNN up to 10% and for CIFAR10 up to 5%. The BNNs trained with CEL suffer from significant accuracy drop for lower BERs, when the BER during training is high, e.g., CEL 20% and/or CEL 30% in Fig. 4.3 at low BER. The BNNs trained with MHL, however, do not suffer from this accuracy drop.

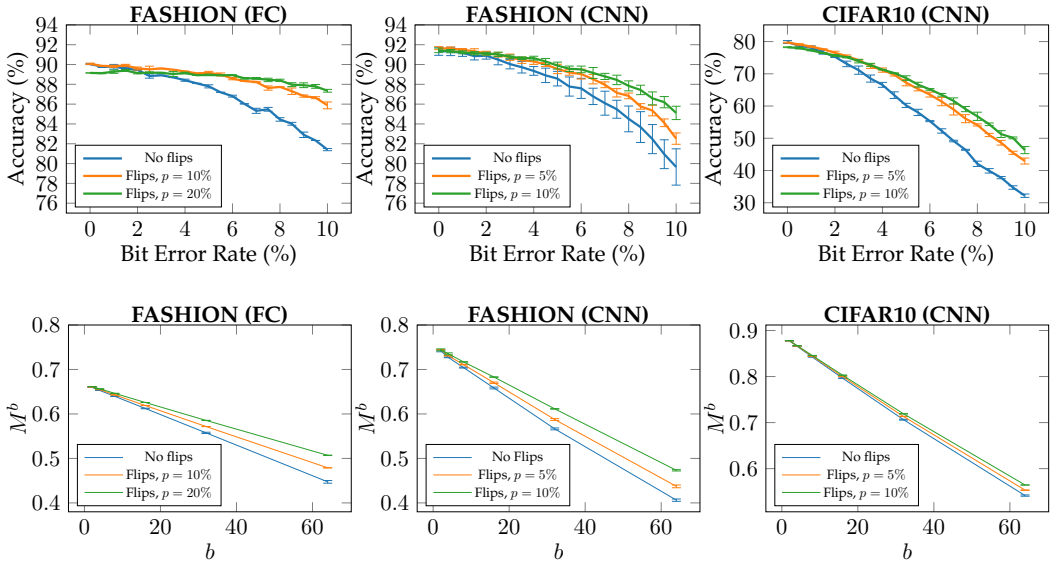


Figure 4.1: The relationship between accuracy over BER and M^b with $b = \{2, 4, 8, 16, 32, 64\}$. Top row: accuracy over BER, bottom row: M^b values plotted over b. FC means fully connected BNN, CNN means convolutional BNN.

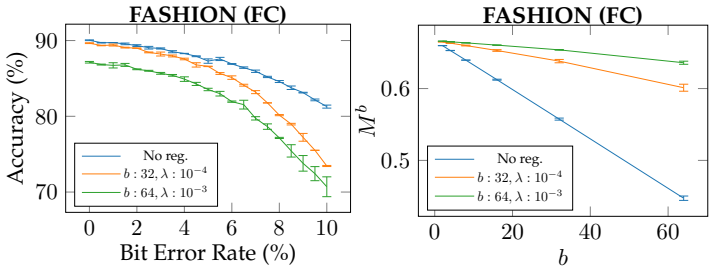


Figure 4.2: The experiment results for the direct regularization.

Although the BNNs trained with CEL 20% and bit flip injections have better accuracy for Fashion CNN in Fig. 4.3 when the error rate is higher than 10%, the accuracy of the BNNs drops by a significant amount, which may be unacceptable. Further investigations should be deployed, to be presented in Sec. 4.4.4.

4.4.4 *MHL Combined with FR*

In this section, we evaluate the BNNs trained with the MHL and bit flip injections under different BERs. In addition, the BNNs trained with the MHL without bit flip injections (i.e., those BNNs generated using the MHL in Sec. 4.4.3 under 0% BER) are

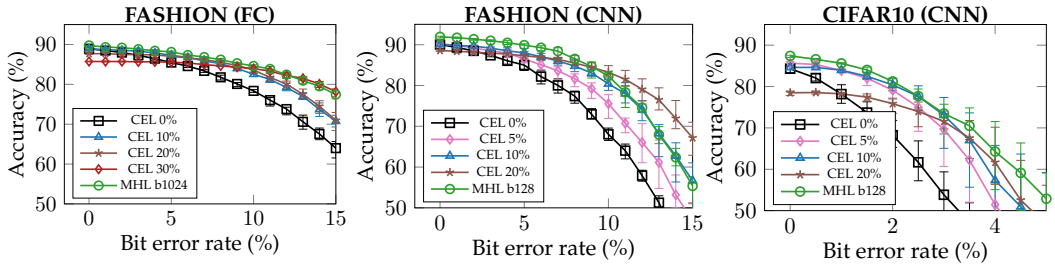


Figure 4.3: Accuracy over bit error rate for BNNs trained with CEL under a given bit flip injection rate (specified in the legend, 0%, 5%, 10%, etc.) and BNNs trained with MHL without bit flip injections for a specified b in Eq. (4.10).

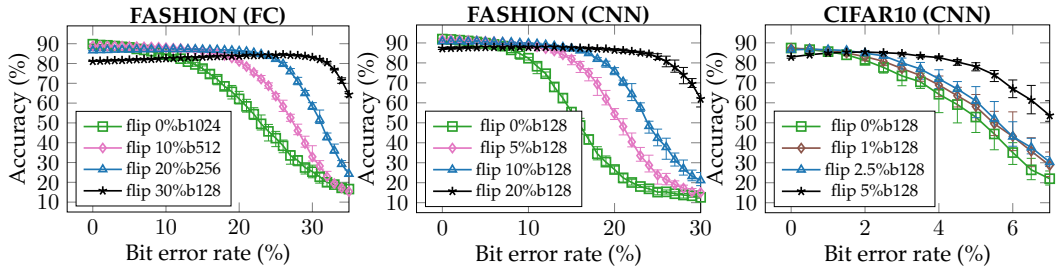


Figure 4.4: Accuracy over bit error rate for BNNs trained with MHL and bit flip injections (denoted as flip 0%, 1%, etc). The number after the b is the value to which the parameter b in the MHL is set during training (see Eq. (4.10)).

included here as the baseline in this subsection. For all configurations, we employed the same parameter search for b as in Sec. 4.4.3.

Fig. 4.4 presents the experimental results of different BNNs with respect to the accuracy over BER (from 0% to up to 30% for Fashion and from 0% to up to 6% for CIFAR10). In all experiments, we observe that the accuracy over the BER of the BNNs trained under MHL and bit flip injections is significantly higher than that of the baseline trained by only MHL. For example, for Fashion in Fig. 4.4, the BER at which the accuracy degrades significantly is extended from 5% (baseline, green curve) to 20% and 15% respectively, with a small tradeoff in the accuracy at 0% BER. If more accuracy at low error bit rates is traded, the BER at which accuracy degrades steeply can be shifted even further. For CIFAR10 in Fig. 4.4, this breaking point can also be increased. However, more accuracy has to be traded compared to the previous cases.

If b is higher than the ones shown, the accuracy for lower BERs suffers similar to using CEL with high BERs. If b is lower, there will be no significant change compared to CEL with 0% BER. We only show the results with the best b.

4.5 CONCLUSION

In this chapter, we proposed a concept of margin to formulate bit error tolerance metrics for the entire output layer. We formally proved that the metric measures the maximum number of any bit flips that can be tolerated. Based on this metric and the well-known hinge loss for maximum margin classification in SVMs, we proposed the modified hinge loss (MHL) for optimizing the bit error tolerance of BNNs. Our results show that the BNNs trained with the MHL achieve higher levels of bit error tolerance and accuracy compared to BNNs trained with the cross entropy loss (CEL) and bit flip injections according to a bit error model, while the MHL does not need a bit error model for achieving error tolerance.

5 BNNS WITH FEFET

In the vision of this dissertation, we propose to use approximate memory in systems that run BNNs. In this chapter, we explore how the emerging FeFET memory can be used as an approximate memory for BNNs.

In Sec. 5.1, we introduce FeFET memory and how it its key properties are simulated for our explorations. Then, in Sec. 5.2, we present the temperature-dependent error model of FeFET and show how errors occurring across the entire range of operating temperature can be tolerated by BNNs when countermeasures are employed. In Sec. 5.3, we use approximate FeFET-based Logic-In-Memory (LiM), which serves as both memory and performs computations and explore its tradeoff regarding LiM latency and BNN accuracy.

5.1 FEFET

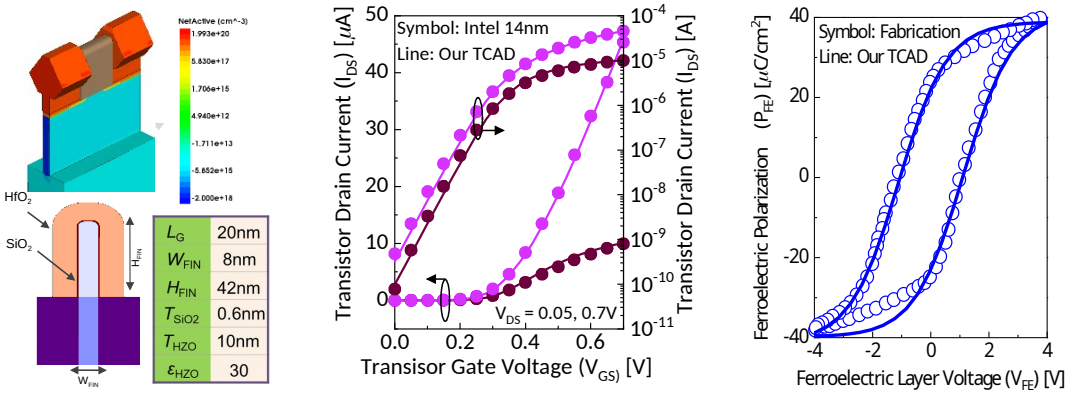
Non-volatile memories (NVMs) for machine learning algorithms may achieve highly energy-efficient and sustainable inference. In particular, NNs with different types of NVMs have been evaluated recently, e.g. resistive RAM (RRAM) [Chi+16; Hir+19b], spin-transfer torque RAM (STT-RAM) or magnetoresistive RAM (MRAM) [Vin+15; Pan+18; Sun+18; Hir+19a; TGW19], multi-level charge-trap flash (CTF) memory [Don+18], and ferroelectric-based memories (FeRAM or FeFET) [Che+18; Lon+18; ZCH19; Yoo+19].

In this section, we present the FeFET memory technology and how its properties are simulated to perform the evaluations in this chapter.

5.1.1 *Overview of FeFET Technology*

The discovery of ferroelectricity in hafnium oxide-based materials in 2012 has enabled Ferroelectric Field-Effective Transistor (FeFET) to become compatible with the existing CMOS fabrication process [Dü+17; Tre+16]. Prototypes from both academia and industry have thereafter successfully demonstrated the ability of turning conventional FET transistors into NVM devices by integrating a ferroelectric layer into the transistor gate stack. This allowed to integrate NVM devices alongside logic transistors within the same silicon die, unlike other existing NVM technologies that still face challenges regarding CMOS compatibility, which is a key factor for semiconductor manufacturing with reasonable cost.

Among all existing memory technologies, FeFET is one of the most promising emerging technologies. FeFET-based memories achieve read and write latencies



(a) 14nm FinFET transistor dimensions.

(b) 14nm FinFET calibration with Intel measurement data.

(c) Ferroelectric capacitor (FeCAP) calibration with fabrication data.

Figure 5.1: FeFET-based NVM device calibration.

within 1 ns, which is comparable to SRAM, while using low energy due to the nonvolatility [Rei+19]. Because FeFET-based memory cells consist of only a single transistor [Rei+19], they also feature a high density, which is of high importance for efficient on-chip memories.

FeFET is similar to conventional FET transistors used in all existing CMOS technologies, with one exception. In FeFET devices, a thick (10nm) layer of a ferroelectric material is additionally included on top of the oxide layer. The presence of ferroelectricity creates a hysteresis behavior by which the electrical property of the underlying transistor considerably shifts based on the previously-applied gate voltage.

More specifically, when a high positive write voltage (e.g. +4V) is applied to the transistor gate, then the polarization direction within the ferroelectric material is switched to a certain direction. After this write voltage ceases, a relatively high drain current will be returned every time by the transistor when a smaller voltage (e.g. 0.25V) is applied. The drain current in these cases is typically called low-V_{th} curve because the transistor has a low threshold voltage (V_{th}, recall that the threshold voltage is the gate voltage at which the transistor switches its state from "high" to "low" or vice versa). On the other hand, when a negative write voltage (e.g. -4V) is applied, the polarization direction within the ferroelectric material is switched to the opposite direction. After this write voltage ceases, a relatively low drain current will be returned by the transistor every time when a smaller voltage is applied. The drain current in these cases is typically called high-V_{th} curve because the transistor has a high threshold voltage.

Due to the hysteresis behavior, the transistor provides a low or a high current, which depends on the write voltage previously applied. The low and high currents allow the differentiation between stored logic "1", (when positive write voltage was previously applied, i.e. low-V_{th} case) and stored logic "0" (when negative write voltage was previously applied, i.e. high-V_{th} case).

5.1.2 *Our Calibrated 14nm FeFinFET Device and Measurements*

To enable working with and analyzing the properties of FeFET, we first implemented a 14nm n-type FinFET device, as shown in Fig. 5.1(a), using Synopsys Technology CAD (TCAD) tool flow [Syn], which is the standard commercial tool to simulate device fabrication of semiconductor technologies. Afterwards, we calibrated the device built to reproduce data measurements for production-quality 14nm FinFET device from Intel [Nat+14]. As shown in Fig. 5.1(b), the electrical characteristics regarding gate voltage ("voltage applied to transistor") and drain current ("current returned by transistor") of the device we built match very well with 14nm Intel measurement data. Afterwards, to enable the ferroelectric properties, we deposited a 10nm ferroelectric layer ($Hf_{0.5}Zr_{0.5}O_2$) on top of the oxide layer. Fig. 5.1(c) shows the hysteresis loop of voltage and polarization, which both capture the nonvolatile property in FeFET devices. The curves match very well the measurement data from a fabricated ferroelectric capacitor [Ni+18]. We will use these simulations and its properties in the following two sections.

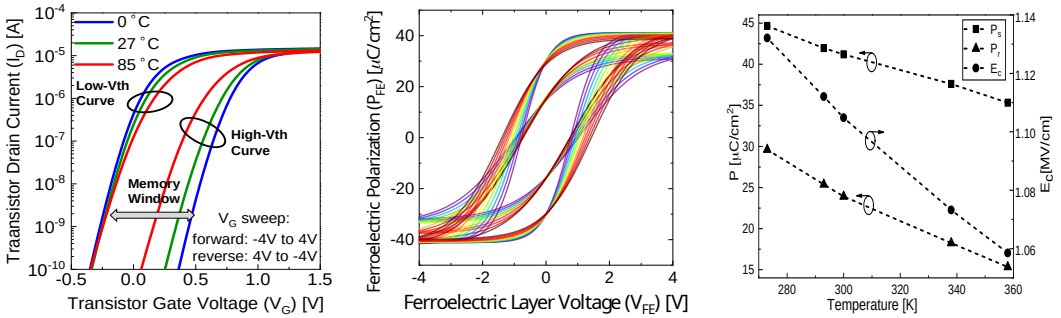
5.2 FEFET-BASED BNNS AND UNDER TEMPERATURE-DEPENDENT BIT ERRORS

In order to use FeFET as memory in systems that execute NNs, it is necessary to analyze its error model. To the best of our knowledge, the sources of bit errors in FeFET were not explored at the time of writing the published manuscript on which this chapter is based on.

Similar to other memory technologies, the errors in FeFET occur due to the physical principles of its operation. As explained in Sec. 5.1.1, in FeFET, the available dipoles, which determine the polarization within the ferroelectric layer, can store information, i.e. logic "0" and logic "1". However, the directions of those dipoles are sensitive to temperature. Specifically, fluctuations in temperature can lead to flippings of the dipoles' direction. This manifests itself as changes in the induced ferroelectricity and, hence, sensing circuits may read the erroneous stored value, i.e. a bit flip will occur during reading. Another source of errors is process variation (PV, unavoidable variations of transistor dimensions during manufacturing), either within the ferroelectric material or within the underlying transistor. PV manifests itself as changes in electrical characteristics of the FeFET device such as the threshold voltage, ON current, OFF current, etc. This leads, similar to temperature effects, to fluctuation in the sensing current during read operations. Therefore, bit flips can occur. This raises the following key question:

(Q1) *What is the bit error model of FeFET?*

We cover both types of bit flip sources in this chapter, i.e. run-time errors induced by temperature effects and design-time errors induced by PV. In addition to the above observation of bit errors on FeFET due to temperature and PV, our model in Sec. 5.2.1 also shows that the effect of temperature manifests itself as highly *asymmetric* bit error rates. This means that the probability of a bit flip from logical "1" to "0" is



(a) Temperature impacts on the memory window of FeFET. A higher temperature impact more considerably the high-V_{th} curve that represents logic "0" than low-V_{th} curve that represents logic "1".

(b) Impact of intrinsic variations in the ferroelectric layer on the induced hysteresis loop. Process variation together with temperature effects cause errors due to read operations of FeFET-based NVM.

(c) Relation between P_r (remnant polarization), P_s (saturation polarisation), and E_c (coercive field) over temperature. These three parameters capture the key properties of the ferroelectric material.

Figure 5.2: FeFET reliability experiments.

different from logical "0" to "1". In the literature, the effects of asymmetric bit error rates on NNs have not been assessed at the time of writing and no countermeasures against them have been investigated yet. For these reasons we explore the following questions:

(Q2) *Do the FeFET asymmetric bit errors cause significant accuracy drop in NNs?*

(Q3) *How can we exploit the asymmetry of the FeFET bit error model to achieve tolerance against FeFET bit errors?*

The remainder of this section is organized as follows. In Sec. 5.2.1, we answer **Q1** by revealing the impact of temperature on the reliability of FeFET-based memories by a temperature-dependent bit error model acquired from precise device parameters and FeFET simulations. Then, we cover the system model and the problem definition in Sec. 5.2.2 and Sec. 5.2.3. To answer **Q2** and **Q3**, we first discuss the less buffer writes (LBW) execution of BNNs in Sec. 5.2.4, propose our bit error countermeasures in Sec. 5.2.5, and evaluate the methods in Sec. 5.2.6.

5.2.1 *Temperature-dependent Bit Error Model of FeFET*

To acquire the bit error model of FeFET, we perform the following high-level steps. After integrating the temperature and variation effects inside our calibrated TCAD models, we perform Monte Carlo (MC) simulations for the entire FeFET device. This provides us with the complete I_D-V_G hysteresis loops. Then, for a certain read voltage, we extract the probability of error in which a high V_{th} curve is wrongly classified as a low V_{th} curve and vice versa.

In Fig. 5.2(a) we show the impact of temperature on the memory window of FeFET. The read operation in FeFET is performed by applying a gate voltage V_G (x-axis) to the transistor and then sensing the provided drain current I_D (y-axis). For different settings of the applied V_G, the temperature-induced shift in the high-V_{th} curve is different and hence reading at different V_G can result in different probability of errors. The memory window is defined as the distance between the low-V_{th} and high-V_{th} curves (i.e. I_D-V_G curves) which represents logic "1" and logic "0", respectively. Importantly, temperature increase has a more considerable impact on the high-V_{th} curve than the low-V_{th} curve. At a higher temperature, the high-V_{th} moves towards the left side whereas the low-V_{th} curve remains almost unaffected. Hence, the memory window becomes smaller and the error tolerance of FeFET-based NVM becomes smaller, due to which the likelihood of errors during read operations becomes larger. Because temperature impacts the low-V_{th} curve and the high-V_{th} curves *asymmetrically*, also errors in logic "1" and "0" will occur *asymmetrically*. This means in practice that the bit error rate p_{10} (flip from $1 \rightarrow 0$) is smaller than the bit error rate p_{01} (flip from $0 \rightarrow 1$) because temperature impacts logic "0" more than logic "1".

In Fig. 5.2(b), which is based on MC simulations in TCAD, we observe that intrinsic variations from PV within the ferroelectric layer strongly impact the induced hysteresis loop. This, in turn, seriously reduces the error tolerance of FeFET-based NVM devices to noise and increases the probability of error. In fact, both temperature and PV effects together degrade the reliability of FeFET-based NVM devices during run-time and thus errors during read operations occur.

In Fig. 5.2(c), we summarize the relation between temperature and P_r (remnant polarization), P_s (saturation polarization), and E_c (coercive field), which capture the key properties of the ferroelectric material. The three parameters degrade linearly with temperature increase. Therefore, we use the value $T_{step} \in \{0, 1, \ldots, 16\}$ for describing the magnitude of temperature. Based on Fig. 5.2(c), the bit error rate at temperature T for any $0°C \leq T \leq T_{peak} = 85°C$ is $BER(T) = \frac{T}{T_{peak}} \cdot (p_{01}, p_{10})$, where p_{01} and p_{10} are defined at 85°C above.

Since the probability of bit error also depends on the applied read voltage (i.e. gate voltage V_G), we estimate the flip probabilities (p_{01}, p_{10}) at different read voltages 0.1V and 0.25V. The bit error rates of the FeFET bit error model at 85°C are $(p_{01}, p_{10}) = (2.198\%, 1.090\%)$ for using read voltage 0.1V and $(p_{01}, p_{10}) = (2.098\%, 0.190\%)$ for using read voltage 0.25V.

5.2.2 *System Model*

In the following, we focus on studying the impact of FeFET-induced bit errors described in Sec. 5.2.1 on the accuracy of BNNs and then investigating the tradeoffs between read energy and reliability of FeFET devices. Because in NN systems, the read operations occur much more frequently than write operations due to data reuse [Sze+17] or low cost of storing data in NVMs, reducing the read voltage provides considerable energy saving. In practice, we assume that FeFET memory

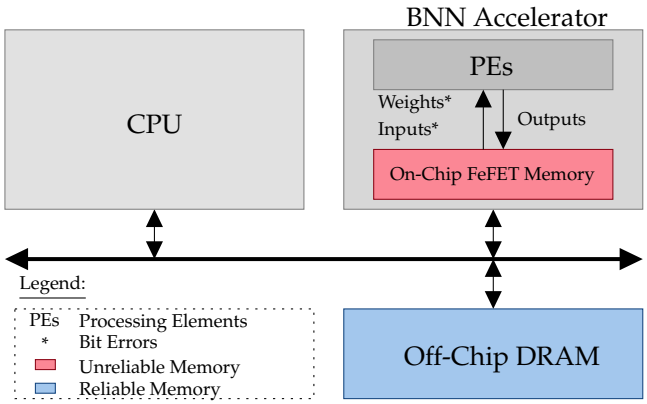


Figure 5.3: System model with unreliable on-chip FeFET memory and reliable off-chip DRAM.

devices are being reliably written, i.e. they receive a sufficiently large voltage to flip all ferroelectric domains during writing. Then, during read operations the bit errors due to temperature occur.

For these reasons, we focus on the scenario that bit errors only occur during the reading processes of the parameters (weights) and inputs (input data and activations) in BNN inference. To clarify, we present the considered system model with in Fig. 5.3. The assumed system consists of reliable traditional off-chip memory (e.g., DRAM) and unreliable emerging on-chip FeFET memory. To perform the computations of one layer, the CPU initiates the retrieval of the weights from the off-chip DRAM to the on-chip FeFET memory. Then, the values are sent to the processing elements (PEs) for executing the operations of BNNs in parallel (i.e. XNOR, popcount, accumulation, and thresholding, etc.). The results of the computations, which are the activations, are then written back to the on-chip FeFET memory in order to be later used in the computations for the subsequent layer. Data and instructions for other operations, which are related to the control of the inference, e.g. from the operating system to provide a run-time environment to initiate the inference, are not stored in the FeFET memory, but are stored in a reliable memory, e.g. off-chip DRAM.

Please note that NVM using ferroelectric transistors is an emerging memory, and commercial processors that employ such technology are not yet publicly available. Therefore, we do not use real FeFET memory for running the experiments. The way we model the usage of FeFET is by applying the corresponding bit error model during training and inference on conventional servers with GPUs. Our focus is on the effect of FeFET bit errors on BNN accuracy, when on-chip FeFET memory would be used for storing the weights, inputs, and activations. Therefore, for evaluating the bit error tolerance of our BNNs with respect to the FeFET-induced errors, the application of the bit error model is sufficient and the real FeFET memory does not need to be used.

5.2.3 *Problem Definition*

The two bit error tolerance (BET) problems below are studied in the following:

- **BET Training Problem**: Given the FeFET temperature bit error model described in Sec. 5.2.1 and a set of labeled input data, the objective is to train a BNN for high accuracy. *The inference of the derived BNN does not have to be executed with any bit error countermeasures at run time.*
- **BET During Inference Problem**: Given the FeFET temperature bit error model as described in Sec. 5.2.1 and a BNN, the objective is to execute the given BNN with bit error countermeasures to reduce the accuracy degradation of the BNN during runtime. *The given BNN does not have to be trained with bit flip injection.*

The solutions to the above two problems can be combined to yield better bit error tolerance.

5.2.4 *BNN Execution with Less Buffer Writes*

Before addressing the two problems in Sec. 5.2.3, we first present the less-buffer-writes (LBW) BNN execution, in which the BNNs are executed in a way such that less layers are prone to bit errors compared to regular BNNs.

The FeFET bit errors can only have an effect on BNN accuracy when values are read from FeFET memory. For this reason, the buffering of data (which implies writes followed by reads) in FeFET memory should be avoided whenever possible. In the regular BNN execution, however, values are buffered multiple times, as shown in Table 5.1, which leads to many (avoidable) reads from FeFET memory. The values that are buffered to memory during execution are the intermediate results, i.e. the outputs of the convolution (C), maxpool (MP), and batch norm (BN) layer.

In order to minimize the buffering to FeFET memory, we use the LBW BNN execution. The LBW execution aims to reduce the reads from FeFET memory so that less values are affected by bit errors. When using the LBW execution, only the BNN parameters, inputs, and outputs of the BN layer (activations) suffer from bit errors. The flow of the LBW execution is listed in Table 5.1 and it is explained in the following.

In the LBW execution, we compute the C and MP in such a way that they are applied without buffer writes to memory. The LBW execution begins with the C layer. In the first iteration of the LBW execution, the C layer computes the convolution results of the first four values $(0,0),(0,1),(1,0),(1,1)$ in the input. By MP, the maximum of the four values is then computed and thresholded with the BN layer to produce a binary output. After the threshold has been applied, the result has to be buffered in memory, so that the next C layer of the BNN can compute with it. In the next iteration of the

Regular execution	In → C → buffer writes
	→ buffer reads → MP → buffer writes
	→ buffer reads → BN → buffer writes
LBW execution	In → 4C → MP → BN → buffer writes

Table 5.1: The regular BNN execution with many buffer writes to memory and the less-buffer-writes (LBW) execution. Another layer configuration that we use is C → BN, in which case the thresholding of the BN is applied directly to the C result.

LBW execution, the next four values in line $((0,2),(0,3),(1,2),(1,3))$ are processed the same way as in the first iteration. These iterations are repeated until the input to the C layer is fully processed. Due to the processing sequence of the LBW execution, only a buffer of two values is needed between the C → MP → BN computation. One value is needed for holding the current maximum and the second for the current MP result. For the case of C → BN layer compositions, the output of the convolutions do not need to be buffered because the thresholding can directly be applied to the output of the convolution.

The number of executed operations in the LBW execution is equal to the number of executed operations in the regular way of execution. The operations of the LBW are simply executed in a different sequence and thus data from the memory is retrieved in a different sequence as well. The implications of the LBW execution on inference efficiency have to be investigated case-by-case for different inference systems during system design. Here, we do not assume any specific inference system. FeFET could e.g. be used as on-chip memory for BNN data while processing elements in an accelerator execute the BNN operations. In these cases, the operations of LBW BNNs are executed in ALUs and the values for accumulation and maxpool operations are stored in registers. We assume that once the values are in these components, they are not affected by bit errors anymore.

Since the correctness of the last-layer BNN outputs and the batch normalization thresholds (see [SBN19]) are indispensable for BNN accuracy, we assume that this small fraction of values is protected by software or hardware measures such as error-correcting code (ECC) or correcting implausible values with memory controller support (see [Kop+19]).

5.2.5 *Methods for Achieving Bit Error Tolerance against FeFET Bit Errors*

We present two different methods against the FeFET bit errors. First, we describe how we take the asymmetry into account in bit flip training in Sec. 5.2.5.1, targeting the BET training problem posed in Sec. 5.2.3. Then, in Sec. 5.2.5.2, we present our bit error rate assignment algorithm (BERA) which exploits the asymmetry of the FeFET bit error model in a layer-wise manner with the goal of minimizing accuracy drop. This targets the BET during inference problem.

5.2.5.1 *Bit Flip Injection During Training*

In order to achieve high accuracy, we train the BNNs by minimizing the cross entropy loss (CEL), as described in Ch. 3. To also train BNNs for bit error tolerance against the FeFET bit errors, we use bit flip injection during training, as proposed in [Hir+19b]. However, simply applying this approach without taking the asymmetry into account can lead to unacceptable accuracy drops, therefore, additional steps to the existing methods need to be taken. Specifically, the asymmetry needs to be taken into account evaluating and training with all bit error rate settings $(p_{01}, p_{10}) \in \{(2.198, 1.090), (1.090, 2.198), (2.098, 0.190), (0.190, 2.098)\}$ by injecting bit flips with (p_{01}, p_{10}) into all the BNN data that is prone to bit errors. By this we find out which setting leads to the least accuracy drop under temperature dependent bit errors.

5.2.5.2 *Bit Error Rate Assignment Algorithm (BERA)*

We present BERA, which exploits the asymmetry of the FeFET temperature bit error model in a layer-wise manner. The goal of BERA is to reduce the effect of the bit errors by finding the layer-wise bit error rate configurations which maximize accuracy, without bit flip training.

BERA operates in two steps. In the first step, the accuracy drop of the entire NN is estimated by measuring the accuracy after injecting bit errors into one layer of the network (with the training set). Because of the random nature of the errors, we repeat the injection a certain number of times and average in the end. In the second step, the setting with the lowest accuracy drop is chosen and assigned to the layer.

The algorithm realizing BERA is shown in Alg. 4. We first set *bers*, the bit error rates, in Line 1. Then we initialize an array for all layers that suffer from bit errors in Line 2, with a subarray for every bit error rate configuration, since we estimate the accuracy drop of every configuration. The value *reps* is the number of repetitions through the entire training data set for each bit error rate setting. With this parameter, the precision of the accuracy drop estimation can be tuned. We measure the accuracy without errors on the training set in Line 4. The accuracy drop is estimated for every bit error rate setting in every layer individually in the loop. Finally, the results of the estimation are stored in an array called *adpl* which is normalized with the number of repetitions after Line 12. Then, from the *adpl* array, the setting with the lowest accuracy drop per layer is chosen and assigned to the layer at hand. We call this assignment the Greedy-A assignment.

5.2.6 *Experiments for FeFET Temperature Bit Error Tolerance*

We first evaluate the LBW execution of BNNs and how it compares to the regular BNN execution in Sec. 5.2.6.1. Then, in Sec. 5.2.6.2, we assess the impact of FeFET bit errors on BNN accuracy without any countermeasures. We apply the well-known bit flip training while taking the asymmetry of the FeFET bit error model into account (addressing the BET training problem). We also evaluate our BERA algorithm that

Algorithm 4: Accuracy drop per layer estimation for our BERA algorithm

Input: $model, (\mathbf{X}_{train}, y_{train})$
Output: adpl
// Initialize bit error rates
$bers = \{c_1, \ldots, c_S\}$
// Accuracy drop per layer (adpl)
$adpl = \{\{ad_{1,1}, \ldots, ad_{1,S}\}, \ldots, \{ad_{L,1}, \ldots, ad_{L,S}\}\}$
// Number of repetitions for a bit error rate setting and layer
$reps = R$
// Measure accuracy of model
$acc_v = accuracy(model(\mathbf{X}_{train}), y_{train})$
for *each layer* $l \in \{0, \ldots, L\}$ **do**
  for *each* c_i *in bers* **do**
    set bit error rate tuple c_i only for layer l
    for *each* r *in* $r \in \{0, \ldots, reps\}$ **do**
      $acc_{l,c_i,r} = accuracy(model(\mathbf{X}_{train}), y_{train})$
      $adpl[l][c_i] = adpl[l][c_i] + acc_{l,c_i,r}$
    reset bit error rates in l
for *each layer* $l \in \{0, \ldots, L\}$ **do**
  for *each* c_i *in bers* **do**
    $adpl[l][c_i] = acc_v - \frac{adpl[l][c_i]}{R}$

exploits the asymmetry by operating in a layer-wise manner to minimize the impact of the FeFET bit errors (addressing the BET during inference problem).

5.2.6.1 *LBW Execution of BNNs*

In all the following experiments we use LBW BNNs, which we proposed in Sec. 5.2.4. Here, we conduct evaluations for demonstrating that our assumption of using the LBW execution of BNNs is reasonable and, additionally, to quantify the impact of such an assumption on system performance. The LBW execution of BNNs buffers less intermediate results to the memory during execution than the usual way of execution. We want to demonstrate that the LBW execution of BNNs does not lead to significantly higher execution times compared to the regular way of execution. We use the LBW execution in the error tolerance evaluations, to justify bit flip injection in the weights, input images, and activations, instead of injecting bit flips in all intermediate results. Here, we evaluate the execution time of LBW BNNs and compare it to regular ones using commonly available CPUs for the BNN operations. As experiment platforms for the execution time measurements we use the same hardware setup as in [Bus+18]. Furthermore, we generate C++ code from PyTorch models with the same framework as the study in [BJBP20], and implement in the C++ code the LBW execution. For

Platform	BNN-type	FASHION (ms/el.)	CIFAR10 (ms/el.)
Intel	Regular	1.09	26.47
	LBW	1.05	32.15
ARM	Regular	12.28	305.09
	LBW	11.03	312.79
PPC	Regular	4.55	210.03
	LBW	12.45	306.30

Table 5.2: Average execution times evaluation for the regular and LBW BNNs on different platforms and datasets. The values are in ms per one BNN evaluation. Each BNN was evaluated 10^4 times as compiled C++ code.

Intel we used a Intel i7-8550U CPU with 1.80GHz and 16 GB RAM. For ARM we used an ARM Cortex-A53 with 1.4 GHz and 1GB RAM (RaspberryPi 3B+). For PPC we used a QorIQ T4240 PowerPC CPU with 1.67 GHz and 6 GB RAM. To summarize the results in Table 5.2, the execution times for FASHION and CIFAR only differ by a large factor for PPC, and for the other settings the execution times do not differ by a large margin.

5.2.6.2 *The Impact of FeFET Bit Errors on BNN Accuracy*

For the experiments, we use the convolutional BNNs, VGG3 and VGG7, from Ch. 3, shown in Table 3.2, with the Fashion, SVHN, and CIFAR10 datasets, shown in Table 3.1. We inject the bit flips in the weights, activations and inputs by implementing a bit flip injection tool in PyTorch, as explained in Ch. 3. All training and testing experiments are repeated 10 times due to the random nature of the bit flips.

We plot the accuracy over bit error rate for all experiments in Fig. 5.4. For the evaluation of accuracy over the temperature-induced bit error rates, we inject bit flips with the bit error rates presented in Sec. 5.2.1, with the combinations $(p_{01}, p_{10}) \in \{(2.198, 1.090), (1.090, 2.198), (2.098, 0.190), (0.190, 2.098)\}$. With the temperature steps $T_{step} \in \{0, 1, \ldots, 16\}$, we evaluate the full bit error range, which corresponds to the entire range of operating temperature considered in our FeFET analysis. We incorporate these temperature steps in the evaluation by defining $T^*_{step} = \frac{1}{16} T_{step}$ in $BER(T^*_{step}) = T^*_{step} \cdot (p_{01}, p_{10})$. In the accuracy over bit error rates plots, we annotate the x-axis with T^*_{step}.

Impact on BNNs with no countermeasures: For BNNs trained without errors (top row of Fig. 5.4) the impact of the temperature bit errors can be substantial if no bit error training is used and when no attention is paid to the asymmetry of the bit error rates. We find accuracy degradation of over 25% for Fashion, over 30% for CIFAR, and over 7% for SVHN at the highest operating temperature $T^*_{step} = 1$.

Bit flip injection during training: When training with bit flip injection (bottom row of Fig. 5.4), we achieve bit error tolerance for the entire range of operating temperature. The asymmetry of the bit error model, however, plays a key role in these

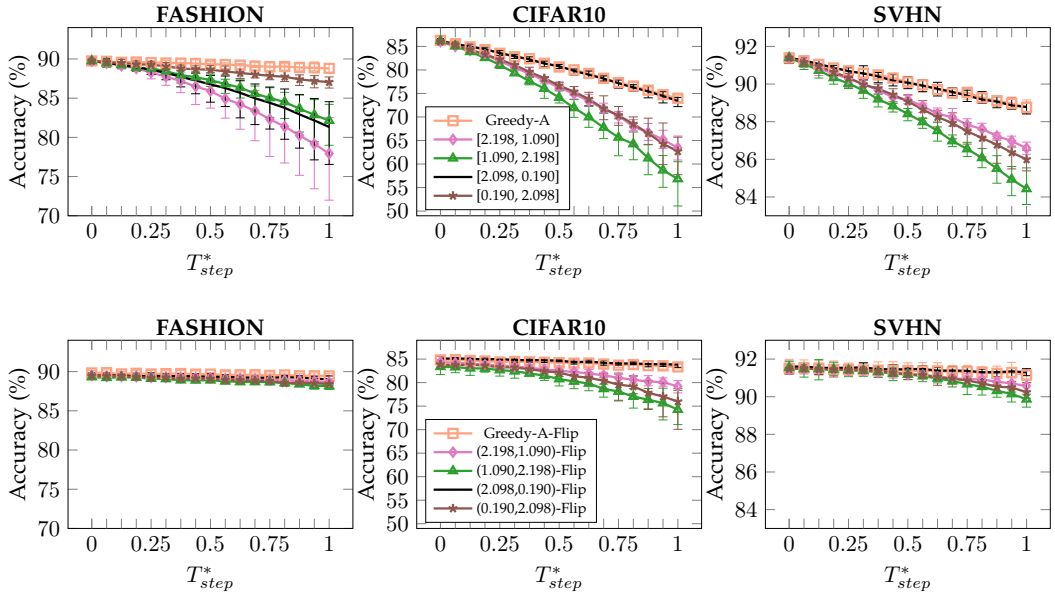


Figure 5.4: Accuracy over temperature-dependent error rates. Top row: No bit flips during training, and in the orange plot (Greedy-A) the result for applying BERA after training. Bottom row: Bit flip training with all bit error rate configurations and in orange (Greedy-A) the retraining with BERA. $T^*_{step} = \frac{1}{16} T_{step}$ and $T_{step} \in \{0, 1, \ldots, 16\}$. $BER = T^*_{step} \cdot (p_{01}, p_{10})$ yields the temperature dependent bit error rate setting. Greedy-A is the accuracy optimal assignment acquired after executing BERA. In the other four settings, every layer of the BNN is configured with the same bit error rate. E.g., when we write $(2.198, 1.090)$, then every layer of the BNN is configured with these bit error rates.

experiments. The differences among the highest and the lowest Fashion curves is below 0.2% for $T^*_{step} = 1$. For $T^*_{step} = 1$ this difference is 0.85%. For CIFAR10, these gap are larger. First, for $T^*_{step} = 0$, the difference between the plot with the highest and the lowest accuracy is 1.75%. For $T^*_{step} = 1$, this difference is 9.6%. Furthermore, when comparing the CIFAR10 plots without bit flip training (left column) with the bit flip trained plots at $T^*_{step} = 0$, the highest plots differ by 1.1%. In this case, the bit flip training drops the accuracy by more than one percent. For the other datasets, this accuracy tradeoff amounts to merely around 0.2%. For SVHN, the difference among the plots at $T^*_{step} = 0$ is below 0.2% and for $T^*_{step} = 1$ it is 1.4% when comparing the highest accuracy plot with the lowest. In all three datasets, the setting $(2.098, 0.190)$ is the best one among the four bit error rate settings.

Bit Error Rate Assignment Algorithm (BERA): We present the result for BERA in the accuracy over bit error rate plots in Fig. 5.4 as Greedy-A (orange plots). In the top row, we show the plots for only using BERA to protect the BNNs and without bit flips during training. The Greedy-A plot is able to protect the BNN from bit

errors to a similar extent as using bit flip training. The difference at $T^*_{step} = 1$ is 0.66% when compared to BNNs retrained with Greedy-A. For the other two datasets, the Greedy-A assignment can only reach the same accuracy as $(2.098, 0.190)$, since that setting is the best one for every layer. We also evaluate the combination of BERA and bit flip training. In the bottom row we show the plots (in orange) for retraining with the Greedy-A setting after training without bit flips. We retrained Fashion BNNs for 10 epochs, CIFAR10 for 200, and SVHN for 50. For Fashion, the Greedy-A assignment improves accuracy by 0.3% compared to the other settings. This eliminates the tradeoff in accuracy when injecting bit flips during training.

5.3 FEFET-BASED LIM FOR BNNS

Several recent studies have demonstrated how a boolean logic function (e.g., XNOR, NAND, etc.) can be realized within conventional SRAMs [Agr+18] and emerging NVMs [Ni+19], which is referred to as Logic in Memory (LiM). In LiM, the memory element acts as both, a storage and a computing unit. Using LiM promises less data movement compared to traditional von Neumann systems, since the data is already present in the processing units, due to which expensive data movements do not need to be performed. As one of the main bottlenecks in systems that execute NNs is the data movement, LiM constitutes a viable design paradigm for efficiency in NNs.

Specifically, realizing LiM based on emerging NVMs is rapidly gaining attention because of the non-volatility. It provides significant power savings compared to SRAMs, which suffers from high static power. Furthermore, the high density (of NVMs) allows accommodating large amounts of data within a small area.

As discussed in Sec. 5.1, among the emerging NVM technologies, FeFET is an outstanding candidate for memory in efficient systems with numerous advantages. On top of its low resource consumption and CMOS-compatibility, realizing a LiM-based XNOR using FeFET only needs 2 transistors compared to 16 transistors using SRAM [Ni+19]. FeFET-based LiM and BNNs are a promising combination of emerging techniques for extremely efficient deployment. For these reasons, we believe that FeFET-based XNOR LiM with BNNs is a highly synergystic opportunity.

Nevertheless, compared to conventional CMOS logic, the latency of LiM using an emerging technology is often longer. As we will show later, a FeFET-based XNOR LiM gate has a latency of around 0.7 ns, whereas popcount and activation circuits that are needed for BNN hardware have a latency of around 0.43 ns and 0.34 ns [CGC21; GNA18], respectively. This makes the FeFET-based XNOR LiM a bottleneck in the processing latency.

To decrease the latency of the XNOR LiM computation, overclocking can be employed, in which a clock speed higher than the recommended logic latency is used. However, setting the clock speed too high may result in errors, i.e. in the form of bit flips. Here, the error tolerance of BNNs can be exploited for higher latency. To the best of our knowledge, at the time of the writing there have not been any studies that exploit the error tolerance of BNNs and trade off BNN accuracy with the ex-

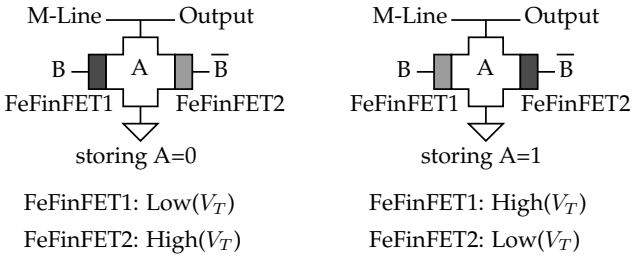


Figure 5.5: The implementation of FeFET-based XNOR that consists of two FeFinFETs storing the logic value A in a complementary manner.

A	B	FeFinFET1	FeFinFET2	Discharge	M-Line	V_{out}	Output
0	0	OFF	OFF	No	match	stays high	1
0	1	ON	OFF	Yes	mismatch	drops	0
1	0	OFF	ON	Yes	mismatch	drops	0
1	1	OFF	OFF	No	match	stays high	1

Table 5.3: The basic operation and the realization of XNOR boolean function by the FeFET-based XNOR gates in Fig. 5.5. The XNOR's output is '0' if and only if A $\neq$ B. Otherwise, the XNOR's output is '1'.

tent of errors in FeFET-based XNOR-LiM computations stemming from low latency configurations.

In this section, we explore the tradeoff between the probability of error and speed that FeFET-based XNOR offers. We then demonstrate how the error error tolerance of BNNs can be increased at the *design-time* and exploited at *run-time* to decrease overall inference latency of LiM. Specifically, we investigate how PV impacts FeFET-based XNOR and model the relation between probability of error (P_{error}) and computation speed. We then explore how XNOR-induced errors can degrade the inference accuracy of BNNs. Afterwards, we present how error tolerance in BNNs can be achieved through proactively training the BNNs in the presence of the XNOR-induced errors. Finally, we investigate the sensitivity of every individual layer in BNNs to XNOR-induced errors. Then, we demonstrate how a selective approach in which the speed of FeFET-based XNOR is adjusted for every layer, trading-off P_{error} with latency. In exploring this trade off, we maximize the inference speed under a certain inference accuracy drop constraint.

The remainder of this section is structured as follows. In Sec. 5.3.1, we present our FeFET-based XNOR-LiM model and in Sec. 5.3.2 we discuss its error model. In Sec. 5.3.3, we present the system model and design objective. The two techniques to trade off errors with latency are introduced in Sec. 5.3.4 and they are evaluated in Sec. 5.3.5.

5.3.1 *FeFET-based XNOR LiM Model*

The LiM realizing the XNOR-logic function can be achieved through coupling two FeFET transistors together [Ni+19], shown in Fig. 5.5 and the detailed states in Table 5.3. A value $A \in \{0,1\}$ is stored inside the XNOR in a complementary manner and $B \in \{0,1\}$ is the input to the XNOR. When $A = 0$, FeFinFET$_1$ is in the low and FeFinFET$_2$ is in the high V_T state. Depending on whether the value B matches the stored value A, the XNOR output will be either "0" or "1". In practice, the match line (ML) is charged to high (V_{dd}) and when A is equal to B, both FeFinFET1 and FeFinFET2 will be OFF. In this case, no conducting path is formed and the voltage remains high. Therefore, the XNOR's output is logic "1". Only when A is not equal to B, a conducting path is formed through the FeFET that is in low V_T state. In this case, the voltage rapidly drops and the output provides logic "0".

To allow the "overclocking", the XNOR gate is implemented as a dynamic logic gate. Dynamic XNOR gates have been proposed in the past [TM00; GDM83] and have the ability to achieve a high speed, small area, full voltage swing, and good driving capability [Goe+06]. Dynamic XNOR gates are clocked and work in two phases: Precharge phase and evaluation phase [Wan+11; RCN03]. In the precharge phase, the output node is charged to V_{dd}, and in the evaluation phase, the output node either discharges or does not discharge depending on the combinations of the inputs applied. Realizing the computations required by BNNs using dynamic XNOR logics in which a clock is available enables run-time adaptation in which the speed of calculation can be adjusted as a tradeoff with accuracy.

5.3.2 *Variability and Error Modeling in FeFET-based XNOR-LiM*

We implement and simulate the FeFET-based XNOR LiM gate (see Fig. 5.5) using the commercial SPICE tool Cadence Spectre. To capture the effects of manufacturing variability, we perform 10,000 Monte Carlo (MC) simulations for the FeFET-based XNOR gate. The analysis is performed for all four possible configurations, i.e. XNOR(0,0), XNOR(1,0), XNOR(0,1), and XNOR(1,1). In Fig. 5.6(a), we present the results for the case of XNOR(0,1) as an example. To acquire P_{error} with variability in the XNOR output, we analyze the obtained results from the MC SPICE simulations. Here, we differentiate between the following two scenarios: Scenario 1: For the case of match, where the input of the XNOR matches the value stored within the XNOR(i.e. XNOR(0,0) and XNOR(1,1)), both FeFinFET$_1$ and FeFinFET$_2$ are in the OFF state (see Table 5.3). Hence, no current will be flowing through any of transistor (except a tiny leakage current). In such a case, the output remains at the high voltage and transistor variations has no impact because both FeFinFETs are OFF. Thus, no errors will be incurred, i.e. $P_{error} = 0$ for XNOR(0,0) and XNOR(1,1). Scenario 2: For the case of *mismatch* in which the input of the XNOR differs from the value stored within the XNOR (i.e. XNOR(1,0) and XNOR(0,1)), one of the two FeFET transistors is in the ON state (see Table 5.3). In this case, a conducting path is formed and a current flows through one of the FeFinFETs.

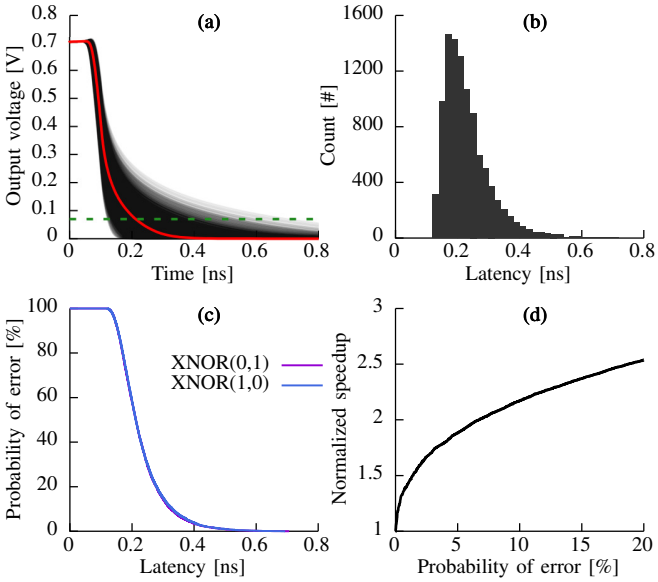


Figure 5.6: (a) Impact of process variation on FeFET-based LiM for XNOR(0,1). (b) The resulting distribution of XNOR output voltage subject to the XNOR latency (speed). (c) The probability of error (P_{error}) of FeFET-based XNOR as function of latency. (d) The tradeoff in FeFET-based XNOR between speed and reliability (P_{error}).

The output drops from high to low (see Fig. 5.6(a)) and transistor variations have an impact leading to errors in the XNOR output.

Due to variation effect, errors are incurred. In practice, the output is considered "0" when the voltage drops below the threshold of 10% of V_{dd}. Therefore, depending on the selected latency (i.e. propagation delay) the XNOR output may mistakenly be "1" instead of "0" due to variation effects. In Fig. 5.6(b), we present the distribution of the XNOR latency at which the XNOR output drops below the voltage threshold (10% $V_{dd} = 0.07$V). In Fig. 5.6(c), we present the P_{error} as a function of latency. For a smaller latency (i.e. at higher XNOR speed), the P_{error} becomes larger, as expected. Note that P_{error} for the case of XNOR(1,0) and XNOR(0,1) is approximately the same, because both FeFinFETs that form the XNOR are symmetric and subject to the same amount and source of variation. Also note that here, the probability of error goes up to 100%, which would be the case when the input is applied and only a short time (0.1 ns) passes. The output will always be "1". Finally, Fig. 5.6(d) summarizes the tradeoff between the speedup of the FeFET-based XNOR and reliability regarding P_{error}.

5.3.3 *System Model and Design Objective*

We model the usage of FeFET-based XNOR in our system by assuming that the results of XNOR operations in our BNNs follow the same probability of error obtained from our hardware analysis (details in Sec. 5.3.2). This means, with every execution of XNOR(0,1) or XNOR(1,0), the XNOR result can flip from "0" to "1" with probability

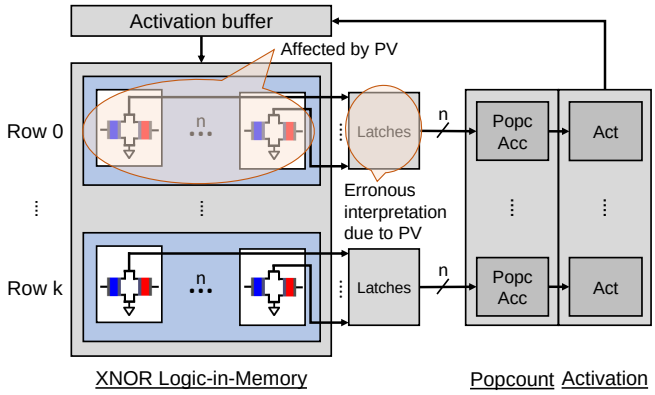


Figure 5.7: The considered system model used for BNN computations. The FeFET-based XNOR gates, which store the binary weights, are organized as rows, operate in parallel, and are connected to Popcount and activation circuits. Note that the XNOR gates are implemented as dynamic logic. Hence, depending on the clock speed, internal XNOR latency, and process variation the an erroneous computation might be latched and passed to the Popcount. The shown system configuration is just an example and it should be considered without loss of generality.

P_{error}, which is a function of XNOR latency (see Fig. 5.6(d)). Note that when computing XNOR(0,0) and XNOR(1,1) during inference, P_{error} of 0 is applied, as explained in Sec. 5.3.2. Convolutional and fully connected layers are subject to the XNOR-induced errors because only in these layers XNOR operations are performed.

In BNN accelerators that use the FeFET-based XNOR gates, data needs to move from the XNOR gates to the popcount units, and then to the activation units. These three components might be connected in various system configurations. One example is shown in Fig. 5.7. Note that this work focuses on how errors caused by the XNOR gates, when traditional CMOS-based XNOR gates are replaced with emerging FeFET-based XNOR gates, can propagate from there to corrupt the popcount and activation units and ultimately reduce the BNN inference accuracy. In general, every XNOR gate needs to be connected to a popcount unit by a wire, e.g. by multiplexing when popcount units are shared, or direct connections when a certain number of XNOR gates each are connected to own popcount units. The signals coming from the XNOR output then need to be latched, to enable the signal propagation to the popcount unit. When they are latched, the XNOR outputs are interpreted as a logic "1" or logic "0". Due to the impact of variation on FeFET-based XNOR, errors when latching the XNORs output could occur. Modeling these errors and then injecting them in the BNN is what our study focuses on. Note that after latching, all subsequent computations of a layer (i.e. popcount and activation) are performed in a "reliable" way. However, the errors propagating from the previous component (i.e. FeFET-based XNOR) may still corrupt them.

Previous work on BNN hardware operate with similar assumptions [Che+18]. Popcount and activation can either be computed in the analog domain (e.g. by

the sum of currents and analog comparators [Che+18]) or in the digital domain (e.g. popcount and digital comparison units [CGC21]). The optimization of these components is not considered in this work. However, it is important to note that state-of-the-art popcount units operate in the range of sub-nanoseconds (e.g. 0.43 ns, see [CGC21]), while the activation can be computed in the same latency scale as well (e.g. 0.34 ns, see [GNA18]). Lower latency can certainly be achieved in these components, depending on the realization. The tradeoffs need to be evaluated case-by-case. Here we want to show that the optimization of the XNOR gate latency (0.71 ns in this work) has significant effects on the latency of BNN accelerators, since the FeFET-based XNOR gate is one of the main bottlenecks in latency.

Design Objective: Our key objective is how the demonstrated tradeoff between reliability and latency in FeFET-based XNOR (shown in Fig. 5.6(d)) can be employed in error tolerant BNNs, in which higher inference speed is obtained at a small accuracy loss. We perform our investigation in a scenario with a given set of labelled input data and a BNN with stochastically binarized inputs performing multiclass classification. Our goal is to evaluate solutions to the following design problem: *Given the tradeoff between the latency and P_{error} of FeFET-based XNOR, find the smallest latency (i.e. the highest P_{error}), at which the XNOR can operate, while still sustaining a certain inference accuracy of BNN*. Here, we define the setting $\mathbf{p}$ as the set containing the P_{error} of each layer, where $\mathbf{p} = \{p_0, \ldots, p_L\}$ has elements p_ℓ with $0 \leq \ell \leq L$, and $\ell = L$ denotes the last layer in the BNN. To achieve this objective, we propose two techniques to obtain error tolerant BNNs.

(1) Design-Time Technique: The BNN is optimized with error tolerance training. Based on the achieved tolerance, $\mathbf{p}$ is set, such that a certain inference accuracy is still sustained. Here, the same p_ℓ is set for every layer in the BNN.

(2) Run-Time Technique: Unlike the design-time technique, no re-training is required here. $\mathbf{p}$ is derived by post-training evaluations in which different layers exhibit different P_{error} (i.e. XNORs in different layers operate at different speeds), while a certain inference accuracy is still sustained.

5.3.4 *Trading-off Reliability and Speed: Error Tolerant BNNs under XNOR Errors*

In this subsection, we cover the two techniques for obtaining error tolerant BNNs, the design-time technique in Sec. 5.3.4.1 and the run-time technique in Sec. 5.3.4.2.

5.3.4.1 *Error Tolerant BNNs during Design-Time*

We use the modified hinge loss (MHL) for achieving high accuracy and error tolerance in our BNNs from Ch. 4. The MHL directly maximizes the margins in the output layer, which leads to higher accuracy and error tolerance than using the standard cross entropy loss. To achieve even higher error tolerance, the suggested method in Ch. 4 is to combine the MHL with application of the error model during training.

Before discussing the error model application during training, we first develop an intuition for the effects that the XNOR errors cause in the computations of the BNNs. The effect of the XNOR error model (as described in Sec. 5.3.2) is a positive offset in the result of the popcount. This means, the result popcount(XNOR($\mathbf{W}^\ell, \mathbf{A}^{\ell-1}$)) from Eq. (2.4) gets shifted by $E(\ell, p_\ell)$, leading to

$$\text{popcount}(\text{XNOR}(\mathbf{W}^\ell, \mathbf{A}^{\ell-1})) + \mathbf{E}(\ell, p_\ell),$$

because only the results of XNOR(0,1) and XNOR(1,0), which are both "0", can flip to "1" with a given error probability of p_ℓ. The error tensor is defined by

$$\mathbf{E}(\ell, p_\ell) = \text{popcount}(\text{XNOR}_{\underline{p_\ell}}(\mathbf{W}^\ell, \mathbf{A}^{\ell-1})) - \text{popcount}(\text{XNOR}(\mathbf{W}^\ell, \mathbf{A}^{\ell-1})),$$

where $\text{XNOR}_{\underline{p_\ell}}$ specifies the usage of the XNOR gates that have probability of error p_ℓ. The tensor $\mathbf{E}(\ell, p_\ell)$ contains the error values that shift the result of the popcount due to the XNOR errors. Note that the error values in $\mathbf{E}(\ell, p_\ell)$ are always positive, depend on the number of XNOR (1,0) and XNOR (0,1) operations to compute a convolution in layer ℓ, and on a probability of error specified as p_ℓ.

Applying this error model during training is a challenge. For error model application during training, the output of affected XNOR operations need to be flipped, which requires replacing the highly optimized MAC libraries used in NN training frameworks with customized ones. This slows down the training considerably. In order to achieve efficient error model application during training, we develop a novel method, in which we construct tensors $\mathbf{E}^*(\ell, p_\ell)$ such that their values follow the same distribution as the actual error values in $\mathbf{E}(\ell, p_\ell)$. Due to the properties of the XNOR error model, the values in the error tensor follow a normal distribution, i.e. $\mathbf{E}(\ell, p_\ell) \sim \mathcal{N}(\mu, \sigma^2)$. To acquire distributions from which error tensors can be sampled, we first train a BNN with MHL, without application of any error model. We then sample error tensors of $\mathbf{E}(\ell, p)$ for a specified p_ℓ in a specified layer ℓ during inference of a BNN under the exact XNOR error model. We then construct a normal distribution from the obtained samples of values in the error tensors. With the acquired mean μ and standard deviation σ^2 for a combination of ℓ, p_ℓ, we can sample values of error matrices $\mathbf{E}^*(\ell, p)$ during training.

For applying the XNOR error model during training, we add $\mathbf{E}^*(\ell, p)$ to the result of the popcount, and get

$$\text{popcount}(\text{XNOR}(\mathbf{W}_i^\ell, \mathbf{A}^{\ell-1})) + \mathbf{E}^*(\ell, p),$$

which is a precise approximation. This way of applying the XNOR error model has similar error distributions as in the precise XNOR error model. It is also applicable in any training framework, while having low overheads.

Note that values for the mean and standard deviation of the error tensor can only be derived theoretically if the exact number of XNOR operations with result 0 per layer are known. This number would have to obtained by a similar way of sampling as above. For this reason we derive the mean and standard deviation of the error

tensor by sampling the error tensors that are produced when applying the exact XNOR error model on a BNN that was trained without any error model application during training.

5.3.4.2 *Error Tolerant BNNs during Run-Time*

The architecture and NN parameters are often obtained through high efforts and using lots of training resources. Training with errors may not be desired because the parameters will be modified, while injecting errors during training is time consuming or may even be infeasible in some scenarios. Furthermore, using the same p_ℓ for every layer in the BNNs may not be a good option for $\mathbf{p}$, since different layers in NNs exhibit different sensitivities to errors [HLPS20; HWC17].

For these reasons, we develop a layer-wise method for optimizing values in $\mathbf{p}$. One method is to start with high p_ℓ in each layer, and decrease it until a certain accuracy goal is achieved [Ha+21]. In our work, we also have to consider the speedup for setting $\mathbf{p}$. Our goal is to increase the p_ℓ in each layer, such that we achieve the highest speedup under the sacrificed accuracy drop (AD). Additionally, we want to incorporate the number of operations of a layer into the optimization. To achieve this, we define s_{ℓ,p_ℓ} as the XNOR speedup obtained when we divide the baseline (or reference) XNOR latency at $P_{error} = 0$ by the new XNOR latency at $P_{error} > 0$. We then define AD_{ℓ,p_ℓ} as the accuracy drop (AD), when the error model is applied with p_ℓ in layer ℓ only. As the objective for maximization, we denote $\frac{s_{\ell,p}}{AD_{\ell,p}}$. To incorporate the number of operations, we multiply this value by the number of cycles c_ℓ needed to compute a layer ℓ, divided by the maximum number of cycles needed in any layer, denoted as c_{max}. We call this value weighted speed accuracy drop (WSAD):

$$WSAD_{\ell,p_\ell} = \frac{s_{\ell,p_\ell}}{AD_{\ell,p_\ell}} \frac{c_\ell}{c_{max}} \tag{5.1}$$

The higher the WSAD-value, the higher is the speedup for a certain AD, while weighting values of layers with more operations higher. We use the WSAD-values to select p_ℓ for each layer using a greedy strategy.

5.3.5 *Experiment Results*

In the following we present the experiment setup in Sec. 5.3.5.1. We then evaluate the design-time and run-time techniques in Sec. 5.3.5.2 and Sec. 5.3.5.3, respectively.

5.3.5.1 *Experiment Setup*

To evaluate the error tolerance of BNNs against the XNOR errors, we use a framework based on PyTorch, as described in Ch. 3. We also use the BNN models VGG3 and VGG7 (see Table 3.2), which use the stochastic input binarization from Sec. 2.2.3, with the Fashion and CIFAR10 datasets (see Table 3.1), as described in Ch. 3. In

the PyTorch MAC-library, the exact XNOR error model cannot be applied, since the code for matrix-multiplication is based on proprietary CUDA libraries. Due to this it cannot be easily modified. Therefore, to apply the exact error model in our framework, it was necessary to replace the MAC-library in PyTorch with our own customized MAC-library with CUDA kernels.

To describe the obtained speedup of our methods, we divide the latency at $P_{error} = 0$ by the latency at the P_{error} obtained by our technique. The P_{error} can be translated to the corresponding XNOR latency (see Fig. 5.6(d)). For evaluations, we consider the following set of accuracy budgets: $\{0.25, 0.5, 1, 2, 3, 4, 5\}$ percentage points below the baseline BNN accuracy, which is 90.43% for Fashion, and 83.64% for CIFAR10 in the absence of XNOR errors, see black plots in the left column of Fig. 5.8((a) and (d)).

5.3.5.2 *Design-Time Error Tolerance of BNNs*

In Fig. 5.8 (leftmost column), we present the accuracy over P_{error} for Fashion (Fig. 5.8(a)) and CIFAR10 (Fig. 5.8(d)) BNNs trained with our design-time technique presented in Sec. 5.3.4.1. We apply the error models during training using P_{error} with 1%, 2%, 3%, and using P_{error} with 0.25%, 0.5%, 0.75% for the datasets Fashion and CIFAR10, respectively, by which the BNNs become error tolerant. Note that in both dataset cases, 0% indicates the baseline BNN without any errors during training. The achieved error tolerance allows to trade off latency with accuracy.

With the error tolerant BNNs we can evaluate the LiM speedup under an accuracy budget. The speedup values are derived by utilizing the baseline (0%) and design-time method (1-3% for Fashion or 0.25-0.75% for CIFAR10) from the plots in Fig. 5.8((a) and (d)) for a certain accuracy budget by finding the probability of error at which the average accuracy is high enough such that the accuracy drop is within the accuracy budget. We then convert the probability of error to a speedup value in Fig. 5.6 (see rightmost column in Fig. 5.8, (c) and (f), for the results). For example, in the leftmost column, consider the black plot in Fig. 5.8(a). With an accuracy budget of 1%, the average accuracy is 89.53% at 1% probability of error. Then, the 1% probability of error is translated to 0.495 ns. Finally, we acquire a speedup of 43% by dividing the baseline latency 0.706 ns by 0.495 ns.

In the rightmost column of Fig. 5.8, (c) and (f), we observe that the higher the accuracy budget, the higher the speedups, but the returns become small, e.g. above 1% budget. For the achieved speedups based on this method, for Fashion Fig. 5.8(c), we observe that for the small accuracy budgets 0.25%, 0.5% and 1%, we achieve speedups of 43%, 53%, and 63% respectively for 1% P_{error} during training and speedups of 67%, 68%, 75% respectively for 2% P_{error} during training. For CIFAR10 in Fig. 5.8(f), we observe that for the budgets 0.25%, 0.5% and 1%, we achieve speedups of 26%, 30%, and 36% respectively for 0.25% P_{error} during training and speedups of 33%, 36%, 38% respectively for 0.5% P_{error} during training.

Importantly, for the design-time method, the higher P_{error} is during training, the more the BNNs exhibit adaptation to the P_{error}. In Fig. 5.8(a) the 3%-curve has an

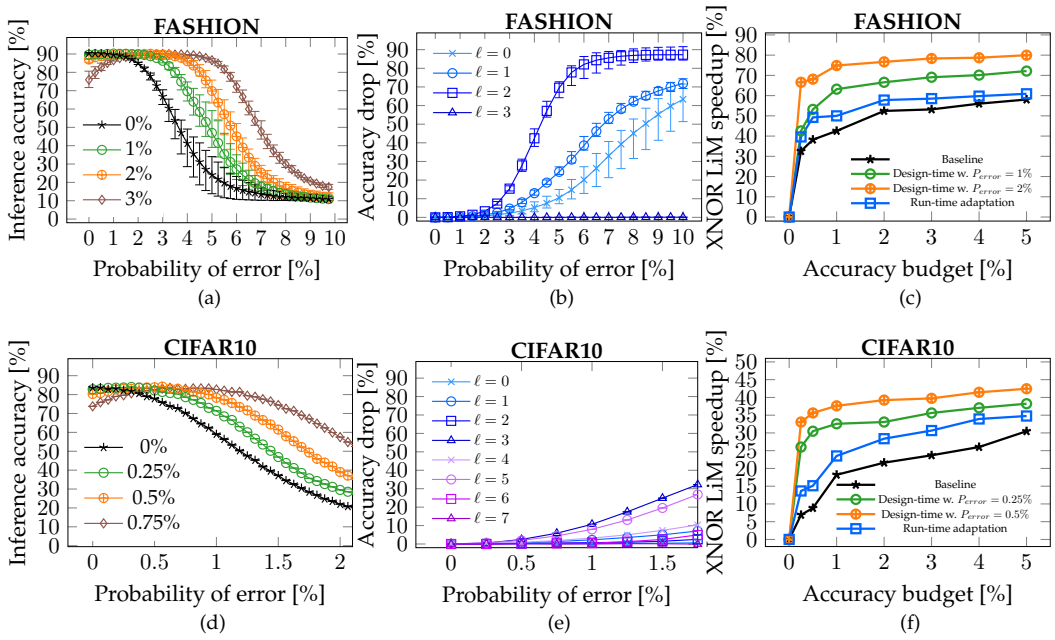


Figure 5.8: Experiment results for the FeFET-based XNOR LiM speedup with the datasets Fashion and CIFAR10. Left column ((a) and (d)): Accuracy over probability of error. Middle column ((b) and (e)): Accuracy drop (AD) over probability of error. Right column ((c) and (f)): XNOR LiM speedup values over accuracy budgets.

average accuracy of 75% for P_{error} of 0%. In Fig. 5.8(d), the 0.75%-curve has an accuracy of 73.78% at P_{error} of 0%. This makes these BNNs inflexible and impractical. In such a case, BNNs cannot be used for inference in the absence of errors. The reason for the adaptation is that the errors manifest themselves as positive terms added to the popcount result, because the XNOR errors only flip "0" to "1" with a certain probability of error, and never "1" to "0", meaning the result of the popcount can only increase, as explained in Sec. 5.3.4.1. Note that we also observe a small adaptation effect for BNNs trained with lower P_{error} during training. In other studies on BNN inference with bit errors, e.g. on RRAM [Hir+19b], the equation for computations is $\text{popcount}(\text{XNOR}(\mathbf{W}^{\ell}, \mathbf{A}^{\ell-1})) \pm \mathbf{E}(\ell, p_{\ell})$, since there the assumption is that the binary weights can flip from "1" to "0" or vice versa with the same probability of error, which may cause less severe adaptations.

5.3.5.3 *Run-Time Error Tolerant BNNs*

To use WSAD-values from Eq. (5.1) in Sec. 5.3.4.2 for setting p_{ℓ}, it is first necessary to evaluate the accuracy drop (AD) per layer. We apply the exact XNOR error model in a single layer and then evaluate the inference accuracy on the training set. The AD-values per layer are shown in the middle column in Fig. 5.8((b) and (e)). Based on

the AD-values, we calculate the WSAD-values. The number of cycles per layer (c_ℓ), needed for Eq. (5.1), are obtained from the MAESTRO tool [Kwo+20].

We find the setting of $\mathbf{p}$ using a greedy approach. In every step of the algorithm, we select the highest remaining WSAD value and after every step evaluate whether we retain our accuracy budget. For example, for Fashion, one setting we achieve by this for 1% accuracy budget is $\mathbf{p} = \{p_0 = 0.5\%,\ p_1 = 1\%,\ p_2 = 1.5\%,\ p_3 = 0\%\}$, with an average accuracy of 89.53%. The next step in the algorithm would be to increase p_1 to 1.5%, as its WSAD value is the highest remaining value. For CIFAR10, with an accuracy budget of 1%, the setting $\{p_0 = 1.5, p_1 = 0.5, p_2 = 0.5, p_3 = 0.25, p_4 = 0\%, p_5 = 0\%, p_6 = 0.25, p_7 = 0\%\}$ leads to an average accuracy of 83.18%. The other settings have higher or lower error rates depending on the accuracy budget. We acquire the speedup values from $\mathbf{p}$ by multiplying the number of computing cycles of a layer by the corresponding latency at p_ℓ. We perform this for each layer and then compute the sum over all layers. Afterwards, we divide the baseline latency (0.706 multiplied by the number of cycles for a BNN) by the computed sum and report the speedup in Fig. 5.8((c) and (f)), in the rightmost column in the black plots.

Our run-time method outperforms the baseline for all considered accuracy budgets in both datasets. For example, for Fashion, with the budgets 0.25%, 0.5%, and 1%, we observe speedups of 40%, 49%, 50%, respectively. For CIFAR10, with the budgets 0.25%, 0.5%, and 1%, we observe speedups of 14%, 15%, 24%, respectively. However, in some cases, the speedup values can deviate from the trend. This is due to the stochastic nature of the input binarization and error injection. In comparison to the design-time methods, the run-time method achieves lower speedups. Note that the run-time method is more flexible, as it does not need any retraining and does not suffer from accuracy loss at 0% probability of error. The run-time method can also be applied in cases in which the model is given and cannot be retrained.

5.4 CONCLUSION

In this chapter, we explored the use of FeFET as an approximate memory for BNNs. FeFET is one of the most promising emerging technologies to date. Due to its CMOS compatibility, it is expected to be integrated into products in the near future. FeFET is an emerging technology, therefore some aspects of it need further attention. Here, we focused on two aspects: The temperature susceptability of FeFET and the latency of FeFET-based LiM. The sections addressing the two aspects are summarized below.

In Sec. 5.2, we first analyzed the effects of variable temperature on FeFET memory and proposed an asymmetric bit error model that exhibits the relation between temperature and bit error rates. We then evaluated the impact of FeFET asymmetric temperature bit errors on BNN accuracy when no countermeasures are applied and showed that the accuracy can drop unacceptably. To deploy BNNs with high accuracy using FeFET memory despite the temperature effects, we proposed two countermeasures to the bit errors: (1) Bit flip training while taking the asymmetry into account and (2) a bit error rate assignment algorithm (BERA) which estimates

accuracy drops per layer and assigns layer-wise the bit error rate configuration with the lowest accuracy drop. With these methods, the BNNs achieve bit error tolerance for the entire range of operating temperature and FeFET memory can be used on the low-power edge for BNNs despite the temperature-dependent bit errors.

In Sec. 5.3, we investigated how the error tolerance of BNNs can be exploited for decreasing the FeFET-based XNOR-LiM latency, which is a major bottleneck, at the cost of errors. To perform this investigation, we employed the tradeoff between XNOR latency and the extent of the errors occurring in the XNOR-LiM. We proposed two methods to increase the latency: First, by employing design-time training with errors and secondly, with a run-time approach that does not require training with the errors. Both methods lead to significant LiM speedups.

6 HW/SW CODESIGN FOR EFFICIENT BNN INFERENCE

In the vision of this dissertation, we propose to use approximate computing units in systems that run BNNs. In this chapter, we present HW/SW Codesign methods that exploit the error tolerance of BNNs in approximate analog-computing units for inference efficiency. In HW/SW Codesign, we concurrently design the SW (here BNNs) and the hardware (here HW for accelerating the operations of BNNs) with the aim of finding synergistic solutions that lead to high efficiency (here in the domain of analog computing).

Specifically, in Sec. 6.1, we propose a method on the SW level to approximate the global thresholdings in BNNs with local thresholdings, which leads to highly efficient interface circuit designs in analog computing based crossbar accelerators for BNNs. In Sec. 6.2, we exploit the redundancy of quantization levels in the SW of BNNs to reduce the required capacitor size in analog computing based Integrate and Fire Spiking NNs (IF-SNNs) circuits.

The methods in both sections lead to measurable reductions in area, energy, and latency, leading to highly efficient systems that execute BNNs. We present them in the following.

6.1 GLOBAL BY LOCAL THRESHOLDING IN BNNS FOR EFFICIENT CROSSBAR ACCELERATOR DESIGN

For processing the workloads of BNNs efficiently, analog computing based crossbar accelerators have recently been proposed in [Che+18; Sol+20b]. Crossbars are organized in columns, where each column consist of XNOR gates, circuits to perform the popcount (e.g. through Kirchhoff's circuit law in analog computing), and *interface circuits*, shown in Fig. 2.5. The interface circuits employ analog-to-digital conversion and accumulate intermediate results for further processing. This necessitates the use of analog-to-digital converters (ADCs), registers connected to adders for accumulation, and digital comparators. Specifically, ADCs are one of the most critical building blocks in crossbar accelerators. It has been reported that in crossbar accelerators, the ADCs use the most on-chip area and energy. For example, in the ISAAC accelerator, ADCs use a large portion of the tile power and tile area [Sha+16b].

To avoid using the ADCs and digital components, the classical approach [Che+18] is to use an "analog path" (AP) in the interface circuit, which only uses an analog comparator. In contrast, the "digital path" (DP) uses ADCs and other digital components. The AP and DP are shown in Fig. 2.5(c). However, the AP of the interface circuit is and can only be used for a very small number of weights due to technological limitations (e.g. up to 64 in [Che+18]), meaning it is *not used* in common BNN

architectures (which generally require a large number of weights β per neuron, see e.g. Table 6.3). Therefore, the DP, with the ADCs and digital components, is used in every crossbar invocation in BNN inference, causing high energy usage and high latency. Furthermore, each column in the crossbar needs one interface circuit, causing high area usage. For example, with 64 crossbar columns, 64 interface circuits with both APs and DPs are needed.

Without the DP of the interface circuits, only analog comparators would be needed for the BNN operations. The elimination of the DP would lead to breakthroughs in small area usage, low energy consumption, and low latency inference in BNN accelerators.

However, without the DP, the thresholdings in BNNs cannot be performed with the full result of the popcount, referred to as "global thresholding". Instead, when using only the AP, the global thresholdings have to be approximated by "local thresholdings" using the local popcounts in each crossbar column. To enable this, thresholds for the local thresholdings have to be derived, while the results of the local thresholdings have to be combined to reach an approximate global solution. Despite the high error tolerance of BNNs, this way of approximation may cause high accuracy drop, which necessitates the evaluation of the tradeoffs between interface circuit efficiency and BNN inference accuracy.

In this section, we present a novel computing scheme for BNNs, which approximates the global thresholdings by combining the results of local thresholdings, called local thresholding approximation (LTA). To tolerate the approximations of LTA, we propose to train the BNNs with approximations. We propose an efficient interface circuit design for the LTA and show that it needs less resources than the state of the art. Since LTA is a novel computing scheme, we propose tailored BNN workload mapping strategies. In the experiments we reveal the impact of using the LTA in BNNs and show that using the LTA may lead to significant accuracy degradation if no countermeasures are employed. We then demonstrate that when training with the approximations from the LTA, we consistently achieve high accuracy even under noise.

The remainder of this section is structured as follows: In Sec. 6.1.1, we present the problem definition. We introduce the LTA in Sec. 6.1.2 and the LTA training scheme in Sec. 6.1.3. We discuss the hardware concepts required for realizing the LTA , the dataflow, crossbar, interface circuit, and the workload mapping in Sec. 6.1.4. The experiments are in Sec. 6.1.5.

6.1.1 *Problem Definition*

In the classical approach by Chen et al. [Che+18], one crossbar column has one interface circuit, each consisting of an analog comparator, an ADC, an adder, registers, and a digital comparator, illustrated in Fig. 2.5. In their design, m interface circuits are needed for a crossbar of size $(m \times n)$. To avoid using the ADCs and other digital components, their approach employs an "analog path" (AP) in the interface circuit,

which only uses analog comparators. In contrast, the "digital path" (DP) uses ADCs and other digital components.

However, the AP is only used when the number of weights β per neuron is very small, i.e. $\beta \leq n$, which is rarely the case (in [Che+18], n is only 64). High-performing BNN architectures actually use a large β, and the size of β of a layer depends on the number of neurons in the previous layer. For example, in our VGG-based BNNs, β is large in every case, as shown in Table 6.3. Therefore, the DP is used in every crossbar invocation, causing high energy usage and high latency. Further, the interface circuit is the most area demanding part of the crossbar accelerator, mainly due to the ADCs and registers, while the other components also cannot be neglected.

Problem Definition: Given a trained BNN with high accuracy and a crossbar accelerator, as described in Sec. 2.3.2.2, in this section, we focus on the problem of reducing the complexity of interface circuits in BNN crossbar accelerators, to achieve inference using small area, low energy, and low latency.

Next, we present our novel method, which enables us to use the AP through most of the execution (i.e. for $\beta \leq mn$) and needs only one interface circuit for all crossbar columns (instead of m interface circuits in the SOTA), at the cost of approximations.

6.1.2 *LTA Execution*

For simplicity of presentation, we consider that the weights and activations are binary values in $\{-1, 1\}$. This way, XNOR can be denoted as multiplication and popcount as summation. We rely on the notation in Sec. 2.1.1 and, to be able to distinguish weights from the activations, in this chapter we use $\mathbf{X}$ for the inputs to a layer. The weights of one neuron (a certain row in $\mathbf{W}$) are described as $W = (w_1, w_2, \ldots, w_\beta)$, with $w_j \in \{-1, 1\}$ and with β as the number of weights. The input (a column in $\mathbf{X}$) is denoted as $X = (x_0, x_1, \ldots, x_\beta)$ with $x_j \in \{-1, 1\}$. We assume that the layers have the following structure, without any operations inbetween: Convolution, batch norm, binary activation. With this assumptions in BNNs, the activations are computed by

$$a = \mathbf{1}[\sum_{i=1}^{\beta} w_i x_i \geq T]. \tag{6.1}$$

T is the threshold for the neuron in W and $\mathbf{1}[predicate]$ is a modified Iverson bracket, which returns 1 if the condition *predicate* holds and -1 otherwise. The computation in Eq. (6.1) is precise, i.e. without any errors. To perform precise computations, m interface circuits with APs and DPs are needed, as described in Sec. 6.1.1.

To use the AP and interface circuits with less complexity, we employ local thresholdings using subsequent samples of size n, and combine the results of local thresholdings with a majority function to obtain an approximate global result. By this, the computation result of a crossbar column with n XNOR gates is represented by one binary value. The majority vote of m binary values from all crossbar columns is then performed to reach a final approximate decision. This means that local thresholdings are performed to reach an approximate global thresholding result.

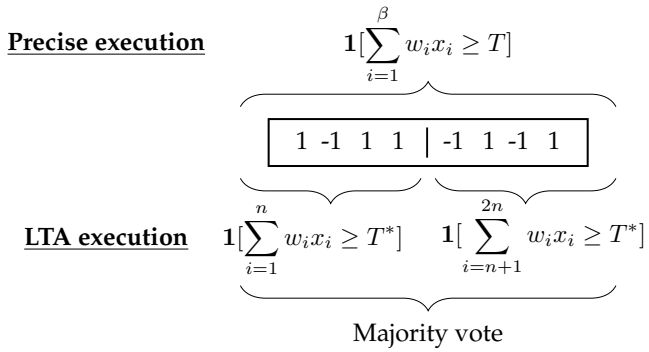


Figure 6.1: Precise and LTA execution in BNNs for $\beta = 8$, $n = 4$.

We call this way of computing the local thresholding approximation (LTA). In the LTA, the AP is triggered throughout most of the execution, i.e. for $\beta \leq mn$. Furthermore, only one interface circuit is needed for the crossbar, whereas the SOTA by Chen et al. [Che+18] requires m interface circuits. We focus on the interface circuit and the corresponding workload mapping in Sec. 6.1.4, with a comparison to the state of the art.

For the LTA, local thresholdings in the form $\mathbf{1}[\sum_{i=1}^{n} w_i x_i \geq T^*]$ are performed, i.e. the sums are computed up to a value n, which is the number of XNOR gates in a column of a crossbar. The local thresholdings are performed with a local threshold T^*, followed by the majority vote of all local thresholdings.

The LTA is defined as:

$$\begin{aligned}\mathbf{1}[\sum_{i=1}^{\beta} w_i x_i \geq T] &\approx \langle a_1, a_2, \ldots, a_N \rangle \\ &= \langle \mathbf{1}[\sum_{i=1}^{n} w_i x_i \geq T^*], \mathbf{1}[\sum_{i=n+1}^{2n} w_i x_i \geq T^*], \ldots, \mathbf{1}[\sum_{i=(N-1)n+1}^{\beta} w_i x_i \geq T^*_{last}] \rangle.\end{aligned} \tag{6.2}$$

The majority is denoted as $\langle a_1, \ldots, a_N \rangle = Majority(a_1, \ldots, a_N)$. The threshold T^* for the local thresholdings, except the last, which is acquired by dividing the global T by the number of local comparisons $N = \left\lceil \frac{\beta}{n} \right\rceil$. The threshold T^*_{last} for the last window (which may have smaller window size than other windows) is derived by scaling T^* according to the size of the rest $\beta - (N-1) \cdot n$. The formulas for deriving the thresholds in Eq. (6.2) are

$$T^* = round\left(\frac{T}{N}\right), \quad T^*_{last} = round\left(T^*\left(\frac{\beta}{n} - (N-1)\right)\right), \tag{6.3}$$

where the *round* function rounds to the nearest integer and ties are broken by rounding up. An example for the LTA with $n = 4$, $\beta = 8$, and $N = 2$ is shown in Fig. 6.1.

Algorithm 5: Forward pass for LTA training

Input: *model*, $\mathbf{X}$
Output: $\mathbf{X}$

1 **for** *each binarized layer* **do**
2 $\quad \mathbf{X}_{copy} \leftarrow \mathbf{X}.clone().detach()$
$\quad$ `// Regular execution`
3 $\quad execution(\mathbf{X}, \text{regular})$, according to Eq. (6.1)
$\quad$ `// LTA execution`
4 $\quad$ **for** *every* $neuron_j$ *for* $j = 1, 2, \ldots, \alpha$ **do**
5 $\quad\quad$ derive μ_j,σ_j, ψ_j, and η_j, see Sec. 2.2.2.1, derived from Line 3;
6 $\quad\quad T_j \leftarrow \mu_j - \frac{\sigma_j}{\psi_j}\eta_j$;
7 $\quad execution(\mathbf{X}, \text{LTA})$, according to Eq. (6.2) (applied for each neuron);
8 $\quad \mathbf{X}.data \leftarrow \mathbf{X}_{copy}.data$
9 return $\mathbf{X}$

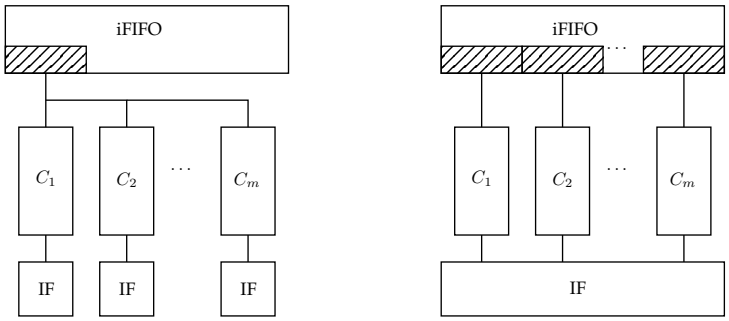
6.1.3 *Training with LTA*

One method is to use the inherent error tolerance of BNNs to tolerate the approximations stemming from the LTA. However, the approximations may depend on the configurations. To make BNNs error tolerant, applying errors during the training has been suggested in the literature [Hir+19b].

We use the idea of error application during training to make the BNNs tolerant to the approximations of the LTA. To this end, we replace the correct activations by the approximate activations in the forward pass of the training procedure.

Our method for training with the LTA is illustrated in Alg. 5. We refer to the functions of BNNs that perform the operations of a layer (both 1D and 2D) as *execution*. During inference, the BNN layers are first executed in the regular way (i.e. convolution, batch norm, htanh, activation, Line 3), according to Eq. (6.1). After the regular execution is finished, the thresholds for each neuron in the layer are extracted (Line 6), from which the thresholds for the windows in the LTA can be computed, see Eq. (6.3). Then the BNN operations are performed with the LTA according to Eq. (6.2) (Line 7). After the LTA, the tensor of the correct activations is replaced with the tensor of the (approximate) LTA activations (Line 8).

In this method, the LTA operations are removed from the computation graph. They are not considered during the backpropagation, due to the copy and detach in Line 2. Furthermore, only the values of $\mathbf{X}$ are replaced by the approximated values $\mathbf{X}_{copy}$, see Line 8. This means that the training is performed with the approximations and that the weights and batch norm parameters of the layer in Line 3 are adapted in the backpropagation based on the LTA.



(a) Input data flow for baseline. (b) Input data flow for LTA.

Figure 6.2: Comparison of input data flow between the (a) baseline and (b) the LTA execution. iFIFO: FIFO for input data. $C_1, \ldots C_m$: Crossbar columns. IF: Interface circuit.

6.1.4 *Dataflow, Interface Circuit, Workload mapping*

In the following we present the dataflow in Sec. 6.1.4.1, the interface circuit in Sec. 6.1.4.2, the crossbar in Sec. 6.1.4.3, and the workload mapping in Sec. 6.1.4.4.

6.1.4.1 *Dataflow*

Our LTA approach requires a different way to apply inputs (i.e. types of input data flows) compared to the baseline in [Che+18]. In the following, we consider two ways of input data flow to an accelerator with sub-components, such as crossbar columns in BNN accelerators in our case. The two types of input data flows have been explained by Kung in 1982 [Kun82], in the context of systolic arrays: (1) One input is broadcasted to all columns (e.g. see design "B1" in [Kun82]), which is a well-known method in modern NN accelerators and is also applied in [Che+18]. (2) Multiple (different) inputs are applied to different columns, e.g. the first input is applied to the first column, the second input to the second column, etc. (see the designs "F" in [Kun82]). Our LTA method requires the second type of input data flow.

Fig. 6.2 illustrates the input data flow type required by the LTA execution and the input data flow required by the baseline for comparison. To move the input data to the accelerator, FIFOs are used, which are called iFIFOs here. The system architecture regarding the FIFOs is inspired by [CES16]. In the baseline case in Fig. 6.2(a), one input (striped area) is accessed from the iFIFO. Then, the input is broadcasted to all columns. In the LTA case in Fig. 6.2(b), m different inputs (striped areas in the iFIFO) are accessed from the iFIFO. Then, the inputs are moved into the crossbar columns.

In both cases in Fig. 6.2(a) and in Fig. 6.2(b), techniques and corresponding circuits need to be used to move the data into the crossbar columns, while considering the design tradeoffs. For example, for both cases, the inputs can be moved into the crossbar array in an iterating manner over all columns and inputs can be applied to multiple columns in parallel. The difference between (a) and (b) with respect to the iFIFOs is that in (a) a single input needs to be accessed and supplied to the columns,

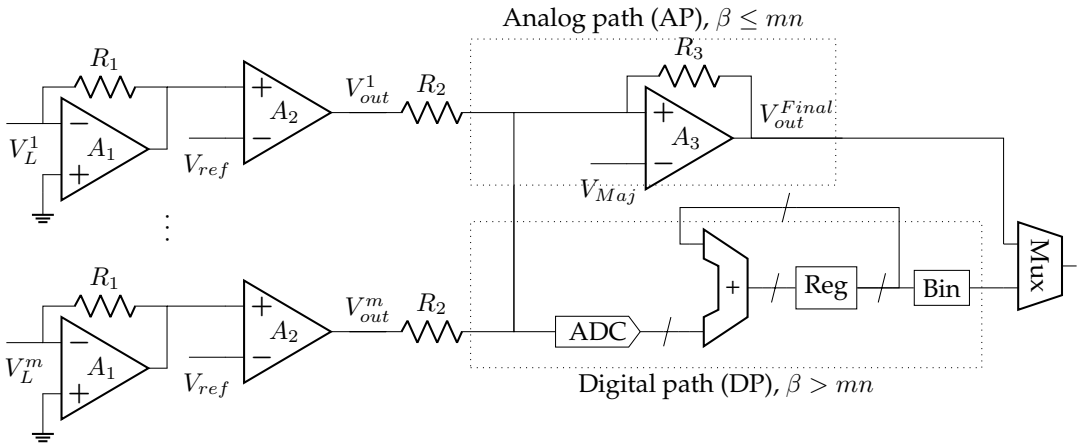


Figure 6.3: Our interface circuit design for our proposed LTA method. The voltages V_L^1 and V_L^m in are input voltages from Fig. 2.5(a).

while in (b), m inputs need to be accessed. If inputs are applied to multiple columns in parallel, then in the case of (a), the iFIFO needs one output port (since one input is accessed), while in (b) multiple output ports are needed (since multiple different inputs are accessed).

6.1.4.2 *Interface Circuit*

Our interface circuit design for an entire crossbar with m columns and n XNOR gates computing with the LTA is shown in Fig. 6.3. See also Fig. 6.2 for the overall architecture, i.e. where the interface circuits are and that m interface circuits are needed in the baseline, while only one interface circuit is needed for the entire crossbar for the LTA. The interface circuit in Fig. 6.3 has m incoming summed currents from m crossbar columns. These currents are converted into voltages (e.g. V_L^1) by the resistors R_1 and then amplified with the operational amplifiers (opamps) A_1. The resulting voltages are compared to thresholds in the analog comparators A_2. The reference voltages V_{ref} are derived by dividing the original threshold, see Eq. (6.3). The outputs of A_2 are voltage levels, which are converted to currents by the resistors R_2. In the analog path (AP) of the circuit, these currents are summed by Kirchhoff's circuit law, converted back to a voltage by R_3, and then a majority vote is performed with the analog comparator A_3, by which the final output V_{out}^{final} is obtained.

The AP is employed in our LTA interface circuit design when the workload has size $\beta \leq mn$. If the workload is larger than that, i.e. $\beta > mn$, then the digital path (DP) is used. In the DP, the sum of currents from the opamps A_2 are converted into the digital domain using an ADC. Then, accumulations are performed, whose results are stored in registers. When the accumulation is finished, a binary comparison is performed and the final result is obtained (details are described in Sec. 6.1.4.4).

Note that, for the LTA, the AP is used when $\beta \leq mn$ and only *one* interface circuit for the entire crossbar is needed. In comparison, the AP is used only when $\beta \leq n$ in the SOTA [Che+18], and m interface circuits are needed for the entire crossbar, which all need an ADC, an accumulator, registers, and binarization logic. The specifications of the interface circuit for the LTA and a comparison to the state of the art are summarized in Table 6.1, and the table for explaining the notations is in Table 6.2.

6.1.4.3 *Crossbar for Inference*

The crossbar computes the matrix multiplication $\mathbf{O} = \mathbf{W} \times \mathbf{X}$. The computation can be separated into two stages, programming and application. In the programming stage, the weights of neurons are programmed into the crossbar. In the application stage, the inputs are applied to the crossbar and the results of the matrix multiplication are returned.

The work in [Che+18] applies a tile-based approach using a strided workload mapping scheme, which minimizes the number of reprogammings – an important issue in NVM-based crossbars. In their approach, the crossbar is programmed with a weight tile of n weights from m neurons, where each neuron occupies one column of the crossbar, as shown in Fig. 6.4(a) in $\mathbf{W}$. Note that the tile may not consist of all weights of the neurons. Then, parts of the columns of $\mathbf{X}$ of size n, which are the corresponding inputs for the programmed weights, are pushed to the crossbar, such that all inputs that are possible to be processed with the programmed weights are processed. This is performed so that the programmed weights are reused without reprogramming. After all the input columns (arrow in $\mathbf{X}$ in Fig. 6.4(a)) are finished for the loaded weights of the current tile, the weights of the next tile are programmed into the crossbar (arrow in the $\mathbf{W}$ matrix for SOTA). This process is repeated over the columns of $\mathbf{X}$.

However, this mapping scheme rarely uses the AP, since usually $\beta \gg n$, resulting in heavy use of the DP with a high cost to digitalize and buffer many intermediate results. For these operations, m DPs (for m crossbar columns) are employed, using m ADCs, $m\delta$ registers, and m other digital components with a large number of bits.

6.1.4.4 *Workload Mapping for LTA*

Consider now that a crossbar is given with the interface circuit in Sec. 6.1.4.3. In the LTA, the weights of one neuron in $\mathbf{W}$ (one row in $\mathbf{W}$) are programmed into the crossbar, as shown with mn for the loaded weights in Fig. 6.4(b). The weights are partitioned to the columns of the crossbar as described in Eq. (6.2). Each column of the crossbar is mapped to one window of Eq. (6.2). After programming, a column of $\mathbf{X}$ is pushed to the crossbar as input, as shown in Fig. 6.4(b). The crossbar is invoked and the majority vote of all m local thresholdings is performed. This is repeated with the same weights until all the columns in $\mathbf{X}$ are finished (arrow in $\mathbf{X}$ in Fig. 6.4(b)). When all columns in $\mathbf{X}$ are processed, the same procedure is repeated for the next

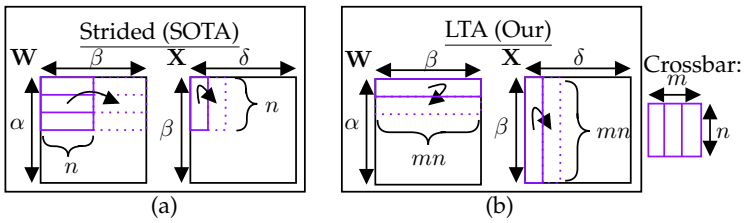


Figure 6.4: Mapping schemes. (a): SOTA [Che+18]. (b): LTA.

Specification	This work (LTA)	State of the art (SOTA) [Che+18]
ADC resolution	$\lfloor \log_2(m) \rfloor + 1$	$\lfloor \log_2(n) \rfloor + 1$
Width digital path	$\left\lfloor \log_2(m \left\lceil \frac{\beta}{mn} \right\rceil) \right\rfloor + 1$	$\left\lfloor \log_2(n \left\lceil \frac{\beta}{n} \right\rceil) \right\rfloor + 1$
Registers	δ	$m\delta$
Crossbar invocations	$I_{cb}^{lta} = \delta\alpha \left\lceil \frac{\beta}{mn} \right\rceil$	$I_{cb}^{sota} = \delta \left\lceil \frac{\alpha}{m} \right\rceil \left\lceil \frac{\beta}{n} \right\rceil$
Area	$A_{cb} + (m+1)A_{acomp} + A_{adc} + A_{add}^{lta} + \delta A_{reg}^{lta} + A_{bin}^{lta}$ For $\beta \leq mn$: $A_{cb} + (m+1)A_{acomp}$	$A_{cb} + m(A_{acomp} + A_{adc} + A_{add}^{sota} + \delta A_{reg}^{sota} + A_{bin}^{sota})$ For $\beta \leq n$: $A_{cb} + mA_{acomp}$
Energy	$I_{cb}^{lta}(E_{cb}^{lta} + mE_{acomp} + E_{adc} + E_{add}^{lta} + E_{reg}^{lta}) + \alpha\delta E_{bin}^{lta}$ For $\beta \leq mn$: $I_{cb}^{lta}E_{cb} + (m+1)I_{cb}^{lta}E_{acomp}$	$I_{cb}^{sota}E_{cb}^{sota} + I_{cb}^{sota}m(E_{adc} + E_{add}^{sota} + E_{reg}^{sota}) + \left\lceil \frac{\alpha}{m} \right\rceil m\delta E_{bin}^{sota}$ For $\beta \leq n$: $I_{cb}^{sota}E_{cb} + mI_{cb}^{sota}E_{acomp}$
Latency	$I_{cb}^{lta}(L_{cb} + L_{acomp} + L_{adc} + L_{add}^{lta} + L_{reg}^{lta}) + \alpha\delta L_{bin}^{lta}$ For $\beta \leq mn$: $I_{cb}^{lta}L_{cb} + 2I_{cb}^{lta}L_{acomp}$	$I_{cb}^{sota}(L_{cb} + L_{adc} + L_{add}^{sota} + L_{reg}^{sota}) + \left\lceil \frac{\alpha}{m} \right\rceil \delta L_{bin}^{sota}$ For $\beta \leq n$: $I_{cb}^{sota}L_{cb} + I_{cb}^{sota}L_{acomp}$

Table 6.1: Crossbar interface circuit comparison between LTA (this work) and the state of the art (SOTA) in [Che+18], for a crossbar size of m columns and n XNOR gates per column. The notations are described in Table 6.2. The formulas for the SOTA in [Che+18] can acquired by the following substitutions: $\alpha = C_{out}$, $\beta = W_F H_F C_{in}$, $\delta = W_O H_O$, $S = \left\lceil \frac{\beta}{n} \right\rceil$, $m = N$, and $n = M$.

neuron (arrow in **W** in Fig. 6.4(b)). Only the AP is applied when the weights of one neuron fit into one crossbar, i.e. when $\beta \leq mn$.

When $\beta > mn$, the DP needs to be used to digitalize and buffer the intermediate results, since the weights of one neuron do not fit into the entire crossbar. When using the DP, the crossbar is first programmed with mn weights of one neuron in **W**. Then the inputs that need to be processed with the programmed nm weights are supplied to the crossbar. The sum of m currents from m analog comparators is converted to the digital domain using the ADC. The results are accumulated in a designated register for the column of **X**. This is repeated until all columns of **X** are processed. Then, the next nm weights of one neuron are processed. After all operations for a neuron are finished, the values in the registers are binarized. The same procedure is repeated for the subsequent neurons.

In Table 6.1, we summarize the equations for the interface circuit properties and equations regarding area, energy, and latency. As mentioned above, the intermediate storage of values is only necessary, if the crossbar is too small to hold all the values ($\beta > mn$). In this case, the number of registers needed for the interface circuit

Variable	Definition
m	Number of columns in a crossbar
n	Number of XNOR gates per column
α	Number of neurons in a layer
β	Number of weights (of neurons) in a layer
δ	Second dimension of the input matrix
I_{cb}	Number of crossbar invocations
A, E, L	Area, energy, and latency of a component, respectively
lta	Local thresholding approximation
$sota$	State of the art
cb	Crossbar
$acomp$	Analog comparator
adc	Analog-to-digital converter
reg	Register
bin	Digital binarizer
add	Digital accumulator

Table 6.2: Notation for Table 6.1.

of the crossbar in LTA is δ. The ADC needs a resolution of $\lfloor \log_2(m) \rfloor + 1$. The number of bits needed in the registers, accumulator, and binarization component are $\lfloor \log_2(m) \rfloor + \left\lceil \frac{\beta}{mn} \right\rceil$.

Note that, when β is small compared to the crossbar size, i.e. $\beta \ll mn$, the LTA mapping above may not fully utilize the crossbar in one invocation, since one neuron always occupies the entire crossbar. For example, with 576 weights per neuron, and 4096 XNOR gates in total, the crossbar utilization is merely around 14.1% (and seven neurons could be computed in parallel). This may lead to a long latency, since the number of crossbar invocations is larger than the SOTA. To alleviate this, when $\beta \leq \frac{mn}{2}$, multiple neurons can be mapped to one crossbar, enabling parallel computation, which increases the crossbar utilization. We call this extension LTA maximum utilization (LTA-MU). Compared to LTA, LTA-MU additionally divides the number of crossbar invocations I_{cb}^{lta} in Table 6.1 by the factor $f = \lfloor \frac{mn}{\beta} \rfloor$, which means f neurons are processed in parallel in one crossbar. However, when LTA-MU is used, the number of interface circuits with analog comparators in Table 6.1 needs to be multiplied by f. These tradeoffs are evaluated and compared to the SOTA in Sec. 6.1.5.3. Note that when $\beta \geq \frac{mn}{2}$, LTA-MU cannot be used, since multiple neurons cannot be computed in parallel in this case. We note that using LTA-MU instead of LTA never influences the inference accuracy, because the computation are the same. The only difference is that LTA-MU exploits parallelism.

6.1.5 *Experiments*

In this subsection, we compare our LTA approach to the state of the art (SOTA). The structure is as follows. We discuss the experiment setup in Sec. 6.1.5.1. In Sec. 6.1.5.2, we evaluate the accuracy tradeoff in our proposed LTA method. In Sec. 6.1.5.3, we evaluate the area, energy, and latency benefits of our LTA. Since there can be different types of noise in the circuit when analog computing is used, we perform a general noise analysis and how it can be overcome by training with the noise together with the LTA in Sec. 6.1.5.4. In Sec. 6.1.5.5, we show the feasibility of the input data flow. We discuss other methods we explored for finding local thresholds in Sec. 6.1.5.6, then we discuss other BNN models Sec. 6.1.5.7, and provide a discussion of ADC modifications in Sec. 6.1.5.8.

6.1.5.1 *Experiment Setup*

To evaluate the LTA, it is required to implement it in practice. However, it cannot be easily implemented in deep learning frameworks, as it is a profoundly different computing scheme compared to standard way of inference. For instance, the well-known framework PyTorch [Pas+19] uses a proprietary MAC engine, which cannot be easily modified. To still be able to execute the BNNs with the LTA in PyTorch, we developed our framework (`https://github.com/myay/LTA-BNN`) using a custom MAC library, with our own custom CUDA extensions. The calls to nn.linear and nn.conv2D layers in PyTorch are redirected to our custom CUDA kernels, which implement the local thresholding and majority voting according to the LTA scheme for our BNNs.

We use the VGG3 and VGG7 BNNs from the experiment setting in Ch. 3 and use the MHL from Ch. 4 with $b = 128$ as a loss for optimization in all cases. We halve the learning rate every 10th epoch for Kuzujishi, Fashion, SVHN, and halve it every 50th epoch for CIFAR10 and Imagenette.

When we train with the LTA, i.e. apply Alg. 5, we always train from scratch with the LTA execution. This means we always start a completely new training process when we train with the LTA. This includes the cases in which we train with a certain number of XNOR gates (see Sec. 6.1.5.2) and the cases in which we perform the noise analysis (see Sec. 6.1.5.4).

6.1.5.2 *Accuracy Tradeoff in LTA*

In Fig. 6.5, we show the accuracy tradeoff for the LTA. We use Eq. (6.3) for acquiring the local thresholds from the global thresholds and in the inference we apply the LTA computation scheme as shown in Eq. (6.2). In the black line, the accuracy under the LTA without any countermeasures is shown. For all datasets, we observe that with an increase of the number of XNOR gates (n), the accuracy increases. However, the accuracies fluctuate and the accuracy drops can be large, compared to the original

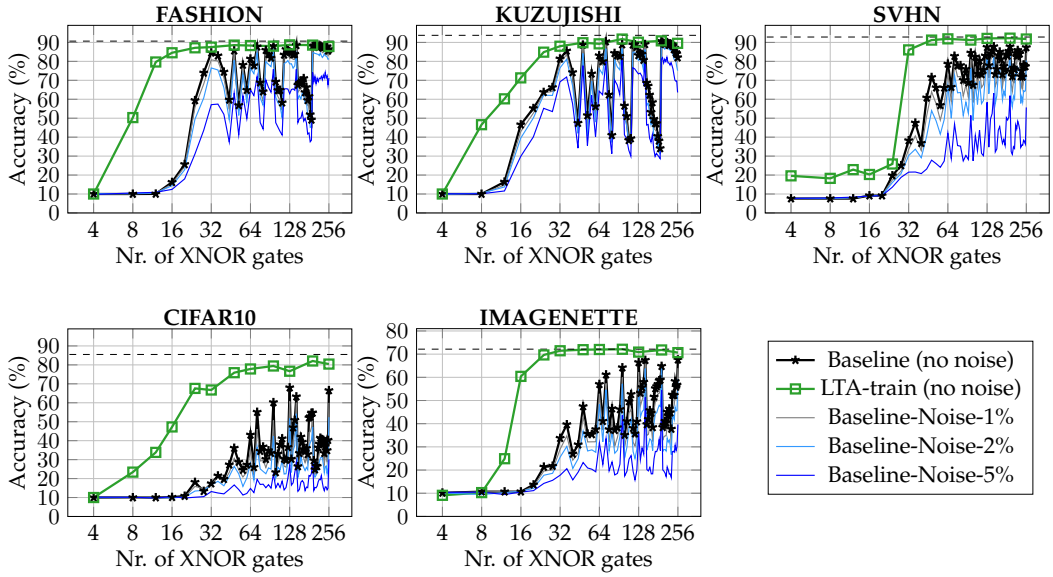
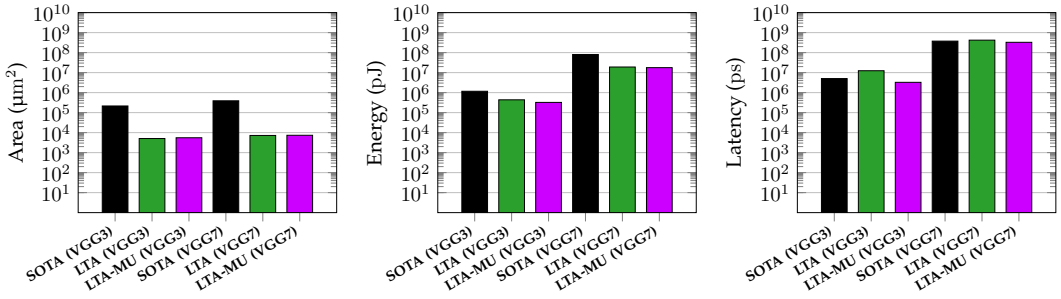


Figure 6.5: Accuracy over number of XNOR gates for different datasets. In the baseline case, the BNN is executed with the LTA for varying numbers of XNOR gates, called "baseline" (the training employs only standard methods). In the "LTA-train" case, the BNN is trained with a specified number of XNOR gates. The original test accuracy is shown with the dashed line. The accuracy tradeoff is explained in Sec. 6.1.5.2. Since there can be different types of noise in the analog circuit, we show the baseline BNN accuracy with a general type of noise injected alongside applying the LTA (the evaluations regarding noise are explained in Sec. 6.1.5.4).

accuracy. An explanation for this is that the majority vote is disturbed by the rest in Eq. (6.3) because of the layer dimensions.

To alleviate these issues, we apply the LTA-train method in Sec. 6.1.3. In these cases, the accuracy is high consistently, and the difference to the original accuracy becomes small without any fluctuations. For example, for $n = 64$ XNOR gates, which is a reasonable number for a crossbar (see Sec. 6.1.5.3 for the explanation), the accuracy of Kuzushiji is 89.25% (93.74% baseline), for Fashion 88.34% (90.68% baseline), for SVHN is 91.88% (92.84% baseline), and for Imagenette 72.00% (72.15% baseline). For CIFAR10, the accuracy tradeoff is larger, i.e. it is 77.85 (85.50% baseline), which reaches the maximum of 82.05% for 192 XNOR gates. The reason for the low accuracy in case of CIFAR10 is that the BNN cannot tolerate the approximations for this dataset. SVHN uses the same BNN architecture and is a less challenging dataset than CIFAR10, while having high accuracy. For CIFAR10, a more approximation tolerant BNN model needs to be chosen to apply the LTA with higher accuracy.



(a) Area (b) Energy (c) Latency

Figure 6.6: Area, energy, and latency comparison for the crossbar and interface circuits between the state of the art (SOTA), and our proposed method (LTA). Note that the y-axes are in log scale. For the BNN models, VGG3 and VGG7 are used (Table 3.2).

6.1.5.3 *Area, Energy, Latecy*

The area, energy, and latency are calculated based on the formulas in Table 6.1. The formulas are composed of variables for A (area), E (energy), and L (latency). The variables are defined in Table 6.2. The values for these variables (A, E, and L) are acquired for digital components by own simulations (in Cadence Genus using 28nm FDSOI technology). For the ADC and the analog comparator, the values are taken from other studies. For the latency and energy usage of the FeFET-based crossbar, we perform circuit simulations based on HSPICE. We explain the details in the following.

We assume the same crossbar configurations as in [Che+18], i.e. it is based on FeFET technology and it has the dimensions $m = 64 \times n = 64$. In the literature, FeFET-based XNOR crossbars have been built for 64×64 in [Che+18] and for 48×64 in [Sol+20b]. Importantly, in [Sol+20a], a full FeFET-based crossbar array for a commercial 28nm technology node from GlobalFoundries is demonstrated, and their design only supports activating 64 FeFET gates at once to combat effects from variations and avoid the cost of large ADCs. In order to compare the LTA to the SOTA in our manuscript, we use the size 64×64 for a specific example of a realistic analog-based BNN accelerator.

In our evaluations, the crossbar energy and latency for application are relevant, since the number of crossbar invocation differs among the LTA/LTA-MU and SOTA execution schemes. To estimate the energy and latency of the crossbar, we use the FeFET simulations and the FeFET transistor introduced in Ch. 5 and build the FeFET-based XNOR logic according to Sec. 5.3.1. From these components we form a FeFinFET-based XNOR device array and we then measure the delay, power, and energy using HSPICE. Depending on the number of mismatches between the input data and stored data, the array will conduct more or less current (i.e., the higher the mismatches level, the larger the current). We then calculate the energy and latency for the worst case in which the highest level of mismatches occurs.

Based on the above, the energy usage of the crossbar for application is 1.32 pJ per crossbar column. To calculate the energy usage of an entire crossbar, this number is multiplied by the number of used crossbar columns. The latency of one FeFET-based XNOR gate (and therefore the entire crossbar due to parallelism) is 706 ps according to the study in Sec. 5.3. We do not need to incorporate the reprogramming performance of the XNOR gates, since the number of reprogrammings is the same in the LTA and the SOTA execution. The area usage of the XNOR gates is also not incorporated, since the crossbar is assumed to be the same in each technique.

To evaluate the LTA, the area, energy, and latency of the analog and digital components of the interface circuit is reported in Table 6.4. For the digital path we synthesized the hardware for the two highest values of $\beta = 3136$ and 8192 (for VGG3 and VGG7 respectively, such that all layers of one BNN model can be executed with one device) using Cadence Genus and is mapped to a commercial 28nm FD-SOI technology. For the analog components, we have selected from the literature the ADC and the analog comparator designed in the same technological node size as we used to synthesize the digital paths (i.e. 28nm). In particular, we selected the ADC in [Oh+20] for our study because it can be inferred from the survey in [Mur] that it has the best area and energy tradeoff among the reported 28nm ADC. In a similar way, we selected the analog comparator in [RST19] since it has the best performance among the 28nm comparators we could find in the literature.

By considering that the interface circuit of our analog BNN accelerator is composed of these analog and digital subcomponents, it enables us to compare our proposed LTA to the SOTA with up-to-date subcomponents. For calculating area/energy/latency and compare our LTA to the SOTA, we rely on the formulas in Table 6.1, for which the notation is explained in Table 6.2.

With the crossbar sizes $n = m = 64$ and the corresponding interface circuit data, we evaluate the area, energy, and latency of our LTA/LTA-MU computation schemes and compare them to the SOTA in Fig. 6.6. For the BNN models in our work, in VGG3 only the AP in Fig. 6.3 is used. For VGG7 the DP is only used for layers 5 and 6 (see Table 6.3). Our results show that the proposed LTA technique is able to reduce the total area by a factor of 42× and 54× for VGG3 and VGG7, respectively, compared to the SOTA. Furthermore, the energy consumption is reduced by a factor of 2.7× and 4.2× for VGG3 and VGG7 compared to the SOTA, respectively. LTA does not show a reduction of latency. It is 2.5× and 1.12× higher than the SOTA, for VGG3 and VGG7 respectively. However, when the LTA-MU scheme is employed, the latency is reduced by 3.8× and 1.15× compared to the SOTA for VGG3 and VGG7, respectively. As a drawback, the LTA-MU requires 9.2% (VGG3) and 2.2% (VGG7) more area compared to its LTA counterpart.

6.1.5.4 *Impact of Noise on LTA*

When n XNOR gates are in a crossbar column, then the analog comparator needs to be able to differentiate between n different states. However, n cannot be arbitrary

NN Architecture	Layer index	$\mathbf{W}(\alpha, \beta)$	$\mathbf{X}(\gamma, \delta)$
VGG3	1	(64, 576)	(576, 196)
	2	(2048, 3136)	(3136, 1)
VGG7	1	(128, 1152)	(1152, 1024)
	2	(256, 1152)	(1152, 256)
	3	(256, 2304)	(2304, 256)
	4	(512, 2304)	(2304, 64)
	5	(512, 4608)	(4608, 64)
	6	(1024, 8192)	(8192, 1)

Table 6.3: Matrix dimensions of the weight matrix **W** and input **X**.

Interface circuit parameters (28 nm technology node)				
Component	Specification	Energy (pJ/op)	Area (μm^2)	Latency (ps)
Analog Comparator	See [RST19]	0.163	78	74
ADC	See [Oh+20]	2.55	2000	1000
Digital path SOTA	VGG3 | VGG7	1.61 | 4.51	1282.10 | 4011.00	270
Digital path LTA	VGG7	0.223	150.9	240

Table 6.4: Energy, area, and latency configurations of the interface circuit's subcomponents, based on the literature and own evaluations (i.e. in Cadence Genus using commercial 28nm FDSOI technology). For the digital components, $\beta = 3136$ for VGG3, and $\beta = 8192$ for VGG7. Note that for VGG3 under LTA, digital components are not used. The total energy, area, and latency of the BNN crossbar and interface circuit are calculated based on the values in this table, which are substituted in the area, energy, and latency formulas in Table 6.1.

large. Due to inherent variations of the analog signals an arbitrary number of different states cannot be accomodated.

There are multiple potential sources of noise in our analog hardware design. Although the sources of noise are not limited to the following cases, for explaining the concept, we focus on noise caused by variation sources due to: (1) FeFET-based crossbar columns and (2) resistance value of the resistors. (1) In the crossbar columns, the FeFET-based XNOR gates consist of FeFET devices, which are prone to errors when there are temperature fluctuations during run-time (however, in our study, we assumed no major fluctuations in operating temperature during run-time). In our previous chapter, we have demonstrated and investigated the impact of temperature in Sec. 5.2 for single FeFET devices. Since FeFET-based XNOR gates consist of two FeFET devices, the current coming out of the XNOR gates will have variations due to high temperature fluctuations as well. Due to the variations of the XNOR gate currents, the summed current also experiences variation. This in turn may lead to

Name	No noise, no LTA	No noise	**No noise+LTAtr**	Noise 1% | **Noise+LTAtr**	Noise 2% | **Noise+LTAtr**	Noise 5% | **Noise+LTAtr**
FashionMNIST	90.68	81.34	**88.34**	78.79 | **87.89**	76.38 | **87.41**	65.03 | **86.90**
KuzushijiMNIST	93.74	83.05	**89.25**	80.50 | **88.55**	76.16 | **87.19**	64.31 | **85.38**
SVHN	92.84	78.62	**91.88**	73.39 | **91.01**	67.13 | **90.54**	42.09 | **88.14**
CIFAR10	85.50	43.07	**77.85**	35.08 | **74.46**	28.23 | **73.44**	14.79 | **67.31**
IMAGENETTE	72.15	57.02	**72.00**	52.89 | **71.69**	48.31 | **70.55**	31.75 | **70.37**

Table 6.5: Comparison of test accuracy (%) for the assumed cases in our with crossbar size 64 × 64. The LTA approximation is applied in each column unless specified otherwise. In cases of "no noise"/"no LTA", we train without noise/without LTA. "Noise" followed by a percentage means that noise with this percentage is injected, while in "Noise train", we train with the noise. "Noise+LTAtr" refers to the case in which we inject noise during the training while simultaneously applying Alg. 5.

errors of the comparator outputs (flips from "0" to "1" or from "1" to "0") that binarize the result of one crossbar column. (2) The resistors connected to the output of the comparators (see R_2 in Fig. 6.3) may suffer from varying resistance values as well (e.g. due to issues in fabrication or temperature), leading to current variation. The effect of the noise from points (1) and (2) above continues to propagate through the circuit, which has two paths: The analog path and the digital path. In the analog path, the result of the majority comparator may be flipped due to the current variations (i.e., the output may also flip from "0" to "1" or from "1" to "0"). In the digital path, the ADC may convert the analog signal to an erroneous digital value, which will be accumulated and binarized in the digital domain. In both cases, any variations in the analog computations (e.g. in points (1) and (2) above) will affect the result of the *last binarization*. In the analog path, the output of the last analog comparator will be affected. In the digital path, the output of the digital binarizer will be affected. Therefore, we simulate the noise in the circuit by flipping the output of the last binarization with a certain error probability.

Noise Analysis: To conduct a general noise study, we model the noise by using the flip probabilities 1%, 2%, and 5% for the outputs of the last binarizations. Please note that we are not limited to these error rates, they are merely examples. Any other error rate can be applied in our open-source framework. When we write "Noise training" or "Noise+LTAtr", we refer to the cases in which we always train from scratch with noise and also the LTA (Alg. 5). In Fig. 6.5, we show the plots for the accuracy results achieved by the BNNs under the noise in combination with the LTA. We observe that for small noise, the accuracy degradation is also small. The larger the noise, the higher the accuracy degradation. In Table 6.5, we also show the result of the retraining together with the LTA approximations and noise in combination for the crossbar column dimension $n = 64$. We also add the results with no noise for reference. We observe that in all cases of our experiments, although the noise causes large accuracy drops, a significant amount of accuracy can be regained by training with the noise.

6.1.5.5 *Feasibility of LTA Input Data Flow*

To show the feasibility of the input application method, we implement the two types of data flow (SOTA and LTA, as presented in Sec. 6.1.4.1) in VHDL. The crossbar dimensions are $n = m = 64$. For the SOTA case, we consider that one input is broadcasted to all computing columns. For the LTA case, we assume that each column receives a different input. In both designs, the weights are set once and stay the same throughout the computations to simulate the strided move.

To focus on the dataflow, we simulate the input application to the XNOR gates of all columns. We synthesize the designs using Cadence Genus with 28nm FD-SOI technology (as in the previous digital circuits). For the SOTA dataflow, 15928 μm^2 is required, along with 273.55 mW of power consumption. For the LTA dataflow, 24733 μm^2 is required, along with 360.05 mW of power consumption. Both data flows are configured for 300 ps.

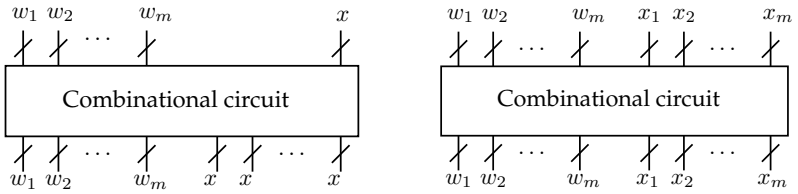
We observe that the LTA data flow uses 31.6% more power and 55.3% more area. However, note that when considering the entire crossbar accelerator, the SOTA requires 128000 μm^2 of area for the ADCs alone, while the LTA only needs 2000 μm^2 of area for the ADCs alone. This is a decrease by 64× (since in the SOTA circuit, $m = 64$ ADCs are needed for $m = 64$ computing columns, and in the LTA only 1 ADC is needed for $m = 64$ computing columns) and the mere increase of LTA by 55.3% is small compared to the 64× area reduction. Calculations in the similar scale can be performed for the energy consumption as well.

The main contribution for area and power in the LTA dataflow implementation is the increased number of wires (for area) and the different signals (power) that need to be driven, compared to the SOTA. We illustrate this by the implementation in Fig. 6.7 using combinational circuits consisting of wires, which are placed between the memory and the BNN crossbar. Note that memory and the crossbar could also be connected in a different way, Fig. 6.7 is one example. In the SOTA, in Fig. 6.7(a), a single input bitvector x (with length n) is retrieved, and then it is replicated as many times as there are parallel columns (i.e. m) in the crossbar. This means, to enable the SOTA dataflow, the single input bitvector x is replicated m times with a combinational circuit (i.e. single in, multiple out) and then the m same bitvectors of x are passed to the crossbar. In the LTA method, m different inputs are needed, labeled x_i in Fig. 6.7(b), which are then applied to the m crossbars (multiple in, multiple out). In summary, the increased area and power usage of the LTA are due to the number of wires and their driving with different signals.

6.1.5.6 *Discussion of other Methods*

We have also evaluated other methods to improve the accuracy of BNNs under LTA. However, the only method that worked consistently is the LTA-train method. We summarize these ideas here.

A possible idea to increase the accuracy under the LTA is to use different thresholds for each window, instead of using the same threshold for all windows. We have



(a) Input data flow circuit for SOTA. (b) Input data flow circuit for LTA.

Figure 6.7: Examples of using combinational circuits consisting of wires to connect the BNN crossbar to memories (e.g between the FIFOs in Fig. 6.2 and the BNN crossbar). (a) SOTA and (b) the LTA connections using combinational circuits. The w_i are the weight vectors that are composed of n bits. The weight wires transfer the weights to the ith column of the crossbar. In the SOTA, the input vector (composed of n bits) is the same for each crossbar column (it is replicated). In the LTA, each crossbar column i receives a different input x_i.

attempted to increase/decrease the thresholds based on the mean popcount (over the training data) in windows of n bits. However it did not lead to higher accuracies. We observed that most popcount results in the windows of size n have similar values that are closely around the mean, and that there are not any patterns to exploit for different thresholds (we show histograms of the popcount values for BNNs in Fig. 7.1). For other n and other datasets, we also observe the mean $\frac{n}{2}$ and similar distributions. In general, modifying the thresholds in BNNs manually to optimize certain properties has been reported to be unsuccessful in other work as well [Bus+20], which is also included in Sec. 4.2.1.

Another idea for the LTA is to use a smaller stride than n by moving the windows such that there are overlaps, leading to more local thresholdings. In our explorations, this only alleviated the sharp drops in accuracy by a small amount (see the accuracy drops in the black plots in Fig. 6.5). It did not lead to an increase in accuracy for the peaks. With a smaller stride, more computations need to be performed, taking up crossbar space, which is a high cost.

We observed negative results for modifying the majority vote as well. In our framework, the majority can be shifted, meaning it can be configured that for a "1" (instead of "0") as output, there needs to be a majority and a certain number of additional local thresholds that are "1", for a "1" in the final output. Introducing majority vote shifts based on the observed mean "1"s in the local thresholds also lead to no accuracy benefits or when the shifts become high, to poorer accuracy results.

The operation with custom thresholds, the shifted majority votes, and using different strides are all implemented in our framework as command line parameters and can be can be evaluated by the users to perform more research.

6.1.5.7 *Discussion of other BNN models*

To demonstrate the proof of concept of the LTA approximation, we assume in Sec. 6.1.2 that the layers have the following structure, without any operations inbetween: Convolution, batch norm, activation. Currently, our proposed LTA cannot be used with architectures that do not comply with this assumption. One notable example are skip connections, such as those in ResNets (see Table 3.2). For computing the skip connection, a convolution is computed, then the result of another convolution is added to the previous result, after which an activation is applied. To enable the LTA for other structures, such as the skip connections, the BNN structure needs to be modified, which may also require the modification of the BNN acceleration hardware and the training procedure.

6.1.5.8 *Discussion of ADC Modifications*

In Table 6.4, we observe that the ADC resource usage is significantly higher than the other components. Therefore, we discuss the expected impact of using an ADC with smaller resolution compared to our selected 8-bit ADC from [Oh+20].

The ADC we have selected is not configurable with respect to the resolution. Therefore, we rely on the data in the study in [Yu+21], where the resource consumption (area, latency, power, energy) of ADCs is analyzed as a function of the number of bits (3-6 bits) at the 40 nm technology node. Based on that study, we expect that when a smaller number of bits are used in the ADC, the resource costs for the ADC will be proportionally smaller.

However, when an ADC with a smaller resolution is used for our proposed LTA method, then the analog path will be triggered in less cases. This opposes our main design idea to trigger the analog path in as many cases as possible. Recall that in the LTA method, the condition for triggering the analog path for a layer is $\beta \leq mn$. In our study, the required ADC resolution depends on the crossbar dimension m (number of crossbar columns), i.e. the required ADC resolution is $\lfloor \log_2(m) \rfloor + 1$. Therefore, the smaller the m, the smaller the required resolution of the ADC. The drawback of this is that when m is reduced, then the analog path will be triggered in less cases, because the right side of the condition $\beta \leq mn$ for triggering the analog path will be smaller. In addition to that, a smaller number columns increases the inference latency approximately by the factor of the crossbar column reduction.

6.2 CAPMIN

As stated in the previous chapter, performing the computations of NNs in the analog domain by using Ohm's and Kirchhoff's laws can achieve high resource efficiency [Chi+16; Sha+16a]. An example of a computing scheme that exploits this for efficiency is Integrate-and-Fire (IF) Spiking Neural Networks (SNNs) [Wei+21a]. In IF-SNNs, neural activity is event-driven and described by the integration of current over a certain amount of time, for which a capacitor is used. If the charge in the

capacitor becomes high, a predetermined threshold potential is exceeded, causing the firing of an output spike. The IF-SNN operations use efficient coding and enable efficient operation, because simple analog components can be employed instead of costly components such as ADCs. Yet, analog computing based IF-SNNs suffer from nonidealities and variations. For example, analog multipliers or other components exhibit noisy behavior due to process variation or factors such as temperature. Furthermore, the capacitor may be too small for correct operation, and analog comparators have limited gain causing undefined output signals for connected digital components.

In particular, the capacitor size is a major bottleneck in the circuit design of IF-SNNs, leading to high energy, area, and latency cost [Xia+20; Dut+20; Wei+21b]. Furthermore, it determines the tolerance of the system to nonidealities or variations caused by inherent and external factors. To the best of our knowledge, principled approaches for minimizing the capacitor size, especially considering nonidealities or variations, did not exist at the time of the writing. The work in [Dut+20] compares several implementations of circuits for spike-based inference, where the capacitor size is determined empirically under the engineering constraints. The studies in [Wei+21b; Wei+21a] sweep the capacitor size and pick the one satisfying the accuracy constraint.

These approaches do not use the insights in the SW (NN models), to optimize the HW (IF-SNN circuits). In the IF-SNN HW, spike times are required to represent the MAC values that occur with low frequency, wasting valuable capacitor size, in turn leading to high resource cost. In this work, we focus on the capacitor size reduction in IF-SNNs executing BNNs. Due to their binary nature, BNNs are highly resource efficient and robust to variations [Bus+21], making them excellent candidates for analog-based computing with IF-SNNs. We reveal that many of the MAC values in the BNN SW have a low probability to occur during inference (also in NNs with higher precision regarding weights and inputs, see [Zha+19b; Din+19]). The lowest and the highest MAC values occur five to seven orders of magnitude less frequently compared to the MAC value at the mean (see histograms in Fig. 6.8 for five benchmarks). According to the results shown in Fig. 6.8, the histograms of the MAC values are normally distributed, with a sharp peak at the mean.

The key focus of this section is exploring HW/SW Codesign methods to achieve IF-SNN operation with a small capacitor, leading to efficient operation through reductions in energy, area, and latency. We focus on BNNs, which are highly efficient and robust to variations, making them excellent candidates to be executed with IF-SNN HW. The insights and methods gained by researching BNNs in this study may also be applicable to higher-precision NNs, if a significant portion of the MAC values in these models have a low probability to occur as well.

The remainder of this section is structured as follows. In Sec. 6.2.1, we explain the basics of analog-based computing in binarized IF-SNNs and in Sec. 6.2.2 we present the problem definition. In Sec. 6.2.3, we introduce our two methods: (1) CapMin, which decreases the required capacitor size, and (2) CapMin-V, which increases variation tolerance. Finally, in Sec. 6.2.4, we present the experiment results.

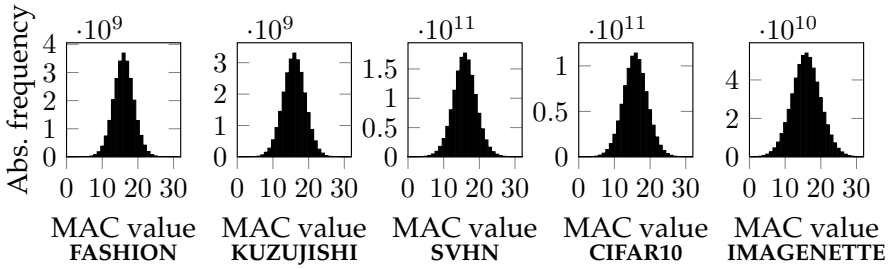


Figure 6.8: Abs. frequencies of MAC value occurrences (summed over layers) for training sets. Details of the BNN models are in Table 3.2.

6.2.1 *System Model of IF-SNNs*

We first present the basic operation of the HW used in IF-SNNs in Sec. 6.2.1.1. Then, in Sec. 6.2.1.2, we explain the basic concept of the capacitor, which plays a central role in IF-SNNs.

6.2.1.1 *Operation of Binarized IF-SNNs*

The circuit performing the operations of binarized IF-SNNs is shown in Fig. 6.9. In the computing array, a is the array size, x_i, i.e. x_1 to x_a, are the input spikes, and M_i, i.e. M_1 to M_a are the XNOR gates. To realize the XNOR, different techniques can be used, e.g. Ohm's law [Wei+21b]. The neuron circuit consists of a membrane capacitor $\mathrm{C}_{\mathrm{mem}}$ with capacitance C, an analog comparator A, and a flip flop (FF). The steps for computations of the MAC results in SNNs are as follows:

(1) The XNOR gates will be loaded with the correct weights or are assumed to be already loaded. We assume that the inputs x_i are provided to all multipliers in parallel, and that the multiplications are all computed in parallel as well.

(2) In the neuron circuit, the incoming summed current from the XNOR gates charges $\mathrm{C}_{\mathrm{mem}}$. Once the voltage across $\mathrm{C}_{\mathrm{mem}}$ reaches the threshold voltage V_{th}, an output spike is generated with the analog comparator. The spike time t_{fire} is acquired by a counter that tracks the clock cycles until the FF latches the spike signal.

(3) The spike time is converted to a MAC value by $\frac{v}{t_{\mathrm{fire}}} = \sum_{i=1}^{a} w_i x_i$, where $v = x_{\max} \frac{C V_{\mathrm{th}}}{I_{\mathrm{ON}}}$ and I_{ON} is the on-state current of the multiplier. The conversion can be described by mappings between sets. Consider the set of spike times $S_{\mathrm{FIRE}} = \{t_1, t_2, \ldots, t_L\}$, where t_L is the largest firing time, and $t_j < t_{j+1}$. Consider also the set of MAC-values $S_{\mathrm{MAC}} = \{q_1, q_2, \ldots, q_L\}$, where $q_j \leq q_{j+1}$. In the relation $S_{\mathrm{FIRE}} \rightarrow S_{\mathrm{MAC}}$, the values are mapped using $m_j : t_j \rightarrow q_{L-j+1}$. We organize the index of q_{L-j+1} in a reversed manner to describe the reciprocal relationship between the spike time and the MAC value. In the state of the art, L is chosen such that each MAC value has a unique spike time. After completing the calculations, the neuron is reset by the

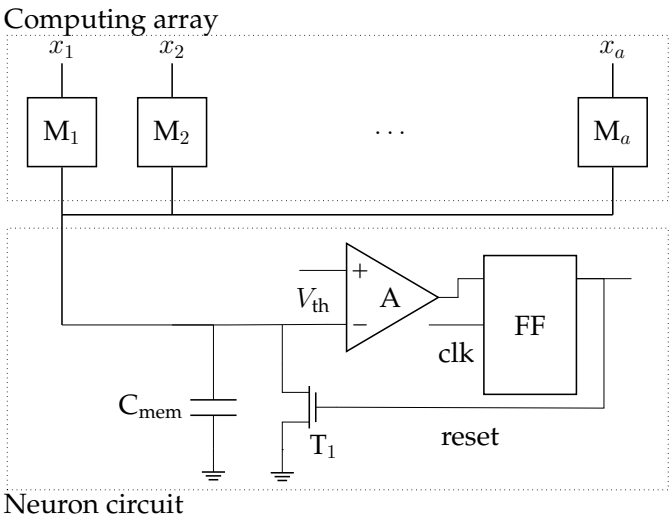


Figure 6.9: IF-SNN circuit. Top: Computing array with inputs x_1 to x_a and multipliers M_1 to M_a. Bottom: Neuron circuit with the membrane capacitor C_{mem}, analog comparator A and the FF.

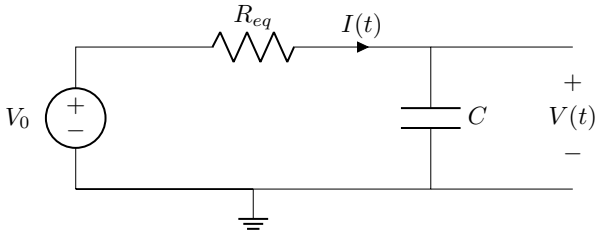


Figure 6.10: Equivalent representation of IF-SNN circuit in Fig. 6.9. V_0: Supply voltage. R_{eq}: Equivalent resistance of computing array. Voltage $V(t)$ across capacitor C is measured over time. $I(t)$ is the current over time flowing into the capacitor.

transistor T_1. Because of the limited computing array size, a large vector product (with dimension higher than a) is separated into multiple smaller vector products, requiring circuits for addition and accumulation.

6.2.1.2 *Capacitor in IF-SNNs*

Capacitors have the capacitance $C = \frac{Q}{V}$ (in F for Farad), determined by the charge Q placed on the capacitor divided by the voltage V caused by that charge. A capacitor is charged when a voltage, e.g. V_0 is applied, which causes a current to flow into it. The charging of a capacitor is described by the equation

$$V(t) = V_0(1 - e^{(-\frac{t}{\tau})}), \tag{6.4}$$

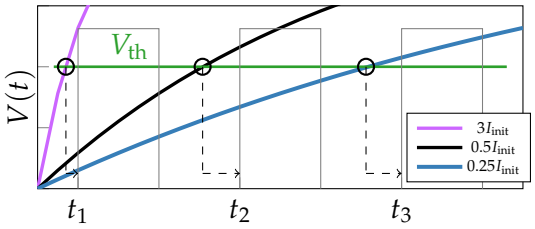


Figure 6.11: Voltage across capacitor over time, based on different initial currents. t_1, t_2, t_3 are spike times recorded by clock of the FF. Rectangle signal: Clock. Circled points: Ideal spike times.

where V_0 is the supply voltage, t the time, $\tau = R_{eq}C$ the time constant, and R_{eq} is the equivalent resistance of the connected circuit from the capacitor's perspective. We assume that no initial charge is in the capacitor. When charging, the capacitor voltage increases rapidly first, but slows down and stops at the maximum capacitor charge $Q = CV_0$. Since τ is in the denominator, smaller C lead to larger absolute values in the exponent, in turn causing faster capacitor charging. In contrast, a larger capacitor leads to slower charging. The same holds for smaller and larger R_{eq}.

In the IF-SNN circuit, R_{eq} plays an important role. It depends on the total resistance of all multipliers. The multipliers can have high or low resistance states, based on the programmed weights. Due to this, R_{eq} determines the size of the initial current I_{init} that flows into the capacitor. As the current is the first order derivative of the charge with respect to time, i.e., $I(t) = \frac{dQ}{dt}$, by adopting Eq. (6.4) for derivation, we have $I(t) = C\frac{dV(t)}{dt} = \frac{V_0}{R_{eq}}e^{-\frac{t}{\tau}}$. Thus, $I_{\text{init}} = \frac{V_0}{R_{eq}}$, the initial current at $t = 0$, is the largest current. When the capacitor is fully charged ($t = \infty$), no current flows. With I_{init} and V_0, the resistance of the computing array is $R_{eq} = \frac{V_0}{I_{\text{init}}}$ by Ohm's law. When inserting R_{eq} into Eq. (6.4), we get:

$$V(t) = V_0(1 - e^{(-\frac{t}{C}\frac{I_{\text{init}}}{V_0})}). \tag{6.5}$$

Therefore, the larger I_{init}, the faster the capacitor is charging. In Fig. 6.11, the voltage curves for different I_{init} are shown. The equivalent RC circuit with V_0, R_{eq}, $I(t)$, and $V(t)$ is in Fig. 6.10.

The charging properties of capacitors are used to realize the operation of IF-SNN circuits. A spike occurs ideally when the charge, which is the integrated current from the computing array over time, in the capacitor leads to $V(t) = V_{\text{th}}$. The ideal firing times, where the voltage curve and the V_{th}-line cross, are marked with circles in Fig. 6.11. A spike can only be registered by the FF at the rising edges of the clock, which is shown in the gray signal in Fig. 6.11). The time points of these clock-spike times are collected in S_{FIRE}.

With a fixed clock frequency, the size of the capacitor is chosen such that all required firing times in the set S_{FIRE} are represented uniquely. The higher the number of firing times to include, the larger the required capacitor size.

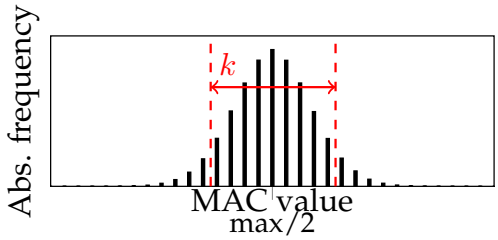


Figure 6.12: Role of inclusion parameter k in histogram of MAC values. All MAC values within borders get a unique spike time value assigned. The larger k, the more values within the borders.

6.2.2 *Problem Definition*

We are given a BNN model and an IF-SNN circuit (see Sec. 6.2.1) to perform its computations. The IF-SNN circuit has a capacitor with capacitance C, whose behavior is described in Eq. (6.4). For every different summed current that flows from the computing array to the capacitor, a unique spike time is placed in S_{FIRE}, representing a MAC value in S_{MAC}. The higher the number of spike times used to represent the MAC values, the larger the required capacitor size.

In this work, our first goal is to construct the sets S_{FIRE} and S_{MAC}, such that the capacitor size is minimized and therefore *energy, area, and latency* of the IF-SNN circuit, while limiting the inference accuracy drop of the BNN. Our second goal is to modify the above acquired sets S_{FIRE} and S_{MAC}, such that the tolerance to process variation, which can significantly affect the correctness of analog computing schemes, is increased.

6.2.3 *Our Proposed Methods: CapMin and CapMin-V*

In Sec. 6.2.3.1, we propose our method CapMin, in which spike times in S_{FIRE} are only assigned to the most important MAC values. CapMin does not protect against process variation, therefore, in Sec. 6.2.3.2, we present CapMin-V, which aims to achieve variation tolerant IF-SNN operation by trading off with capacitor size.

6.2.3.1 *Our Method CapMin for Capacitor Minimization*

We consider that the most important MAC values are the ones that occur most frequently during the inference of NNs. In Fig. 6.8, we present the histogram of all MAC value occurrences in forward passes with the training set. From this intuition, we propose a capacitor minimization procedure. We reduce the number of required spike times in S_{FIRE} based on the absolute frequency of the observed MAC value occurrences in S_{MAC}. This reduces the capacitor size.

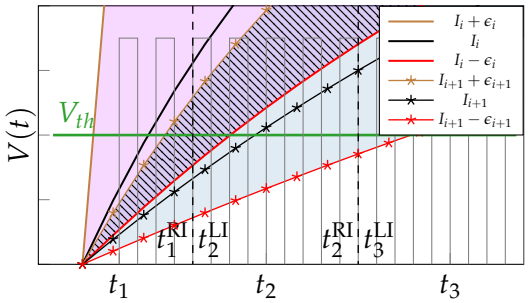


Figure 6.13: Effect of current variation on capacitor charging. Charging is shown in black for I_i and I_{i+1}. Depending on the sign of the variation (ϵ_i or ϵ_{i+1}), the capacitor may charge faster (brown) or slower (red). Variations can cause any deviation in the purple (for I_i) or the blue area (for I_{i+1}). Charging curves under variation may overlap (striped area).

We denote the MAC values occurring during IF-SNNs operation with $S_{\mathrm{MAC}} = \{q_1, q_2, \ldots, q_L\}$. To these values, spike times in $S_{\mathrm{FIRE}} = \{t_1, t_2, \ldots, t_L\}$ are assigned bijectively. There are L mappings $m_j : t_j \rightarrow q_{L-j+1}$ (see Sec. 6.2.1.1). For each MAC value in S_{MAC}, we extract its absolute frequency of occurrences (AFO), i.e. $F_{\mathrm{MAC}} = \{f_1, f_2, \ldots, f_L\}$, where f_i counts the number of occurrences of the MAC value q_i. This is achieved by tracking the MAC values in the computing array (Fig. 6.9) during the inference of the NN.

We use the set F_{MAC} to minimize the number of MAC values in S_{MAC}, to construct a set $S_{\mathrm{MAC,min}}$. To this end, only the k MAC levels with the highest AFO are added to $S_{\mathrm{MAC,min}}$. MAC values that have low AFO are not added to $S_{\mathrm{MAC,min}}$ and get mapped to the nearest MAC level in $S_{\mathrm{MAC,min}}$. The role of k and its clipping behavior in the histogram of F_{MAC} is shown in Fig. 6.12. The value of k can be configured based on the desired number of MAC levels in $S_{\mathrm{MAC,min}}$. Since the mapping from $S_{\mathrm{MAC,min}}$ to $S_{\mathrm{FIRE,min}}$ is bijective, the number of spike times in $S_{\mathrm{FIRE,min}}$ that are needed to represent the MAC values are limited by k. In turn, less spike times require a smaller capacitor in the neuron circuit, leading to its minimization.

As a result of CapMin, the original set S_{MAC} is clipped to the set $S_{\mathrm{MAC,min}}$ based in the information in F_{MAC} and on k. For clipping the set S_{MAC} to $S_{\mathrm{MAC,min}}$ the following function is used, where the smallest MAC value in $S_{\mathrm{MAC,min}}$ is q_{first}, q_{last} the largest, and the MAC value is $M = \sum_{i=1}^{a} w_i x_i$:

$$M = \begin{cases} M, & \text{for } q_{\mathrm{first}} \leq M \leq q_{\mathrm{last}} \\ q_{\mathrm{first}}, & \text{for } M \leq q_{\mathrm{first}} \\ q_{\mathrm{last}}, & \text{for } M \geq q_{\mathrm{last}} \end{cases}. \tag{6.6}$$

6.2.3.2 *CapMin-V*

We solve Eq. (6.5) for t when $V(t) = V_{\text{th}}$. Denoting I_i with index i for the ith initial current instead of I_{init}, we get

$$t(I_i) = -\frac{CV_0}{I_i} \ln(1 - \frac{V_{\text{th}}}{V_0}), \tag{6.7}$$

where I_i is decreasing with increasing i, i.e. $I_i > I_{i+1}$. In IF-SNNs, I_i leads to the spike time t_i in S_{FIRE}. For example, I_1 is the largest current leading to the shortest spike time t_1 (mapped to the highest MAC value q_L). I_L is the smallest current leading to the longest spike time (mapped the smallest MAC value q_1). Since the currents coming out of the XNOR cell are all the same (assuming the same states), the difference between I_i and I_{i+1} is constant: $I_i - I_{i+1} = c > 0 \; \forall i$.

Without variations, the function $t(I_i)$ is deterministic. It produces the same spike time t_i (in the set S_{FIRE}) for a certain I_i (see black plots in Fig. 6.13). If I_i has variations, $t(I_i)$ will also change. The variations in I_i are proportional to I_i (a certain percentage of it), with a certain mean and variance. We define the measured maximum of I_i variation as ϵ_i. Due to ϵ_i of I_i, t may fall into the interval $\mathcal{E}_i = [t(I_i + \epsilon_i), t(I_i - \epsilon_i)]$, where $|\mathcal{E}_i|$ is its length. In this case, a t that is not in the set S_{FIRE} will be calculated in Eq. (6.7). The result of Eq. (6.7) under variations is assigned to the nearest t_i in S_{FIRE}, where the midpoints between two spike times are the assignment thresholds. The assignment threshold on the right of t_i is $t_i^{\text{RI}} = t_i + \frac{t_{i+1} - t_i}{2}$ and $t_i^{\text{LI}} = t_i - \frac{t_i - t_{i-1}}{2}$ on the left. We define the interval $B_i = [t_i^{\text{LI}}, t_i^{\text{RI}}]$ and its length as $|B_i|$. The interval boundaries are shown in the dashed vertical lines in Fig. 6.13. Any spike time that occurs in B_i is assigned to t_i. If the variation of I_i is large enough to make Eq. (6.7) cross the interval borders t_i^{LI} or t_i^{RI}, I_i will erroneously be assigned to a wrong spike time, e.g. t_{i-1}, t_{i+1}, or other spike times farther away. This is shown in the striped area in Fig. 6.13.

The probabilities for t_i to assume t_j, which may be different than t_i due to current variations, are modeled in the matrix in Eq. (6.8). The first index in P_{map} describes the spike time that has variations. The second index describes the erroneous spike time selected due to variations. For example, t_1 has the probability $p_{1,1}$ to assume t_1 and $p_{1,2}$ for t_2. If all the diagonal elements are "1" and the rest "0", then it is equivalent to the direct mapping mapping in Sec. 6.2.3.1 (ideal case, no variations).

$$P_{\text{map}} = \begin{bmatrix} p_{1,1} & p_{1,2} & \dots & p_{1,L} \\ p_{2,1} & p_{2,2} & \dots & p_{2,L} \\ \dots & \dots & \dots & \dots \\ p_{L,1} & p_{L,2} & \dots & p_{L,L} \end{bmatrix} \tag{6.8}$$

To increase variation tolerance, it needs to be known which spike times have lower or higher variation tolerance. Consider the differences in length between the intervals $\mathcal{E}_i$ and B_i. Due to the increase of $t(I_i)$ with smaller I_i, $|B_i|$ gets larger. In the same way, $|\mathcal{E}_i|$ gets larger as well. However, the variation ϵ_i becomes smaller with smaller

currents, since $I_i - I_{i+1} = c > 0 \; \forall i$ is assumed to be constant, while ϵ_i is proportional to the size of I_i. To conclude, the intervals B_i and $\mathcal{E}_i$ both get larger with increasing i, but the variations ϵ_i become smaller, i.e. as I_i becomes smaller, the ratio $r_i = \frac{|B_i|}{|\mathcal{E}_i|}$ gets larger, with a larger margin for tolerating variations. Based on this, we hypothesize that the spike times with larger t_i are more tolerant to variations than spike times with smaller t_i.

Based on this hypothesis and the error matrix in Eq. (6.8), we propose the method CapMin-V to increase the tolerance of the IF-SNNs to variations at the cost of capacitor size. As the starting point, we use the set $S_{\text{FIRE,min}}$ with k elements and extract its P_{map}. We aim to increase the probabilities in P_{map} for a spike time to assume correct values under current variations. Therefore, the criterion for optimization is to maximize the individual probabilities on the diagonal, i.e. the $p_{i,i}$. In CapMin-V, the $p_{i,i}$ are increased by merging the spike times t_i in $S_{\text{FIRE,min}}$ of the smallest $p_{i,i}$ with the neighboring spike times. By this way, a spike time with higher $p_{i,i}$ is created, which in turn leads to higher tolerance to current variations since the r_i is increased.

Before merging, t_i and t_{i+1} in $S_{\text{FIRE,min}}$ are different spike times. With application of CapMin-V, the time intervals of t_i and t_{i+1} are merged to create a new, variation tolerant spike time. When the neighboring t_{i+1} (or t_{i-1}) is merged with t_i, the spike time t_i^+ is created, which has a larger time margin to the subsequent spike time compared to t_i. The new spike time intervals of t_i^+ are $t_{i+1}^{\text{RI+}} = t_i + \frac{t_{i+2}+t_{i+1}}{2}$ and $t_i^{\text{LI+}} = t_i - \frac{t_i+t_{i-1}}{2}$ on the left (stays the same). The interval border to the right of t_i^+ is larger than the one from t_i, since $t_{i+1} + t_i < t_{i+2} + t_{i+1}$. The same holds for merging time t_i with t_{i-1}, just the different way around. To merge, the probabilities of two neighboring columns in Eq. (6.8) need to be added, i.e. $p_{i,j+1} \leftarrow p_{i,j+1} + p_{i,j} \forall j$ for merging t_i with t_{i+1}, and $p_{i,j-1} \leftarrow p_{i,j-1} + p_{i,j} \forall j$ for merging t_i with t_{i-1}. Due to the summing of probabilities, the probabilities on the diagonal are increased.

The procedure of CapMin-V is in Alg. 6. $S_{\text{FIRE,min}}$ (from CapMin) and ϕ (nr. of mergings to be performed) are the inputs. First, $S^V_{\text{FIRE,min}}$ is initialized as $S_{\text{FIRE,min}}$. To maximize the $p_{i,i}$, the minimum $p_{i,i}$ in P_{map} is determined and the index is stored in j. If j is the right bound, a left merge will be performed, and the other way around for the left bound. Then the column of the smallest $p_{i,i}$ is merged with a neighboring column. Whether to merge it left or right is decided by the $p_{i,i}$ of the left or right neighbor. If $p_{i-1,i-1}$ (diagonal entry of left neighbor) is smaller than $p_{i+1,i+1}$ (diagonal entry of right neighbor), a left merge will be performed. Otherwise, a right merge will be performed. Boundary cases are merged to inner directions and ties are broken arbitrarily. After adding the probabilities, in P_{map}, the column of the merged spike time $p_{i,i}$ is removed (since it has been added to the neighboring spike time) and its row as well, since the spike time does not occur any more. Then, k_V is decremented. After the specified number of mergings ϕ, the algorithm pads P_{map} with zeros on the left and right, and adds 1s to realize the clipping from CapMin. Finally, $S^V_{\text{FIRE,min}}$ is returned and its spike times are mapped to the k most frequently occurring MAC values.

Algorithm 6: CapMin-V: Constructing the set $S^V_{\text{FIRE,min}}$

Input: ϕ, $S_{\text{FIRE,min}} = \{t_1, t_2, \ldots, t_k\}$
Output: $S^V_{\text{FIRE,min}}$

$S^V_{\text{FIRE,min}} \leftarrow S_{\text{FIRE,min}}$
$\phi_{\text{step}} \leftarrow 1$, $k_V \leftarrow k$
while $\phi_{step} \leq \phi$ **do**
  $j \leftarrow \text{argmin}(\text{diag}(P_{\text{map}}))$
  Handle out-of-bound cases
  if $p_{j-1,j-1} < p_{j+1,j+1}$ **then**
    for i *in* $\{1, \ldots, k_V\}$ **do**
      $p_{i,j-1} \leftarrow p_{i,j-1} + p_{i,j}$
  else
    for i *in* $\{1, \ldots, k_V\}$ **do**
      $p_{i,j+1} \leftarrow p_{i,j+1} + p_{i,j}$
  Remove column and row j from P_{map}
  Remove t_j from $S^V_{\text{FIRE,min}}$
  $\phi_{\text{step}} \leftarrow \phi_{\text{step}} + 1$, $k_V \leftarrow k_V - 1$
Add padding to P_{map}
return $S^V_{\text{FIRE,min}}$

6.2.4 *Experiments*

In this section we evaluate the benefit of CapMin and Capmin-V regarding accuracy, area, energy, and latency. In Sec. 6.2.4.1, we present the experiment setup. CapMin in evaluated in Sec. 6.2.4.2 and CapMin-V in Sec. 6.2.4.3.

6.2.4.1 *Experiment Setup*

We use PyTorch for the high-level simulation of BNNs for the SW and use SPICE for the device-level simulations of the HW.

Setup for PyTorch: To demonstrate the effectiveness of our proposed methods, we employ BNNs which are executed as IF-SNNs using the hardware configuration in Fig. 6.9. We use the datasets Fashion, Kuzujishi, SVHN, CIFAR10, and Imagenette, and other training settings that are explained in Ch. 3. As the loss we use the MHL (Ch. 4) with $b = 128$ in all cases. We use the training sets to extract F_{MAC} for the methods in Sec. 5.2.5 and evaluate the accuracy using the test sets. Note that we do not train with the errors. All our methods are applied in the post-training stage without any modifications to the BNNs. To evaluate the BNNs under CapMin or CapMin-V, our framework in `https://github.com/myay/SPICE-Torch` loads the the

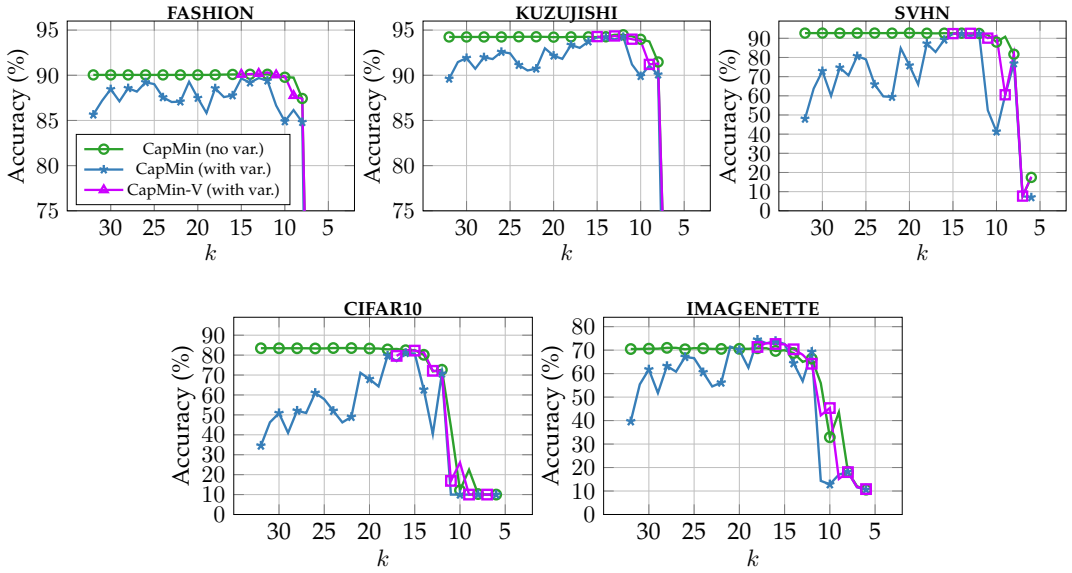


Figure 6.14: Accuracy over k. The higher k, the larger capacitor size. Capacitor size range: From 135.2 pF ($k = 32$) to 1 pF ($k = 5$).

information about the clippings (Eq. (6.6)) and the error models (Eq. (6.8)) and applies them during the MAC computations of the BNNs in PyTorch.

Setup for SPICE: In the computing array (Fig. 6.9), we use SRAM-based XNOR cells with 14 nm FD-SOI technology, of which we reproduce industry measurements with a ultra-thin body and BOX design [Liu+13]. The transistor model-card parameters for the industry-standard compact model of FD-SOI (BSIM-IMG) are carefully tuned until they are in excellent agreement with the measurements. The model is also calibrated to device-to-device variation measurements. For a comprehensive variability representation, all important sources of process variation (gate work function, channel dimension, BOX and channel thickness) are considered. Through SPICE Monte-Carlo simulations based on the calibrated compact model, the standard deviation for each model parameter is tuned to match the observed variation in the measurements. We use an array of $a = 32$ XNOR cells to realize the computing array. Each XNOR cell connects V_0 to the shared ML and forms a conducting path if the weight does *not* match the respective multiplication result, realizing the XNOR operation. Through the shared ML, Kirchhoff's law accumulates the individual results. The resulting current is proportional to the MAC value and charges the capacitor. To reduce SPICE simulation time, we use ideal Verilog-A implementations of the comparator and the FF wich operates with 2 GHz.

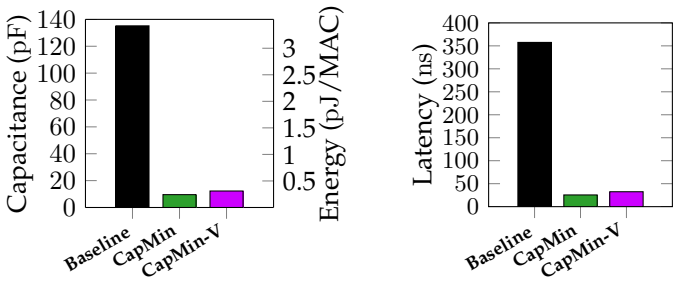


Figure 6.15: Capacitor size and latency comparison of the neuron circuit (based on max. capacitor size over the four datasets) for the baseline and our two proposed methods at 1% accuracy cost.

6.2.4.2 *Minimizing Capacitor Size with CapMin*

We extract F_{MAC} by forward passes with the BNNs using the training datasets. In Fig. 6.8 are the histograms of the absolute frequency of occurring MAC values. Since all histograms are similar, we normalize and add all the absolute frequencies across datasets and use the resulting F_{MAC} in CapMin (Sec. 6.2.3.1).

We apply CapMin using F_{MAC}, k, and $a = 32$ to obtain the set $S_{\text{MAC,min}}$. In Fig. 6.14, the accuracy (test set) for different k is shown in the plots (circle marks), starting with $k = 32$ (max. nr. of levels for $a = 32$) down to $k = 5$. For Fashion and Kuzujishi, the accuracy is sustained until $k = 8$ and then drops sharply for smaller k. For SVHN, CIFAR10, and Imagenette the accuracy is sustained until $k = 8$ (SVHN) as well and $k = 14$ (CIFAR10 and Imagenette) respectively.

In Fig. 6.15, we show the reduction in capacitor size by CapMin. The baseline has one spike time for each MAC level. CapMin reduces the capacitor size by 14×, from 135.2 pF to 9.6 pF. We show this in the bar plot with $k = 14$, which can be used in a neuron circuit to achieve high accuracy in all five datasets. This also leads to lower latency by 14× as shown in Fig. 6.15, where the guaranteed response time (GRT) [Wei+21a] is used to measure latency. The energy reduction per MAC value computation is proportional to the capacitor size reduction, since the energy used in the capacitor is $\frac{1}{2}CV_{\text{th}}^2$, where $V_{\text{th}} = 0.225$ V.

6.2.4.3 *Variation Tolerance with CapMin-V*

When considering process variation the current has variations. This varies the charging speed of the capacitor. Consequently, the spike times may change, potentially leading to wrong MAC values. To extract the error model (matrix P_{map} in Sec. 6.2.3.2), we use a Monte-Carlo approach (1000 samples per spike time). The errors from current variation are injected during the inference of BNNs and the average test accuracy of three runs is reported in the plots (star marks) in Fig. 6.14. For all datasets the accuracy under variations is lower than without. The accuracy drops are expected, due to the probabilities for wrong mappings. Furthermore, the accuracy increases with smaller k. This is due to the procedure of CapMin. With smaller k, CapMin shifts

the important spike times to more reliable, slower spike times. In the case with e.g. $k = 32$, the most reliable (longest) spike time is mapped to the highest MAC value. In the case with $k = 16$, the large and small MAC values are removed. By this way, reliable spike times are mapped to important MAC values. The sweet spots for k and high accuracy under current variations are achieved for $15 \leq k \leq 12$ for Fashion, Kuzujishi, and SVHN, and for $15 \leq k \leq 14$ for CIFAR10. We conclude that CapMin alone leads to variation tolerant operation to some extent.

However, under variations the accuracy drops for smaller k compared to no variations. We apply CapMin-V (Alg. 6) to achieve higher tolerance to current variations. For this, we use the capacitor size at $k = 16$ (12.27 pF) with the corresponding $S_{\text{FIRE,min}}$ as a starting point and evaluate for different ϕ, so ϕ starts at $k = 15$ and ends at $k = 5$. In Fig. 6.14, applying CapMin-V (triangle plots) sustains higher accuracy for more points compared to only CapMin (star plots). In Fig. 6.15, the capacitance in CapMin-V is merely 28% and the latency 27% larger compared to CapMin. The capacitance (therefore energy) and latency are still 11× smaller than the baseline.

6.3 CONCLUSION

In this chapter, we proposed HW/SW Codesign methods that employ approximate analog-computing units to exploit the error tolerance of BNNs. Analog-computing units have excellent synergy with BNNs, as the former uses inherently imprecise representations for computations and the latter has outstanding error tolerance. The methods proposed in this chapter lead to significant reductions in area, energy, and latency. Therefore, the ideas are worth further investigation when designing efficient BNN systems in the future. The two sections in this chapter are summarized below.

In Sec. 6.1, we proposed a novel BNN inference scheme, called Local Thresholding Approximation (LTA), which approximates the global thresholdings in BNNs by local thresholdings. In BNN crossbar accelerators, this enables the use of only analog comparators through most of the execution, which significantly increases the interface circuit efficiency compared to the state of the art. However, employing the LTA without any countermeasures to the approximations results in a significant accuracy drop. To retain the original accuracy, we propose a training scheme that accounts for the degradation induced by the LTA, which consistently achieves high accuracy under approximations. Our results for two BNN models show that using the LTA reduces the area by factors of 42× and 54×, the energy by 2.7× and 4.2×, and the latency by 3.8× and 1.15×, compared to state-of-the-art crossbar-based BNN accelerators.

In Sec. 6.2, we proposed CapMin, a HW/SW Codesign method for capacitor size minimization in analog computing IF-SNNs. CapMin reduces the number of spike times needed in the HW based on MAC level occurrences in the SW. Furthermore, we proposed CapMin-V, which increases the tolerance to current variation. CapMin achieves a 14× reduction in capacitor size over the state of the art, while CapMin-V achieves variation tolerance at small cost. Our methods reduce area usage, energy, and latency, while increasing variation tolerance.

7

EFFICIENT TRAINING OF BNNS

In the vision of this dissertation, we propose to train BNNs on the edge. In this chapter, we explore how the memory-efficiency of the BNN training procedure can be achieved in order to enable BNN training with less memory, by which we aim bringing closer the vision of training BNNs on the edge.

Training of NNs can be efficiently performed on dedicated low-power accelerators in an on-chip setting, such as FPGAs [Zha+16; Luo+19] or ASICs [HTY17]. In addition to energy-efficiency of on-chip training, privacy issues and data transfer overheads are eliminated, since the data does not need to be transferred to the cloud for training [Kuk+19]. For these reasons, resource-efficient models, such as BNNs, should not only be executed but also be trained on the edge.

The idea of training BNNs on the edge device is relatively new. In 2021, Wang et al. [Wan+21] proposed to reduce the memory demand and the operations in BNN training by exploiting redundancies in gradients, batch normalization, and by using standard floating point (FP) encodings for optimizer data. A more general study in [Soh+19] provides a comprehensive overview of memory usage in NN training. As shown in [Soh+19], the major bottleneck in NN training is the memory usage. Classical training methods for BNNs use a large number of FP values during training and use the information stored in them to obtain a binarized model, thereof suffering from high memory usage as well. When one of the most memory-efficient training procedures for BNNs, the Binary optimizer (Bop) [Hel+19], is used, one momentum value encoded as FP per binary weight is stored. We show in Table 7.1 that BNN training methods such as Bop still suffer from high memory usage. This poses a challenge for the design of on-chip BNN training accelerators with limited on-chip memory and energy budgets.

To encode the momentum values in Bop for BNN training, a standard FP format [Hel+19] and the brain FP format [Wan+21] have been used. However, these encodings may use an excessive number of bits to encode the momentum values. Principled approaches to minimize the number of bits to encode the FP representation of the momentum values in Bop, targeting the number of bits in the exponent and mantissa, have not received much attention. Minimizing the number of bits in the FP encoding would lead to training of BNNs with reduced memory size, reduced overheads for data movements, and simpler components for computations, which are important steps to enable BNN training on the edge.

The key focus of in this chapter is to develop methods to obtain memory-efficient FP encodings for the momentum values in Bop, such that they require minimal memory space, without causing a significant loss in accuracy.

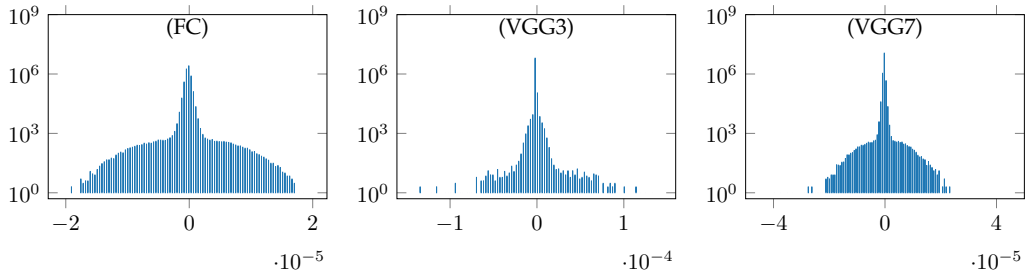


Figure 7.1: Histograms of all momentum values (after 10 epochs, 100 bins for values in FP32) for the three BNN models FC, VGG3, VGG7. The y-axis is in log scale.

This chapter is organized as follows. First, we present the Bop and the general FP format in Sec. 7.1 and Sec. 7.2. We then present the problem statement in Sec. 7.3. In Sec. 7.4, we investigate the impact of FP encodings with reduced number of bits on the momentum values of Bop and provide a theoretical proof for the cases in which momentum value updates are lost. Based on our theoretical observations, in Sec. 7.5, we formulate a metric, determining the number of unchanged momentum values due to the FP encoding. We develop an algorithm to obtain memory-efficient FP encodings for the momentum values, targeting tradeoffs in range (nr. of bits in exponent) and precision (nr. of bits in mantissa). In Sec. 7.6, we compare achieved accuracy and ratio of lost gradient updates of the FP encodings acquired from our method.

7.1 BINARY OPTIMIZER (BOP) IN BNN TRAINING

Recall that to train BNNs (Sec. 2.2.2), the backward pass has to be performed, i.e. the weights need to be updated based on the partial derivative of the loss $\mathcal{L}$ with respect to the corresponding binarized weights $\mathbf{W}_{bin}$, i.e., $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_{bin}}$. To obtain the partial derivatives, error backpropagation based on the chain rule has been widely employed for non-binarized NNs, as explained in Sec. 2.1.2. For BNNs, the study in [Hub+16] proposes to estimate the derivative of the binary activation function using the straight-through estimator, which is also explained in Sec. 2.2.2.

Instead of relying on the full-precision weights in the traditional BNN training method, to update the binary weights, the study in [Hel+19] proposes the binary optimizer (Bop) to directly flip the signs based on momentum signals. The momentum signals are exponential moving averages of the partial derivatives of the loss with respect to the binary weights. The method stores one momentum value $m_k \in \mathbf{M}_t$, encoded in floating point (FP) format, for each binary weight $w_k \in \mathbf{W}_{bin}$.

The algorithm for training BNNs with Bop [Hel+19] is recapped in Alg. 7. First W_{bin} and M_t are initialized (Line 1), where t counts the number of iterations. Then, in a loop for each epoch and batch, the gradients are computed (Line 4), the momentum values are updated (Line 5), and the signs of binary weights are flipped if the magnitude of

Algorithm 7: Training BNNs with the Binary Optimizer (Bop) [Hel+19].

Input: $\mathbf{W}_{bin}$, $\mathcal{L}$, (X_{train}, y_{train}), threshold τ, adaptivity rate γ, batch size B_s, E, B

1 Initialize $\mathbf{W}_{bin}$, $\mathbf{M}_t$
2 **for** *each epoch* $e = 0, \ldots, E$ **do**
3 **for** *each batch* $b = 0, \ldots, B$ **do**
  `// Compute gradients`
4 $\mathbf{G}_t \leftarrow \frac{1}{B_s} \frac{\partial \mathcal{L}}{\partial \mathbf{W}_{bin}} \sum_{(x,y) \in batch} \mathcal{L}(f_{\mathbf{W}_{bin}}(x), y)$
  `// Update momentum terms`
5 $\mathbf{M}_{t+1} \leftarrow (1-\gamma)\mathbf{M}_t + \gamma \mathbf{G}_t$
6 **for** *each pair* $m_k \in \mathbf{M}_{t+1}$, $w_k \in \mathbf{W}_{bin}$ **do**
7 **if** $|m_k| > \tau$ ***and*** $sgn(m_k) = sgn(w_k)$ **then**
8 $w_k \leftarrow -w_k$

the momentum value is larger than a predefined threshold τ and if the signs match (Lines 6-8). More details of the algorithm are in [Hel+19].

Bop has two main advantages over the other BNN optimization procedures. First, the optimization is tailored for BNNs, providing understanding and control of the optimization process, leading to higher accuracy. Secondly, Bop only needs to store one FP value per binary weight, which significantly cuts memory demand for BNN training. Other approaches, such as Adam with latent weights [Hub+16], need two or more FP values per binary weight. This makes Bop, to the best of our knowledge, the most memory-efficient BNN training method to date.

7.2 RECAP OF FLOATING POINT ENCODING

As explained above, floating point (FP) values are employed to encode the momentum values in Bop. The FP encoding is suitable for representing the momentum values, as most of the values are close to zero, and fewer values are far away from zero, as shown in the histograms in Fig. 7.1.

In FP encoding, values are mapped from a set (e.g. in $\mathbb{R}$) to a set with a finite number of values. A FP encoding uses one sign bit, a certain number of bits c for the exponent (determining the magnitude of values), and a certain number of bits p in the mantissa (determining the precision of values). The value encoded is computed by $(-1)^s \times 2^{exp} \times prec$. The value exp is defined by $exp = nn(C_{c-1} \ldots C_1 C_0) - bias$, where $nn(C_{c-1} \ldots C_1 C_0) = \sum_{i=0}^{c-1} C_i 2^i$ is the natural number of the binary representation of the exponent, which is represented as an unsigned integer, and the exponent $bias$ is used for conversion to a signed integer. The value $prec$ is obtained by $prec = 1 + \frac{nn(P_{p-1} \ldots P_1 P_0)}{2^p}$. The number of bits used in the encoding is $c + p + 1$, which we aim to minimize.

NN	FC	VGG3	VGG7
Binarized model	0.76 MB	0.83 MB	1.64 MB
Activations	0.34 MB	0.77 MB	10.0 MB
Optimizer (Bop)	23.36 MB	25.97 MB	51.95 MB

Table 7.1: Memory usage of BNN training with Bop, based on the categorization in [Soh+19]. Binarized model: 1 bit for each weight and bias, 32 bits for each batch normalization parameter. Activations: One bit per activation value (batch size 256). Optimizer: 32 bits for each BNN model parameter. See Table 3.2 for details of BNN models.

7.3 PROBLEM DEFINITION

We apply the categorization of data that needs to be stored in the memory throughout NN training process, as proposed by [Soh+19], to BNNs with Bop: (1) BNN model memory, consisting of the binary model parameters and the floating-point batch normalization parameters, (2) optimizer memory, consisting of the FP momentum values in Bop, and (3) activation memory, which consists of the binarized activations. In Table 7.1, we show that, out of the data that needs to be stored during the entire training procedure of BNNs, *the momentum values in Bop consume by far the largest portion of memory.* This leads us to the following problem statement.

Problem Definition: Given a set of labelled input data, the objective is to train a BNN with high accuracy. In this chapter, we focus on the problem of obtaining memory-efficient FP encodings of the momentum values in Bop, such that the number of bits needed for encoding in the exponent and mantissa (c and p respectively) are minimized, without causing a loss in accuracy.

7.4 IMPACT OF FLOATING POINT ENCODING IN BOP

To reason about the impact of memory-efficient FP encodings in BNN training, we consider a single momentum value $m_t \in \mathbf{M}_t$ in the BNN training process in Alg. 7. Its value is computed by $m_{t+1} \leftarrow (1-\gamma)m_t + \gamma g_t$. We denote the update to a momentum value in a BNN as

$$\Delta m_t = m_{t+1} - m_t = (1-\gamma)m_t + \gamma g_t - m_t = \gamma(m_t - g_t). \tag{7.1}$$

FP encodings have a finite number of levels, which we denote here as an ordered set $Q = \{q_0, q_1, \ldots, q_L\}$, where $q_v < q_{v+1}$ for $0 \leq v \leq L$. We assume nearest rounding scheme and use the midpoints between two FP levels as rounding thresholds. If an arbitrary FP encoding is applied to m_t, then we describe the FP value as $Q(m_t) = q_v$. The quantization scheme can be uniform (the distances between the quantization levels are all the same) or non-uniform (the distances between the levels can vary).

If a FP encoding as described in Sec. 7.2 is used to encode the momentum values, then information loss can occur. If the update Δm_t is too small, then it is lost ("quantized away"). This means, in cases of small Δm_t, the momentum value will not change.

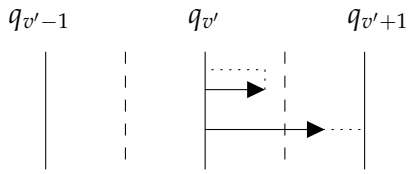


Figure 7.2: Visual presentation of the intuition behind Theorem 2. Only momentum updates that are large enough (pass the dashed line) lead to a change in the quantization level. If the updates are to small (arrow too short), then they are lost.

Thus, the update is not recorded and m_t keeps the same value. This leads us to the following theorem.

Theorem 2. *Assume a nearest-level FP encoding from $\mathbb{R}$ to an ordered set $Q = \{q_0, \dots, q_L\}$ and that a momentum value in the BNN training is encoded using Q, i.e. $m_t = q_v$. The training performs the update scheme $m_{t+1} \leftarrow m_t - \Delta m_t$, where the updated value m_{t+1} is encoded to values in Q after the update is completed. The encoded momentum value will not change the level after the update, i.e. $Q(m_{t+1}) = m_t$, if $\Delta m_t < \frac{q_{v+1}-q_v}{2}$ for increasing m_{t+1}, or if $\Delta m_t < \frac{q_v-q_{v-1}}{2}$ for decreasing m_{t+1} (see Fig. 7.2 for a visual presentation).*

Proof. Case $\Delta m_t < 0$, the momentum value is increased by the update ($m_t \uparrow$). Assume that m_t is encoded to a level v'. In the update step, m_t gets increased by Δm_t to an updated value m_{t+1}. Here, m_{t+1} can only be encoded to another level $q_{v'+r}$, if $|\Delta m_t| \geq \frac{|q_{v'+1}-q_{v'}|}{2}$, since m_t is always encoded to the nearest value in the FP encoding, as the nearest rounding scheme is used and the midpoints between two FP levels are taken as rounding thresholds. It follows that updates with $\Delta m_t < \frac{q_{v+1}-q_v}{2}$ do not cause m_{t+1} to change, thus $Q(m_t) = m_{t+1}$.

Case $\Delta m_t > 0$, the momentum value is decreased ($m_t \downarrow$). If $|\Delta m_t| < \frac{|q_v-q_{v-1}|}{2}$ then the momentum value will not change after the update. Here, m_{t+1} can only be encoded to another level $q_{v'-r}$, if $|\Delta m_t| \geq \frac{|q_{v'}-q_{v'-1}|}{2}$, as in the first case. □

From Theorem 2, we derive design principles for memory-efficient FP encodings of the momentum values. For effective training, it has to be ensured that momentum values change during training. To track the number of unchanged values, we use Theorem 2 to construct a metric. The ratio of momentum values that do not change their FP level is

$$U(Q, \mathbf{M}_t) = \frac{1}{|\mathbf{M}_t|} \sum_{m_t \in \mathbf{M}_t} \mathbf{1}\left[|\Delta m_t| < \frac{|Q(m_t) - q_{v^*}|}{2}\right], \tag{7.2}$$

where $\mathbf{1}[predicate]$ is the Iverson bracket, which is 1 if the condition *predicate* holds and 0 otherwise, $|\mathbf{M}_t|$ is the number of momentum values, $Q(m_t)$ the encoded momentum value, and q_{v^*} is the next value to $Q(m_t)$ in the encoding, depending on the sign of Δm_t (if negative then the subsequent higher level, if positive then the subsequent lower level). In the following, we refer to this metric as the U-metric.

Algorithm 8: Algorithm to find a memory-efficient encoding for momentum values.

Input: $U^{ref}_{0\to T}$, m^{absmax}_{sample}, τ, α, p_{step}, ϵ, T
Output: $Q_{b,c,p}$
// Calculate bias b **(1)**
$b = -\lfloor \log_2(m^{absmax}_{sample}) \rfloor + 1$
// Calculate nr. of bits c for exponent **(2)**
$c = \lceil \log_2(-\log_2(\alpha\tau) - b + 1) \rceil$
// Derive nr. of bits p for mantissa **(3)**
Initialize $p = -p_{step}$
do
  $p \leftarrow p + p_{step}$
  Apply Alg. Alg. 7 until $t = T$ using $Q_{b,c,p}$
  Extract $U_{0\to T}(Q_{b,c,p}, \mathbf{M})$ from Alg. Alg. 7
while $|U_{0\to T}(Q_{b,c,p}, \mathbf{M}) - U^{ref}_{0\to T}| > \epsilon$

7.5 MEMORY-EFFICIENT ENCODING OF MOMENTUM

To overcome the problem of losing momentum updates, one way is to increase the magnitude of the decay factor γ. For training BNNs with Bop, γ is a hyperparameter and it is initialized to a suitable value. Furthermore, a schedule is defined to reduce γ after a certain number of epochs (e.g. γ is halved every second epoch). However, if γ is chosen to be large, then the training is highly unstable, as reported in [Hel+19]. In practice, suitable initializations and hyperparameter schedules are jointly decided empirically with lots of efforts, and should not be modified after the search. Instead, we propose to design a memory-efficient FP encoding for the momentum values.

The key points for FP encodings with a minimal number of bits are (1) largest value in the encoding, i.e. exponent bias b, (2) the number of bits c in the power-of-two exponent, determining the range down to the smallest value, and (3) the number of bits p in the mantissa, determining the precision, i.e. the number of values to insert uniformly between the powers of two. For these three key points, we propose three design principles to guide the design of memory-efficient FP encodings for the momentum values in the Bop-optimization. In the following, we refer to the FP encoding as $Q_{b,c,p}$, and without loss of generality, leave out the signs of the values.

(1) Modify bias b: As values larger than the largest momentum value do not need to be represented, our aim is to clip the range. In the standard FP encoding, the largest number in the FP encoding is $2^{2^c-1-b} \times (2 - 2^{-p}) < 2^{2^c-b}$. Even if the largest momentum value is given, we can choose different combinations of b and c. We therefore change the representation of the exponent using $exp = nn(C_{c-1} \dots C_1 C_0) - \tilde{b} = -nn(C_{c-1} \dots C_1 C_0) - b$, where $\tilde{b} = 2nn(C_{c-1} \dots C_1 C_0) + b$, which corresponds to adding a constant to the exponent for converting the FP encoding to standard FP encodings. Using the notation $exp = -nn(C_{c-1} \dots C_1 C_0) - b$ allows us to directly calculate the upper bound of the floating point that can be represented by setting b

since $-nn(C_{c-1}\dots C_1C_0) \leq 0$. The largest number that can be represented by the FP format is $2^{-b} \times (2-2^{-p}) < 2^{-b+1}$. Therefore, b should be chosen such that

$$2^{-b+1} > m_{sample}^{absmax} \Leftrightarrow b < -\log_2(m_{sample}^{absmax}) + 1, \tag{7.3}$$

where m_{sample}^{absmax} is the largest absolute value observed in a sample of the momentum values M_{sample}. If b is chosen to be too small, there will be no effects other than potentially unused large levels. If b is chosen to be too large, then large values will be mapped to smaller levels. This may hinder the training, since Bop relies on large $|m_t|$ to flip signs (see Line 7 in Alg. 7).

(2) Nr. of bits in exponent: Sufficient range is needed for Bop to operate correctly. When the bias b is already set, then the choice of the number of bits c in the exponent ($exp = -nn(C_{c-1}\dots C_1C_0) - b$) determines the smallest values that can be represented in the encoding. If c is chosen to be too large, there will be no issue on accuracy, rather than potentially unused small levels. If c is chosen to be too small, then small values will be mapped to large levels, which may severely disturb training, since Bop relies on comparisons of momentum value magnitudes to τ (Alg. 7, Line 7), which is a known hyperparameter in Bop. A lower bound for c is derived analytically with

$$\tau = 2^{-2^c+1-b} \Leftrightarrow c = \lceil \log_2(-\log_2(\tau) - b + 1) \rceil. \tag{7.4}$$

To calculate suitable c for the FP encoding, it is necessary to further include certain momentum values that are smaller than τ. Otherwise, these momentum values may be discarded due to the configuration of τ. We therefore introduce another coefficient α by replacing τ in Eq. (7.4) with $\alpha\tau$ for some $0 < \alpha < 1$. With the help of α, the number $c = \lceil \log_2(-log_2(\alpha\tau) - b + 1) \rceil$ of bits is calculated such that the number of FP levels below τ can be configured for the FP encoding.

(3) Nr. of bits in mantissa: The number of bits in the mantissa, referred to as p, determines the number of levels that are inserted uniformly between the power-of-two levels. If p is chosen too small, then the condition $|\Delta m_t| < \frac{|Q(m_t) - q_{v^*}|}{2}$ in Eq. (7.2) will be true for a large amount of the momentum values, which may severely hinder training, since a large fraction of the momentum updates are lost. For this reason, it is required to insert levels between the powers of two, such that the updates are not lost due to the encoding. However, it would be inefficient to derive p by incrementing it successively. For each p, training until full convergence with the encoding $Q_{b,c,p}$ would need to be performed. The search for p would terminate once same accuracy is achieved as in a reference encoding. Instead, we propose an efficient way for obtaining p, using the U-metric, where training is only needed to be performed with $t = T$ training iterations in Alg. 7, regardless of the accuracy. With a reference $U_{0\to T}^{ref}$, obtained from a reference encoding using which a high test accuracy can be achieved, it needs to be ensured that

$$|U_{0\to T}(Q_{b,c,p}, \mathbf{M}) - U_{0\to T}^{ref}| < \epsilon \tag{7.5}$$

holds, where $\epsilon \in (0, 1)$ describes the required approximation of $U_{0\to T}(Q_{b,c,p}, \mathbf{M})$ to $U_{0\to T}^{ref}$, and the notation $0 \to T$ describes that $U_{0\to T}$ includes all U-values from training

iterations 0 to T (the comparison is vectorized over all U-values). When the condition in Eq. (7.5) holds, it is ensured that enough momentum values change their levels for the training to converge well.

Based on the three design principles, we summarize our method for obtaining memory-efficient FP encodings for the momentum values in Alg. 8. In our algorithm, the bias, exponent, and mantissa are designed such that the number of bits in the FP encoding is minimized, while ensuring that Bop can operate correctly. The algorithm has three main steps, corresponding to the three design principles. (1) A suitable bias b based on the distribution of momentum values is calculated, see Line 1. We calculate b based on Eq. (7.3). Then, the encoding is clipped, such that larger values than the observed maximum are not included in the encoding. Furthermore, we ensure that large FP levels are available in the encoding, which are important in Bop, since it relies on large moments to flip signs. (2) The number of required bits c in the exponent is calculated based on Eq. (7.4) with given b, τ, and α, see Line 2. To ensure that FP levels below τ are included, we multiply τ by a factor $\alpha \in (0, 1)$, e.g. $\alpha = 0.1$ to add approximately three power-of-two levels below τ. By this, we ensure that the comparison to τ in Bop is correct, while avoiding to include small FP levels that may not be necessary. (3) The steps for acquiring the number of bits p in the mantissa require to run Alg. 7 with different p in $Q_{b,c,p}$. To record the $U_{0\to T}(Q_{b,c,p}, \mathbf{M})$, a training process with Alg. 7 using $Q_{b,c,p}$ is conducted until training iteration $t = T$. Bits in the mantissa are added, to reach a similar U-value as in the reference encoding. When the U-metrics are close enough regarding ϵ, the algorithm terminates. This step ensures that similarly many momentum values change in the obtained encoding as in the reference encoding, while unnecessary precision is avoided in the obtained FP encoding. Note that the number of mantissa bits is decided solely based on the U-metrics, without the need to train until convergence and without comparing achieved accuracy. The search for the number of mantissa bits is time-efficient because it is independent from accuracy.

7.6 EXPERIMENTS

In Sec. 7.6.1, we present the experiment setup including details of the training, hyperparameter settings, and the details of setting up Alg. 8 for evaluation. We present the experiment results in Sec. 7.6.2.

7.6.1 *Experiment Setup*

In the following, we first explain which hyperparameters we use in the training and then how we set up the memory-efficient encoding with Alg. 8.

Training: We run Bop for optimizing BNNs in a PyTorch-based framework, based on the framework described in Ch. 3. We simulate the FP encoding by applying lookup tables that map any value to a value in a desired quantization set, such as a custom

Encoding	Range	Precision	Realization
PT6	o	-	1 sign, 5 exp.
FP8	o	o-	1 sign, 5 exp., 2 mant.
FP10	o	o-	1 sign, 5 exp., 4 mant.
FP12	o	o	1 sign, 5 exp., 6 mant.
FP16b	+	o	1 sign, 8 exp., 7 mant.
FP32	+	+	1 sign, 8 exp., 23 mant.

Table 7.2: FP encodings evaluated in Alg. 8. For easy reference, the following encodings are included: FP16b and FP32. +: High, o: Mid, -: Low.

floating point format in our case. This is achieved by performing the quantization on the momentum tensors during training with custom CUDA kernels. We use the FC, VGG3, and VGG7 models described in Table 3.2, for the data sets Fashion, SVHN, and CIFAR10, shown in Table 3.1. The batch size is 256 for all models. In all models, we use the threshold $\tau = 10^{-8}$, initial decay $\gamma = 10^{-3}$, and decay factor $\eta_{decay} = 0.125$, which is multiplied with γ every 10th epoch for FC and VGG3, and for VGG7 we apply $\eta_{decay} = 0.5$ every 25th epoch. For the batch normalization parameters, we use Adam optimization as proposed by [Hel+19], with an initial learning rate of 10^{-2} for FC and VGG3, and 10^{-3} for VGG7. We halve the learning rate every 5th epoch for FC and VGG3, and halve it every 50th epoch for VGG7. For each model and FP encoding case, we train 50 epochs for FC and VGG3, and 200 epochs for VGG7. For each case, we repeat the experiments five times for FC and VGG3, and three times for VGG7.

Memory-Efficient Encoding: As baselines, we use the FP16b (brain FP format), used in modern processing units to accelerate NN training [Kal+19] and are used in the state-of-the-art for encoding Bop momentum values for efficient BNN training in [Wan+21], and FP32. To acquire memory-efficient encodings for the momentum values with Alg. 8, we sample values for $U^{ref}_{0\to T}$ using FP16b. When running Alg. 8 for different FP encodings, we simulate the use of FP encodings by applying them to the momentum values after each update of the momentum values.

7.6.2 *Experiment Results*

We run Alg. 8 to obtain memory-efficient encoding schemes for the momentum values, with $\tau = 10^{-8}$ (hyperparameter in Bop), $\alpha = 10^{-1}$ (to add approximately three power of two levels below τ), $p_{step} = 2$, and $\epsilon = 0.25$ (such that the majority of values can change their FP levels). We use use these parameters as an illustration of the intermediate steps of the Alg. 8. We now follow the three main steps of Alg. 8 for the three models: (1) For the bias, Alg. 8 calculates $b = 14$ as the highest value among all three models. We use it as the b for all models. To provide context for this choice, we plot histograms of the momentum values in Fig. 7.1. (2) With given $b = 5$, $c = 5$ is calculated as number of required bits in the exponent to include values smaller than

$\alpha\tau$ in the FP encoding. (3) To acquire the number of bits in the mantissa, the algorithm starts with PT6, the encoding with 1 sign bit and 5 exponent bits. Then, for each subsequent iteration, $p_{step} = 2$ mantissa-bits are added. The algorithm stops when $p = 6$, i.e. the condition of the while loop (Line 8) is satisfied. We do not conduct further testing of other configurations. We plot the U-metric and achieved accuracy in Fig. 7.3. The list of the encodings tested in Alg. 8 with their respective properties (and the encodings FP16b and FP32 for easy reference) are shown in Table 7.2.

We plot the U-metric over the training epochs in Fig. 7.3, to explain why certain encoding schemes perform better than others. For all U-curves, the move towards $U = 1$ is noticeable, because the decay factor γ is decreased every 10 (50 for VGG7) epochs. We observe that the encodings with no or few mantissa bits, i.e. PT6 and FP8, have U-values close to 1 early in the training, which severely hinders training due to unregistered updates. This is also reflected in the achieved accuracy in all cases in Fig. 7.3. The case for FP10 has lower U-values than in the previous encodings, but a large fraction of values still do not change their level. For FP12, the U-metric is significantly lower, the difference in the U-metrics with $\epsilon = 0.25$ compared to FP16b is achieved.

Please note that other exponent-only formats than PT6 with more bits for the exponent do not lead to better results, since exponent-only formats merely increase the number of smaller values, which cannot decrease the number of lost momentum updates for momentum values that matter (i.e. that are larger than $\tau = 10^{-8}$). Therefore other exponent-only formats are omitted for brevity.

In summary, we observe that with Alg. 8, optimized encodings for the momentum-values can be obtained, based on samples of momentum values, $\alpha\tau$, and a reference encoding. In our experiments, we observe a correlation between the U-metric and the achieved accuracy. The closer the U-metric of the FP encoding in Alg. 8 to the reference encoding, the closer the achieved accuracy of the FP encoding in Alg. 8 to the reference encoding. In Table 7.1, we summarize the experiment results. We report test accuracy and the memory saving factors obtained by the FP encodings in Alg. 8 that achieve high accuracy, i.e. FP10 and FP12. The entries for FP16b and FP32 are not acquired from Alg. 8, they shown for reference.

7.7 DISCUSSION: GRADIENT COMPUTATIONS WITH CUSTOM FP FORMATS

In the following, we evaluate the impact of using the custom FP encodings from Alg. 8 on the resource efficiency of transprecision (i.e. with configurable number of bits in the exponent and mantissa) Floating Point Units (FPUs) in Sec. 7.7.1. Then, in Sec. 7.7.2, we present the future work of using the custom FP encodings for the gradient computations and the errors caused by them in the computations.

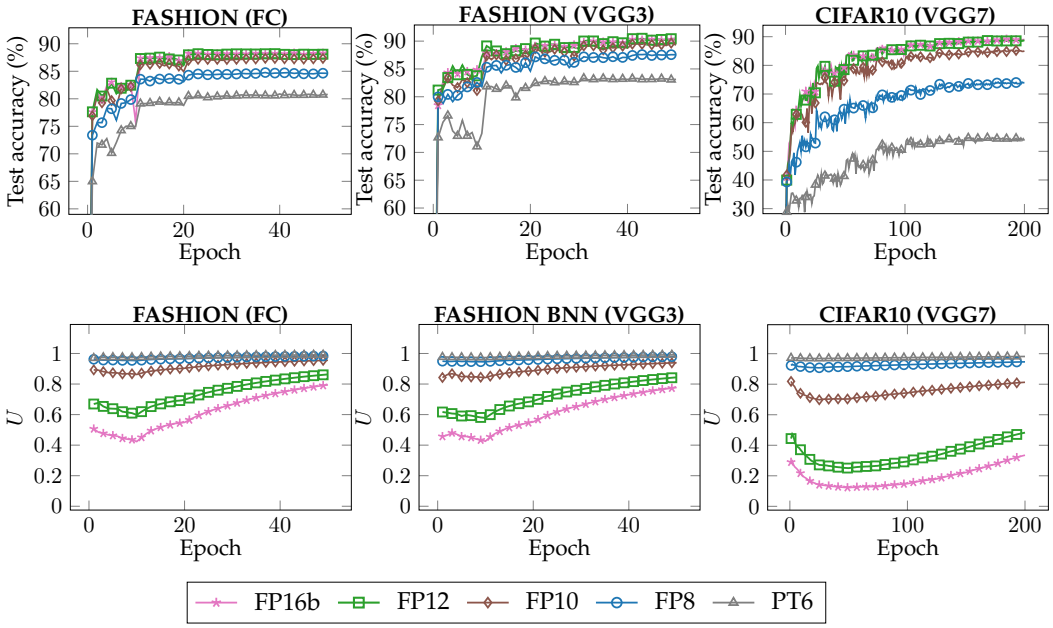


Figure 7.3: Achieved accuracy (top row, the higher the better) and the U-values (bottom row, the lower the better) of the FP encodings in Alg. 8. FP16b is shown here for reference. It is not constructed by Alg. 8.

Encoding Type	Fashion FC accuracy	Fashion VGG3 accuracy	CIFAR10 VGG7 accuracy	Memory saving factor		
				FC	VGG3	VGG7
FP10	**87.35** (0.36, 0.20)	**89.30** (0.69, 0.67)	85.02 (0.15, 0.18)	**2.89**	**2.83**	2.28
FP12	**88.10** (0.13, 0.22)	**90.42** (0.23, 0.24)	**88.56** (0.14, 0.14)	**2.47**	**2.43**	**2.04**
FP16b	88.31 (0.13, 0.14)	90.11 (0.35, 0.66)	89.07 (0.18, 0.14)	1.91	1.89	1.69
FP32	88.23 (0.23, 0.27)	90.48 (0.21, 0.19)	89.14 (0.07, 0.12)	1.00	1.00	1.00

Table 7.3: Avg. and (avg.-min., max.-avg.) accuracy (with observed min. and max.) for the different FP encodings in Alg. 8 on the test sets at the end of the training procedure. We also report the total memory saving factors. The cases with minimal (approx. 0-1%) accuracy degradation compared to the 32-bit case are in bold. FP16b and FP32 are shown for reference, they are not constructed by Alg. 8.

7.7.1 *Impact of Custom FP Formats on FPU Efficiency*

For effective use of the custom FP formats $Q_{b,c,p}$, the gradients g_t and subsequently the momentum values m_{t+1} should be computed using the custom FP formats. To investigate the effects of reduced FP formats in terms of area, energy, latency on the resource efficiency of floating point units (FPUs), we use the open-source transpecision FPU design called FPnew from [Mac+21]. For FPU evaluations, we generate instances of the FPUs with the FP formats FP32, FP16b, FP12, and FP10 (the last two formats

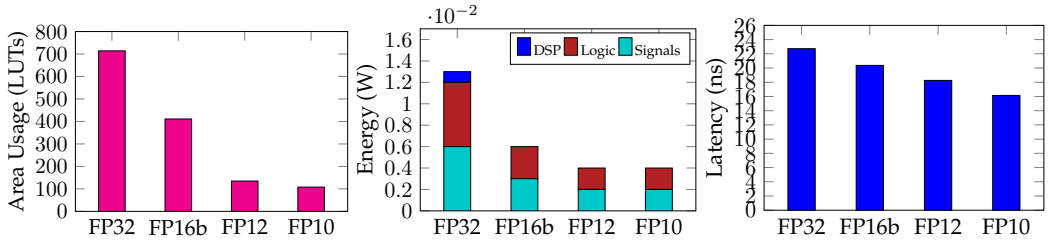


Figure 7.4: Area, energy, and latency evaluations of the transprecision FPU from [Mac+21]. The FPUs are configured with the custom FP formats derived from Alg. 8.

are derived by Alg. 8). We use Vivado to synthesize the FPU for the Zynq UltraScale+ ZCU104 Evaluation Board FPGA and show the results for area, energy, and latency in Fig. 7.4. The results are summarized in the following.

Area: We use Lookup Tables (LUTs) to estimate the area usage, as they are the most important elements to build computing units in FPGAs. The results show that FPUs configured with FP formats with a smaller number of bits use significantly less LUTs than FPUs that realize formats with a higher number of bits. For example, the FPU with format FP16b uses 57% of the LUTs that a FP32 FPU uses. Furthermore, the FPUs with FP12 or FP10 use only 19% and 15% respectively of the LUTs that a FP32 FPU uses. **Power**: For the energy usage we rely on the power report of Vivado to evaluate the amount of power used by the FPGA for one FPU. As the FPGA board uses a static 0.592 W of power, we subtracted this amount from all results to get the dynamic power used by the FPU. The dynamic power is separated into Signals, Logic, and Digital Signal Processing (DSP). The energy measurements show a large difference between the power usage of FPUs using FP32 and FP16b with 0.013 W compared to 0.006 W. Between FP16b and FP12, the difference is less significant with 0.006W compared to 0.004W, while there is no measurable difference between FP12 and FP10. **Latency**: For measuring the latency, we use the delay of the longest path in the timing report of Vivado, which can be used to set the clock cycle. In general, the latency improvements are small, in the scale of a few percentage points, and not as significant as in area and power.

7.7.2 *Discussion: Using Custom FP Formats for Gradient Calculations*

We have not yet discussed the impact on the result quality in backward pass computations when calculating with reduced FP formats. To assess the impact, we first define some useful notation. Assuming that the biases of the FP formats are the same, $Q_{FP16} \subset Q_{FP32}$ holds, since Q_{FP16} has elements from Q_{FP32}, but not all of them. In general, $Q_{b,c',p'} \subset Q_{b,c,p}$ holds when $c' < c$ and $p' < p$.

Based on this, an intuitive approximation is

$$Q_{b,c,p}((m_{t+1})_{FP32}) \approx (m_{t+1})_{b,c,p}, \tag{7.6}$$

where the right side $(.)_{enc}$ states that the encoding enc is used to represent the values and also perform the computations. The left side is the same as in the experiments in Sec. 7.6.2, where m_{t+1} is calculated using a 32 bits FP format, and then the resulting m_{t+1} is quantized using $Q_{b,c,p}$. The intuitive idea behind Eq. (7.6) is that the results of multiplications and additions using $Q_{b,c,p}$ (computing with the encoding, i.e. the FPU that realizes $Q_{b,c,p}$) is approximately the same compared to the case of calculating using 32-bit FP formats and then quantizing using $Q_{b,c,p}$. Additions and multiplications here serve as a model for operations in the backward pass.

Specifically, in future work we plan to show that for the operation between two values v_1 and v_2 that $MULT_{b,c',p'}(v_1, v_2) \approx (MULT_{b,c,p}(v_1, v_2))_{b,c',p'}$ and $ADD_{b,c',p'}(v_1, v_2) \approx (ADD_{b,c,p}(v_1, v_2))_{b,c',p'}$, with $c < c'$ and $p < p'$. However, since many additions and especially multiplications are performed in the backward pass due to the chain rule, these statements also need to be extended to cover chains of operations.

7.8 CONCLUSION

We proposed a method to obtain memory-efficient floating point (FP) encodings for the momentum values in the BNN training using Bop. Our method is based on the hyperparameters of Bop and a metric, for which we prove that it tracks the number of unchanged values due to reduced FP encodings. Our result show that FP encodings with less number of bits can be obtained than the encodings in the state-of-the-art, leading to a significant reduction in memory usage for BNN training. This enables BNN training on the edge with significantly lower memory requirements, which inevitably leads to lower energy, latency, and area consumption, since the memory subsystem is the most critical bottleneck in NN training.

8 CONCLUSION AND OUTLOOK

In the following, we provide a summary of this dissertation in Sec. 8.1 and discuss the future work in Sec. 8.2.

8.1 SUMMARY

This dissertation has proposed a vision for efficient future intelligent systems that are comprised of robust BNNs operating with approximate memory and approximate computing units, while being able to be trained on the edge. We have explored this vision in four chapters of this dissertation.

In Ch. 4, we proposed bit error tolerance metrics for the hidden-layer-neuron level and the output-layer level. We formally proved that our metrics measure the maximum number of any bit flips that can be tolerated without a change of the BNN prediction. Based on these metrics and the hinge loss for maximum margin classification in SVMs, we proposed the modified hinge loss (MHL) for optimizing the bit error tolerance of BNNs. Our experimental results show that the BNNs trained with the MHL achieve higher bit error tolerance and accuracy compared to BNNs trained with the classical bit flip injection method.

In Ch. 5, we explored two scenarios that assume approximate FeFET memory for the BNNs. In Sec. 5.2, we first analyzed the effects of variable temperature on FeFET memory and proposed an asymmetric bit error model that exhibits the relation between temperature and bit error rates. We then evaluated the impact of the asymmetric temperature bit errors of FeFET on BNN accuracy when no countermeasures are used and showed that the accuracy can drop can be unacceptable. To deploy BNNs with high accuracy using FeFET memory despite the temperature effects, we proposed two countermeasures to the bit errors: (1) Bit flip training while taking the asymmetry into account and (2) a bit error rate assignment algorithm (BERA). With these methods, the BNNs achieve bit error tolerance for the entire range of operating temperature. In Sec. 5.3, we proposed to use FeFET-based LiM in the form of XNOR gates for BNN inference. To alleviate the latency bottleneck from the FeFET-based LiM, we investigated the impact from FeFET-based LiM errors on the inference accuracy of BNNs. We demonstrated how the inherent latency tradeoff of FeFET-based LiM can increase the overall latency of BNNs significantly. We showed that a significant decrease in latency is achieved when applying design-time methods through training with errors or when using the run-time methods.

In Ch. 6, we explored two approximate computing approaches for analog computing BNN accelerators. In Sec. 6.1, we proposed the BNN inference scheme coined Local Thresholding Approximation (LTA), which approximates the global thresh-

oldings in BNNs by local thresholdings. In BNN crossbar accelerators, this enables the use of only analog comparators for most of the execution, which significantly increases the interface circuit efficiency compared to the state of the art. Our results for two BNN models showed that using the LTA reduces the area, energy, latency by large factors when compared to state-of-the-art crossbar-based BNN accelerators. In Sec. 6.2, we considered another analog computing based processing scheme for BNNs, the IF-SNNs, which use large capacitors. We therefore proposed CapMin, a method for capacitor size minimization IF-SNNs. CapMin achieves the capacitor size minimization by reducing the number of spike times needed in the HW based on MAC level occurrences in the SW. Additionally, we proposed CapMin-V, a method which increases the tolerance to current variation. CapMin achieves a significant reduction in capacitor size over the state of the art, while CapMin-V achieves variation tolerance at small cost.

In Ch. 7, we proposed a method to obtain memory-efficient floating point (FP) encodings for the momentum values in the BNN training using Bop. Our method is based on the hyperparameters of Bop and a metric, for which we prove that it tracks the number of unchanged values due to reduced FP encodings. Our results show that FP encodings with less number of bits can be obtained than the encodings in the state of the art, leading to a significant reduction in memory usage for BNN training. This enables BNN training on the edge with significantly lower memory requirements, inevitably leading to lower energy, latency, and area consumption, since the memory subsystem is the most critical bottleneck in NN training.

8.2 FUTURE WORK

In the following we discuss the possible directions for how the work presented in this dissertation could be extended in the future.

Error Tolerance Optimization of BNNs: In Ch. 4, we believe that we have developed a fundamental understanding of the bit error tolerance for BNNs and that it provides a cornerstone for the exploration of other NN models. Despite the limitation of using only binary values, the concept of margins of individual neurons and the margins of the output layer can be potentially extended to any NN model with higher precision weights. In these evaluations, alternative ways of margin maximization in the output layer should be investigated, since some NN models do not rely on classifications but on direct return values such as coordinates (e.g. when predicting bounding boxes for objects). Specifically, instead of the last layer, the hidden-layer metric should be explored in more detail, for which still a consistent and provable metric does not exit in the literature. This may be possible by combining the error tolerance of NNs with explainability methods (XAI).

Approximate FeFET Memory for BNNs: For FeFET, in Sec. 5.2, we have considered temperature behavior and latency for reads. However, the energy and latency of the cell programming is also important, especially if the FeFET-based memory cannot hold all the weight data and needs to be rewritten, which will be considered in

the future. Furthermore, temperature issues due to temperature cycling are also interesting to explore in the future.

Approximate Computing Units for BNNs: For analog-based computing, which we explored in Ch. 6, it is important to also explore the maximum number of XNOR gates that can fit into one column, since it plays a key role in the efficiency of analog-based computing. Specifically, the tradeoff between the number of analog states in a computing column and the noise level in the column is an interesting future work, especially considering the interplay with the ADC. Furthermore, one limitation is that only BNNs are considered. We plan to extend the two methods to multi-bit NNs, however, analog computing for higher precision is a challenge due to the larger number of required analog states. For the work in Sec. 6.1, one possible future direction to improve the LTA accuracy is to explore methods that attempt to automatically find local thresholds and majority vote shifts in the LTA. This can be done in a fine-grained manner, i.e. window or neuron based, or in a more coarse-grained manner, i.e. on the layer level. However, automatically finding or training the local thresholds and majority vote shifts introduces many additional parameters, as there can be many thousands or more neurons in a BNN, while the number of local thresholds is approximately one order of magnitude larger than the total number of neurons in the BNN. Furthermore, the optimization of thresholds also competes with the function of the batch norm layer, as the traditional (global) thresholds are derived from the batch norm parameters, without which BNNs are reported to achieve poor training performance [SBN19]. For the work in Sec. 6.2, an interesting extension would be to simulate an entire system, i.e. connecting the buffer memories for the inputs and weights, control circuits, and adders to accumulate subcomputations. Then, the impact of the capacitor reduction could be evaluated on the system level instead of the circuit level.

Training BNNs on the Edge: For enabling the training of BNNs on the edge, in Ch. 7, we have explored the reduction of the FP format of the momentum values. We have not yet discussed the impact on the result's quality of the backward pass when calculating with reduced FP formats. Furthermore, to increase the efficiency of the backward pass, some computations in the backward pass may be redundant or without much impact (e.g. updates are always too small). A suitable extension would be to selectively drop computations of the backward pass, either randomly or in a structured manner.

Note on the BNN Architectures Used: For the proof-of-concept experiments of the contributions in this dissertation, four types of BNNs were used, as presented in Ch. 3. The BNNs perform image classification and have fully connected, convolutional, and skip connections, which are typical building blocks used in resource-constrained NNs. Image classification is also a typical BNN use case in which high-performing models tailored for resource-constrained inference are used. Although the BNNs and the datasets used here are resonable choices for proof-of-concept research, they are limited to image classification. Further evaluations on other use cases, such as language and audio processing, would strengthen the research contributions. To the best of our

knowledge, BNNs or multi-bit NNs trained for these use cases also exhibit bit error tolerance, which can be exploited on the hardware or software level for efficiency. However, complex tasks such as language processing may require more bits than in the binary case, as their binarization leads to significant accuracy costs (at the time of writing the dissertation, this may change for the better in the future). Still, with theory on our side, we know that BNNs can approximate any function that multi-bit NNs can. The challenge is to find BNN architectures that achieve high accuracy, while at the same time being more efficient than multi-bit NNs. After high-performing BNN architectures are built, then they can be optimized for error tolerance, which can then in turn be exploited for efficiency.

BIBLIOGRAPHY

[Agr+18] Amogh Agrawal, Akhilesh Jaiswal, Chankyu Lee, and Kaushik Roy. "X-SRAM: Enabling In-Memory Boolean Computations in CMOS Static Random Access Memories." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2018).

[Ali+18] Milad Alizadeh, Javier Fernández-Marqués, Nicholas D. Lane, and Yarin Gal. "An Empirical study of Binary Neural Networks' Optimisation." In: *International Conference on Learning Representations (ICLR)*. 2018.

[And+18a] Kota Ando et al. "BRein Memory: A Single-Chip Binary/Ternary Reconfigurable in-Memory Deep Neural Network Accelerator Achieving 1.4 TOPS at 0.6 W." In: *IEEE Journal of Solid-State Circuits* (2018).

[And+18b] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. "YodaNN: An Architecture for Ultralow Power Binary-Weight CNN Acceleration." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2018).

[Arm+22] Giorgos Armeniakos, Georgios Zervakis, Dimitrios Soudris, and Jörg Henkel. "Hardware Approximate Techniques for Deep Neural Network Accelerators: A Survey." In: *arXiv:2203.08737* (2022).

[Ban+18] Ron Banner, Itay Hubara, Elad Hoffer, and Daniel Soudry. "Scalable Methods for 8-Bit Training of Neural Networks." In: *Neural Information Processing Systems (NIPS)*. 2018.

[Ban+21] Tom Bannink, Arash Bakhtiari, Adam Hillier, Lukas Geiger, Tim de Bruin, Leon Overweel, Jelmer Neeven, and Koen Helwegen. "Larq Compute Engine: Design, Benchmark, and Deploy State-of-the-Art Binarized Neural Networks." In: *arXiv:2011.09398* (2021).

[BLC13] Yoshua Bengio, Nicholas Léonard, and Aaron C. Courville. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation." In: *arXiv:1308.3432* (2013).

[Bet+18] Joseph Bethge, Marvin Bornstein, Adrian Loy, Haojin Yang, and Christoph Meinel. "Training Competitive Binary Neural Networks from Scratch." In: *arXiv:1812.01965* (2018).

[Bet+19] Joseph Bethge, Haojin Yang, Marvin Bornstein, and Christoph Meinel. "Back to Simplicity: How to Train Accurate BNNs from Scratch?" In: *arXiv:1906.08637* (2019).

[Bou+17] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. "Emerging NVM: A Survey on Architectural Integration and Research Challenges." In: *ACM Trans. Des. Autom. Electron. Syst.* (2017).

[Bou+22] Wadii Boulila, Maha Driss, Eman Alshanqiti, Mohamed Al-Sarem, Faisal Saeed, and Moez Krichen. "Weight Initialization Techniques for Deep Learning Algorithms in Remote Sensing: Recent Trends and Future Perspectives." In: *Advances in Intelligent Systems and Computing*. 2022.

[BMT20] Adrian Bulat, Brais Martinez, and Georgios Tzimiropoulos. "BATS: Binary ArchitecTure Search." In: *arXiv:2003.01711* (2020).

[BT19] Adrian Bulat and Georgios Tzimiropoulos. "XNOR-Net++: Improved Binary Neural Networks." In: *arXiv:1909.13863* (2019).

[Bus+18] S. Buschjager, K. Chen, J. Chen, and K. Morik. "Realization of Random Forest for Real-Time Evaluation through Tree Framing." In: *International Conference on Data Mining (ICDM)*. 2018.

[BJBP20] Sebastian Buschjäger, Katharina Morik Jens Buß, and Lukas Pfahler. "On-Site Gamma-Hadron Separation with Deep Learning on FPGAs." In: *Machine Learning and Knowledge Discovery in Databases (ECML PKDD)*. 2020.

[Bus+20] Sebastian Buschjäger, Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Christian Hakert, Katharina Morik, Rodion Novkin, Lukas Pfahler, and Mikail Yayla. "Towards Explainable Bit Error Tolerance of Resistive RAM-Based Binarized Neural Networks." In: *arXiv:2002.00909* (2020).

[Bus+21] Sebastian Buschjäger, Jian-Jia Chen, Kuan-Hsun Chen, Mario Günzel, Christian Hakert, Katharina Morik, Rodion Novkin, Lukas Pfahler, and Mikail Yayla. "Margin-Maximization in Binarized Neural Networks for Optimizing Bit Error Tolerance." In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2021.

[C+22] Srinivasan C, Sridhar P, Hari Priya V, and Swathi S. "A TinyML based Residual Binarized Neural Network for real-time Image Classification." In: *International Conference on Electronics, Communication and Aerospace Technology*. 2022.

[Cif] *CIFAR10 Dataset, https://www.cs.toronto.edu/ kriz/cifar.html*. Accessed 2023-09-01. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[CP23] Ahmet Enis Cetin and Hongyi Pan. "Hybrid Binary Neural Networks: A Tutorial Review." In: *VLSI Test Symposium (VTS)*. 2023.

[Che+20a] Gang Chen, Shengyu He, Haitao Meng, and Kai Huang. "PhoneBit: Efficient GPU-Accelerated Binary Neural Network Inference Engine for Mobile Phones." In: *Design, Automation Test in Europe Conference Exhibition (DATE)*. 2020.

[Che+20b] Jia Chen, Jiancong Li, Yi Li, Xiangshui Miao, J Chen, J Li, Y Li, and X. S. Miao. "Multiply accumulate operations in memristor crossbar arrays for analog computing." In: *Journal of Semiconductors* (2020).

[Che+15] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems." In: *arXiv:1512.01274* (2015).

[Che+18] X. Chen, X. Yin, M. Niemier, and X. S. Hu. "Design and optimization of FeFET-based crossbars for binary convolution neural networks." In: *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 2018.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks." In: *International Symposium on Computer Architecture (ISCA)*. 2016.

[Chi+16] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory." In: *International Symposium on Computer Architecture (ISCA)*. 2016.

[CGC21] Jeong Hwan Choi, Young-Ho Gong, and Sung Woo Chung. "A System-Level Exploration of Binary Neural Network Accelerators with Monolithic 3D Based Compute-in-Memory SRAM." In: *MDPI Electronics* (2021).

[CBD15a] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "BinaryConnect: Training Deep Neural Networks with binary weights during propagations." In: *Neural Information Processing Systems (NIPS)*. 2015.

[CBD15b] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "Training deep neural networks with low precision multiplications." In: *arXiv: 1412.7024* (2015).

[Cyb89] G Cybenko. "Approximation by superposition of sigmoidal functions." In: *Mathematics of Control, Signals, and Systems* (1989).

[Dav+23] Abhilasha Dave, Fabio Frustaci, Fanny Spagnolo, Mikail Yayla, Jian-Jia Chen, and Hussam Amrouch. "HW/SW Codesign for Approximation-Aware Binary Neural Networks." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2023).

[DSM20] Joydeep Kumar Devnath, Neelam Surana, and Joycee Mekie. "A Low-Voltage Split Memory Architecture for Binary Neural Networks." In: *International Symposium on Circuits and Systems (ISCAS)*. 2020.

[Din+19] Ruizhou Ding, Ting-Wu Chin, Zeye Liu, and Diana Marculescu. "Regularizing Activation Distribution for Training Binarized Deep Networks." In: *Computer Vision and Pattern Recognition (CVPR)*. 2019.

[DLS18] Yukun Ding, Jinglan Liu, and Yiyu Shi. "On the Universal Approximability of Quantized ReLU Neural Networks." In: *arXiv:1802.03646* (2018).

[Don+18] Marco Donato, Brandon Reagen, Lillian Pentecost, Udit Gupta, David Brooks, and Gu-Yeon Wei. "On-Chip Deep Neural Network Storage with Multi-Level ENVM." In: *Design Automation Conference (DAC)*. 2018.

[DSS20] Yuxuan Du, Xinchao Shang, and Weiwei Shan. "An Energy-Efficient Time-Domain Binary Neural Network Accelerator with Error-Detection in 28nm CMOS." In: *Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2020.

[Dut+20] Sourav Dutta, Clemens Schafer, Jorge Gomez, Kai Ni, Siddharth Joshi, and Suman Datta. "Supervised Learning in All FeFET-Based Spiking Neural Network: Opportunities and Challenges." In: *Frontiers in Neuroscience* (2020).

[Dü+17] S. Dünkel et al. "A FeFET based super-low-power ultra-fast embedded NVM technology for 22nm FDSOI and beyond." In: *International Electron Devices Meeting (IEDM)*. 2017.

[Fas] *FashionMNIST Dataset, https://github.com/zalandoresearch/fashion-mnist*. Accessed 2023-09-01. URL: https://github.com/zalandoresearch/fashion-mnist.

[GNA18] Mehdi Ghasemzadeh, Saeid Najafibisfar, and Abdollah Amini. "Ultra Low-power, High-speed Digital Comparator." In: *International Conference "Mixed Design of Integrated Circuits and System" (MIXDES)*. 2018.

[Gho+21] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. "A Survey of Quantization Methods for Efficient Neural Network Inference." In: *arXiv:2103.13630* (2021).

[Goe+06] S. Goel, M.A. Elgamel, M.A. Bayoumi, and Y. Hanafy. "Design methodologies for high-performance noise-tolerant XOR-XNOR circuits." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2006).

[GDM83] N.F. Goncalves and H. De Man. "NORA: a racefree dynamic CMOS technique for pipelined logic structures." In: *IEEE Journal of Solid-State Circuits* (1983).

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. 2016.

[Gou+21] Jianping Gou, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao. "Knowledge Distillation: A Survey." In: *International Journal of Computer Vision* (2021).

[HWC17] C. Ha, Y. Wang, and C. Chang. "Dynamic Power Management for wearable devices with Non-Volatile Memory." In: *International Conference on Applied System Innovation (ICASI)*. 2017.

[Ha+21] Minho Ha, Younghoon Byun, Seungsik Moon, Youngjoo Lee, and Sunggu Lee. "Layerwise Buffer Voltage Scaling for Energy-Efficient Convolutional Neural Network." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[Hak+19] Christian Hakert et al. "Stack Usage Analysis for Efficient Wear Leveling in Non-Volatile Main Memory Systems." In: *Workshop on Machine Learning for CAD (MLCAD)*. 2019.

[HH15] Sarah Harris and David Harris. *Digital Design and Computer Architecture: ARM Edition*. 2015.

[HTY17] Raqibul Hasan, Tarek M. Taha, and Chris Yakopcic. "On-chip training of memristor based deep neural networks." In: *International Joint Conference on Neural Networks (IJCNN)*. 2017.

[He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning for Image Recognition." In: *Computer Vision and Pattern Recognition Conference (CVPR)*. 2016.

[He+20] Xiangyu He, Zitao Mo, Ke Cheng, Weixiang Xu, Qinghao Hu, Peisong Wang, Qingshan Liu, and Jian Cheng. "ProxyBNN: Learning Binarized Neural Networks via Proxy Matrices." In: *European Conference on Computer Vision (ECCV)*. 2020.

[Hel+19] Koen Helwegen, James Widdicombe, Lukas Geiger, Zechun Liu, Kwang-Ting Cheng, and Roeland Nusselder. "Latent Weights Do Not Exist: Rethinking Binarized Neural Network Optimization." In: *Neural Information Processing Systems (NIPS)*. 2019.

[HLPS20] Sébastien Henwood, François Leduc-Primeau, and Yvon Savaria. "Layerwise Noise Maximisation to Train Low-Energy Deep Neural Networks." In: *International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 2020.

[Hir+19a] Tifenn Hirtzlin, Bogdan Penkovsky, Jacques-Olivier Klein, Nicolas Locatelli, Adrien F. Vincent, Marc Bocquet, Jean Michel Portal, and Damien Querlioz. "Implementing Binarized Neural Networks with Magnetoresistive RAM without Error Correction." In: *arXiv:1908.04085* (2019).

[Hir+19b] Tifenn Hirtzlin, Marc Bocquet, Jacques-Olivier Klein, Etienne Nowak, Elisa Vianello, Jean Michel Portal, and Damien Querlioz. "Outstanding Bit Error Tolerance of Resistive RAM-Based Binarized Networks." In: *International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. 2019.

[Hir+19c] Tifenn Hirtzlin, Bogdan Penkovsky, Marc Bocquet, Jacques-Olivier Klein, Jean-Michel Portal, and Damien Querlioz. "Stochastic Computing for Hardware Implementation of Binarized Neural Networks." In: *IEEE Access* (2019).

[How+17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." In: *arXiv:1704.04861* (2017).

[Hu+18] Yuwei Hu, Jidong Zhai, Dinghua Li, Yifan Gong, Yuhao Zhu, Wei Liu, Lei Su, and Jiangming Jin. "BitFlow: Exploiting Vector Parallelism for Binary Neural Networks on CPU." In: *International Parallel and Distributed Processing Symposium (IPDPS)*. 2018.

[Hua21] Chun-Hsian Huang. "An FPGA-Based Hardware/Software Design Using Binarized Neural Networks for Agricultural Applications: A Case Study." In: *IEEE Access* (2021).

[Hub+16] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized neural networks." In: *Neural Information Processing Systems (NIPS)*. 2016.

[Ima] *Imagenette Dataset, https://github.com/fastai/imagenette*. Accessed 2023-09-01. URL: https://github.com/fastai/imagenette.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *arXiv:1502.03167* (2015).

[Jac+] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference." In: *Computer Vision and Pattern Recognition (CVPR)*.

[Jaf+18] Ali Jafari, Morteza Hosseini, Adwaya Kulkarni, Chintan Patel, and Tinoosh Mohsenin. "BiNMAC: Binarized Neural Network Manycore ACcelerator." In: *GLSVLSI*. 2018.

[Jeb+21] F. Jebali, A. Majumdar, A. Laborieux, T. Hirtzlin, E. Vianello, J.P. Walder, M. Bocquet, D. Querlioz, and J. M. Portal. "CAPC: A Configurable Analog Pop-Count Circuit for Near-Memory Binary Neural Networks." In: *International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2021.

[Kal+19] Dhiraj Kalamkar et al. "A Study of BFLOAT16 for Deep Learning Training." In: *arXiv:1905.12322* (2019).

[KBB20] Navid Khoshavi, Connor Broyles, and Yu Bi. "A Survey on Impact of Transient Faults on BNN Inference Accelerators." In: *arXiv:2004.05915* (2020).

[KSC20] Dahyun Kim, Kunal Pratap Singh, and Jonghyun Choi. "Learning Architectures for Binary Networks." In: *European Conference on Computer Vision (ECCV)*. 2020.

[KLC18] Jaehyun Kim, Chaeun Lee, and Kiyoung Choi. "Energy Efficient Analog Synapse/Neuron Circuit for Binarized Neural Networks." In: *International SoC Design Conference (ISOCC)*. 2018.

[Kim+19] Jaehyun Kim, Chaeun Lee, Jihun Kim, Yumin Kim, Cheol Seong Hwang, and Kiyoung Choi. "VCAM: Variation Compensation through Activation Matching for Analog Binarized Neural Networks." In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2019.

[KS16] Minje Kim and Paris Smaragdis. "Bitwise Neural Networks." In: *arXiv: 1601.06071* (2016).

[KB14] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In: *arXiv:1412.6980* (2014).

[Kop+19] Skanda Koppula, Lois Orosa, A. Giray Yağlikçi, Roknoddin Azizi, Taha Shahroodi, Konstantinos Kanellopoulos, and Onur Mutlu. "EDEN: Enabling Energy-Efficient, High-Performance Deep Neural Network Inference Using Approximate DRAM." In: *International Symposium on Microarchitecture (MICRO)*. 2019.

[KB20] Anastasis Kratsios and Ievgen Bilokopytov. "Non-Euclidean Universal Approximation." In: *Neural Information Processing Systems (NIPS)*. 2020.

[Kuk+19] Navjot Kukreja, Alena Shilova, Olivier Beaumont, Jan Huckelheim, Nicola Ferrier, Paul Hovland, and Gerard Gorman. "Training on the Edge: The why and the how." In: *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019.

[Kul+22] Uday Kulkarni, Abhishek S Hosamani, Abhishek S Masur, Shashank Hegde, Ganesh R Vernekar, and K Siri Chandana. "A Survey on Quantization Methods for Optimization of Deep Neural Networks." In: *International Conference on Automation, Computing and Renewable Systems (ICACRS)*. 2022.

[Kun82] Kung. "Why systolic architectures?" In: *Computer* (1982).

[Kwo+20] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. "MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings." In: *IEEE Micro* (2020).

[Lab+21] Axel Laborieux, Maxence Ernoult, Tifenn Hirtzlin, and Damien Querlioz. "Synaptic metaplasticity in binarized neural networks." In: *Nature Communications* (2021).

[LS21] Ang Li and Simon Su. "Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs." In: *IEEE Transactions on Parallel and Distributed Systems* (2021).

[Lia+18] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. "FP-BNN: Binarized neural network on FPGA." In: *Neurocomputing* (2018).

[Lia+21] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. "Pruning and Quantization for Deep Neural Network Acceleration: A Survey." In: *arXiv:2101.09671* (2021).

[Lin+20] Mingbao Lin, Rongrong Ji, Zihan Xu, Baochang Zhang, Yan Wang, Yongjian Wu, Feiyue Huang, and Chia-Wen Lin. "Rotated Binary Neural Network." In: *arXiv:2009.13055* (2020).

[Liu+13] Q. Liu, M. Vinet, J. Gimbert, N. Loubet, R. Wacquez, L. Grenouillet, Y. Le Tiec, et al. "High performance UTBB FDSOI devices featuring 20nm gate length for 14nm node and beyond." In: *International Electron Devices Meeting (IEDM)*. 2013.

[Lon+18] Y. Long, T. Na, P. Rastogi, K. Rao, A. I. Khan, S. Yalamanchili, and S. Mukhopadhyay. "A Ferroelectric FET based Power-efficient Architecture for Data-intensive Computing." In: *International Conference on Computer-Aided Design (ICCAD)*. 2018.

[Luo+19] Cheng Luo, Man-Kit Sit, Hongxiang Fan, Shuanglong Liu, Wayne Luk, and Ce Guo. "Towards Efficient Deep Neural Network Training by FPGA-Based Batch-Level Parallelism." In: *Field-Programmable Custom Computing Machines (FCCM)*. 2019.

[Luo+23] Fei Luo, Salabat Khan, Yandao Huang, and Kaishun Wu. "Binarized Neural Network for Edge Intelligence of Sensor-Based Human Activity Recognition." In: *IEEE Transactions on Mobile Computing* (2023).

[Mac+21] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. "FPnew: An Open-Source Multiformat Floating-Point Unit Architecture for Energy-Proportional Transprecision Computing." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2021).

[Mar+20] Brais Martinez, Jing Yang, Adrian Bulat, and Georgios Tzimiropoulos. "Training Binary Neural Networks with Real-to-Binary Convolutions." In: *arXiv:2003.11535* (2020).

[MGD20] Rahul Mishra, Hari Prabhat Gupta, and Tanima Dutta. "A Survey on Deep Neural Network Compression: Challenges, Overview, and Solutions." In: *arXiv:2010.03954* (2020).

[Mit16] Sparsh Mittal. "A Survey of Techniques for Approximate Computing." In: *ACM Comput. Surv.* (2016).

[Moh+23] Vahidreza Mohaghaddas, Hammam Kattan, Tim Buecher, Mikail Yayla, Jian-Jia Chen, and Hussam Amrouch. "Temperature-Aware Memory Mapping and Active Cooling of Neural Processing Units." In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2023.

[Mra+19] V. Mrazek, Z. Vasicek, L. Sekanina, M. A. Hanif, and M. Shafique. "ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining." In: *International Conference on Computer-Aided Design (ICCAD)*. 2019.

[Mur] Boris Murmann. *ADC Performance Survey 1997-2021 [Online]*. Accessed 2023-09-01. URL: Available : http : / / web . stanford . edu / ~murmann / adcsurvey.html..

[MN+21] Mirembe Musisi-Nkambwe, Sahra Afshari, Hugh J. Barnaby, Michael N. Kozicki, and Ivan Sanchez Esqueda. "The viability of analog-based accelerators for neuromorphic computing: a survey." In: *Neuromorph. Comput. Eng.* (2021).

[Nat+14] S Natarajan, M Agostinelli, S Akbar, M Bost, A Bowonder, V Chikarmane, S Chouksey, A Dasgupta, K Fischer, Q Fu, et al. "A 14nm logic technology featuring 2 nd-generation finfet, air-gapped interconnects, self-aligned double patterning and a 0.0588 μm 2 sram cell size." In: *International Electron Devices Meeting (IEDM)*. 2014.

[Ni+18] Kai Ni, Pankaj Sharma, Jianchi Zhang, Matthew Jerry, Jeffery A Smith, Kandabara Tapily, Robert Clark, Souvik Mahapatra, and Suman Datta. "Critical role of interlayer in Hf 0.5 Zr 0.5 O 2 ferroelectric FET nonvolatile memory performance." In: *IEEE Transactions on Electron Devices* (2018).

[Ni+19] Kai Ni et al. "Ferroelectric ternary content-addressable memory for one-shot learning." In: *Nature Electronics* (2019).

[Nie15] Michael Nielsen. *Neural Networks and Deep Learning*. 2015.

[Nur+16] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. "Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC." In: *International Conference on Field-Programmable Technology (FPT)*. 2016.

[Oh+20] Dong-Ryeol Oh, Kyoung-Jun Moon, Won-Mook Lim, Ye-Dam Kim, Eun-Ji An, and Seung-Tak Ryu. "An 8b 1GS/s 2.55mW SAR-Flash ADC with Complementary Dynamic Amplifiers." In: *2020 IEEE Symposium on VLSI Circuits*. 2020.

[OS89] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. 1989.

[Pan+18] Y. Pan, P. Ouyang, Y. Zhao, W. Kang, S. Yin, Y. Zhang, W. Zhao, and S. Wei. "A Multilevel Cell STT-MRAM-Based Computing In-Memory Accelerator for Binary Convolutional Neural Network." In: *IEEE Transactions on Magnetics* (2018).

[Pas+19] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." In: *Neural Information Processing Systems (NIPS)*. 2019.

[PE23] Andreea Postovan and Mădălina Eraşcu. "Architecturing Binarized Neural Networks for Traffic Sign Recognition." In: *arXiv:2303.15005* (2023).

[RCN03] Jan M. Rabaey, Anantha P. Chandrakasan, and Borivoje Nikolic. *Digital Integrated Circuits: A Design Perspective*. 2003.

[RST19] Athanasios T. Ramkaj, Michiel S. J. Steyaert, and Filip Tavernier. "A 13.5-Gb/s 5-mV-Sensitivity 26.8-ps-CLK–OUT Delay Triple-Latch Feedforward Dynamic Comparator in 28-nm CMOS." In: *IEEE Solid-State Circuits Letters* (2019).

[Ras+16] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks." In: *arXiv:1603.05279* (2016).

[Red+15] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection." In: *arXiv:1506.02640* (2015).

[Rei+19] Dayane Reis, Kai Ni, Wriddhi Chakraborty, Xunzhao Yin, Martin Trentzsch, Stefan Dünkel Dünkel, Thomas Melde, Johannes Müller, Sven Beyer, Suman Datta, et al. "Design and Analysis of an Ultra-Dense, Low-Leakage, and Fast FeFET-Based Random Access Memory Array." In: *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits* (2019).

[Ros+03] L. Rosasco, E. De, Vito A. Caponnetto, M. Piana, and A. Verri. "Are loss functions all the same." In: *Neural Computation* (2003).

[RCB18] Manuele Rusci, Lukas Cavigelli, and Luca Benini. "Design Automation for Binarized Neural Networks: A Quantum Leap Opportunity?" In: *International Symposium on Circuits and Systems (ISCAS)*. 2018.

[Svh] *SVHN Dataset, http://ufldl.stanford.edu/housenumbers/*. Accessed 2023-09-01. URL: http://ufldl.stanford.edu/housenumbers/.

[Sab+23] Muhammad Sabih, Mikail Yayla, Frank Hannig, Jürgen Teich, and Jian-Jia Chen. "Robust and Tiny Binary Neural Networks using Gradient-based Explainability Methods." In: *Workshop on Machine Learning and Systems (EuroMLSys)*. 2023.

[SBN19] Eyyüb Sari, Mouloud Belbahri, and Vahid Partovi Nia. "How Does Batch Normalization Help Binary Training?" In: *arXiv:1909.09139* (2019).

[Say+23] Ratshih Sayed, Haytham Azmi, Heba Shawkey, A. H. Khalil, and Mohamed Refky. "A Systematic Literature Review on Binary Neural Networks." In: *IEEE Access* (2023).

[Sch+17] Catherine D. Schuman, Thomas E. Potok, Robert M. Patton, J. Douglas Birdwell, Mark E. Dean, Garrett S. Rose, and James S. Plank. "A Survey of Neuromorphic Computing and Neural Networks in Hardware." In: *arXiv:1705.06963* (2017).

[Sha+16a] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars." In: *International Symposium on Computer Architecture (ISCA)*. 2016.

[Sha+16b] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars." In: *ACM SIGARCH Computer Architecture News* (2016).

[Shr+20] Kumar Shridhar, Harshil Jain, Akshat Agarwal, and Denis Kleyko. "End to End Binarized Neural Networks for Text Classification." In: *Proceedings of SustaiNLP: Workshop on Simple and Efficient Natural Language Processing*. 2020.

[SL19] Taylor Simons and Dah-Jye Lee. "A Review of Binarized Neural Networks." In: *MDPI Electronics* (2019).

[SZ14] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." In: *International Conference on Learning Representations (ICLR)*. 2014.

[Soh+19] Nimit Sharad Sohoni, Christopher Richard Aberger, Megan Leszczynski, Jian Zhang, and Christopher Ré. "Low-Memory Neural Network Training: A Technical Report." In: *arXiv:1904.10631* (2019).

[Sol+20a] T. Soliman et al. "Ultra-Low Power Flexible Precision FeFET Based Analog In-Memory Computing." In: *IEEE International Electron Devices Meeting (IEDM)*. 2020.

[Sol+20b] Taha Soliman, Ricardo Olivo, Tobias Kirchner, Cecilia De la Parra, Maximilian Lederer, Thomas Kämpfe, Andre Guntoro, and Norbert Wehn. "Efficient FeFET Crossbar Accelerator for Binary Neural Networks." In: *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2020.

[Spa+19] Matteo Spallanzani, Lukas Cavigelli, Gian Paolo Leonardi, Marko Bertogna, and Luca Benini. " Additive Noise Annealing and Approximation Properties of Quantized Neural Networks." In: *arXiv:1905.10452* (2019).

[SGM20] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and Policy Considerations for Modern Deep Learning Research." In: *AAAI* (2020).

[Stu+23] David Stutz, Nandhini Chandramoorthy, Matthias Hein, and Bernt Schiele. "Random and Adversarial Bit Error Robustness: Energy-Efficient and Secure DNN Accelerators." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2023).

[Sun+17] X. Sun, R. Liu, Y. Chen, H. Chiu, W. Chen, M. Chang, and S. Yu. "Low-VDD Operation of SRAM Synaptic Array for Implementing Ternary Neural Network." In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2017).

[Sun+18] X. Sun, X. Peng, P. Chen, R. Liu, J. Seo, and S. Yu. "Binary neural network with 16 Mb RRAM macro chip for classification and online training." In: *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2018.

[Sun+20] Xiao Sun, Naigang Wang, Chia-yu Chen, Jia-min Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. "Ultra-Low Precision 4-Bit Training of Deep Neural Networks." In: *Neural Information Processing Systems (NIPS)*. 2020.

[Sun+18] Xiaoyu Sun, Shihui Yin, Xiaochen Peng, Rui Liu, Jae-sun Seo, and Shimeng Yu. "XNOR-RRAM: A scalable and parallel resistive synaptic architecture for binary neural networks." In: *Design, Automation Test in Europe Conference Exhibition (DATE)*. 2018.

[Sze+17] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." In: *Proceedings of the IEEE* (2017).

[TM00] Kazukiyo Takahashi and Mitsuru Mizunuma. "Adiabatic dynamic CMOS logic circuit." In: *Electronics and Communications in Japan (Part II: Electronics)* (2000).

[TG17] C. Torres-Huitzil and B. Girau. "Fault and Error Tolerance in Neural Networks: A Review." In: *IEEE Access* (2017).

[Tre+16] M. Trentzsch et al. "A 28nm HKMG super low power embedded NVM technology based on ferroelectric FETs." In: *International Electron Devices Meeting (IEDM)*. 2016.

[Tu+22] Zhijun Tu, Xinghao Chen, Pengju Ren, and Yunhe Wang. "AdaBin: Improving Binary Neural Networks with Adaptive Binary Sets." In: *European Conference on Computer Vision (ECCV)*. 2022.

[TGW19] Michail Tzoufras, Marcin Gajek, and Andrew Walker. "Magnetoresistive RAM for error resilient XNOR-Nets." In: *arXiv:1905.10927* (2019).

[Umu+17] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. "FINN." In: *International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2017.

[Vin+15] A. F. Vincent, J. Larroque, N. Locatelli, N. Ben Romdhane, O. Bichler, C. Gamrat, W. S. Zhao, J. Klein, S. Galdin-Retailleau, and D. Querlioz. "Spin-Transfer Torque Magnetic Memory as a Stochastic Memristive Synapse for Neuromorphic Systems." In: *Transactions on Biomedical Circuits and Systems* (2015).

[Wan+21] Erwei Wang, James J. Davis, Daniele Moro, Piotr Zielinski, Jia Jie Lim, Claudionor Coelho, Satrajit Chatterjee, Peter Y. K. Cheung, and George A. Constantinides. "Enabling Binary Neural Network Training on the Edge." In: *International Workshop on Embedded and Mobile Deep Learning (EDML)*. 2021.

[Wan+11] Jinhui Wang, Na Gong, Ligang Hou, Xiaohong Peng, Shuqin Geng, and Wuchen Wu. "Low power and high performance dynamic CMOS XOR/XNOR gate design." In: *Microelectronic Engineering* (2011).

[Wan+18] Yanzhi Wang, Zheng Zhan, Jiayu Li, Jian Tang, Bo Yuan, Liang Zhao, Wujie Wen, Siyue Wang, and Xue Lin. "Universal Approximation Property and Equivalence of Stochastic Computing-based Neural Networks and Binary Neural Networks." In: *arXiv:1803.05391* (2018).

[WHA18] Kaijie Wei, Koki Honda, and Hideharu Amano. "FPGA Design for Autonomous Vehicle Driving Using Binarized Neural Networks." In: *International Conference on Field-Programmable Technology (FPT)*. 2018.

[Wei+21a] Ming-Liang Wei, Mikail Yayla, Shu-Yin Ho, Jian-Jia Chen, Chia-Lin Yang, and Hussam Amrouch. "Binarized SNNs: Efficient and Error-Resilient Spiking Neural Networks through Binarization." In: *International Conference On Computer Aided Design (ICCAD)*. 2021.

[Wei+21b] Ming-Liang Wei, Hussam Amrouch, Cheng-Lin Sung, Hang-Ting Lue, Chia-Lin Yang, Keh-Chung Wang, and Chih-Yuan Lu. "Robust Brain-Inspired Computing: On the Reliability of Spiking Neural Network Using Emerging Non-Volatile Synapses." In: *International Reliability Physics Symposium (IRPS)*. 2021.

[Wei+23] Ming-Liang Wei, Mikail Yayla, Shu-Yin Ho, Jian-Jia Chen, Hussam Amrouch, and Chia-Lin Yang. "Impact of Non-volatile Memory Cells on Spiking Neural Network Annealing Machine with In-situ Synapse Processing." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2023).

[Wen21] Olivia Weng. "Neural Network Quantization for Efficient Inference: A Survey." In: *arXiv:2112.06126* (2021).

[WH10] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 2010.

[Wu+20] Qing Wu, Xiaojin Lu, Shan Xue, Chao Wang, Xundong Wu, and Jin Fan. "SBNN: Slimming binarized neural network." In: *Neurocomputing* (2020).

[Xia+20] Yachen Xiang, Peng Huang, Runze Han, Chu Li, Kunliang Wang, Xiaoyan Liu, and Jinfeng Kang. "Efficient and Robust Spike-Driven Deep Convolutional Neural Networks Based on NOR Flash Computing Array." In: *IEEE Transactions on Electron Devices* (2020).

[Xu+21] Zihan Xu, Mingbao Lin, Jianzhuang Liu, Jie Chen, Ling Shao, Yue Gao, Yonghong Tian, and Rongrong Ji. "ReCU: Reviving the Dead Weights in Binary Neural Networks." In: *International Conference on Computer Vision (ICCV)*. 2021.

[Yan+18] Lita Yang, Daniel Bankman, Bert Moons, Marian Verhelst, and Boris Murmann. "Bit Error Tolerance of a CIFAR-10 Binarized Convolutional Neural Network Processor." In: *International Symposium on Circuits and Systems (ISCAS)*. 2018.

[Yan+17] Tien-Ju Yang, Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "A method to estimate the energy consumption of deep neural networks." In: *Asilomar Conference on Signals, Systems, and Computers*. 2017.

[YC22] Mikail Yayla and Jian-Jia Chen. "Memory-Efficient Training of Binarized Neural Networks on the Edge." In: *Design Automation Conference (DAC)*. 2022.

[Yay+21] Mikail Yayla, Mario Günzel, Burim Ramosaj, and Jian-Jia Chen. "Universal Approximation Theorems of Fully Connected Binarized Neural Networks." In: *arXiv:2102.02631* (2021).

[Yay+22a] Mikail Yayla, Sebastian Buschjäger, Aniket Gupta, Jian-Jia Chen, Jörg Henkel, Katharina Morik, Kuan-Hsun Chen, and Hussam Amrouch. "FeFET-Based Binarized Neural Networks Under Temperature-Dependent Bit Errors." In: *IEEE Transactions on Computers* (2022).

[Yay+22b] Mikail Yayla, Simon Thomann, Sebastian Buschjäger, Katharina Morik, Jian-Jia Chen, and Hussam Amrouch. "Reliable Binarized Neural Networks on Unreliable Beyond Von-Neumann Architecture." In: *IEEE Transactions on Circuits and Systems I: Regular Papers* (2022).

[Yay+22c] Mikail Yayla, Zahra Valipour Dehnoo, Mojtaba Masoudinejad, and Jian-Jia Chen. "TREAM: A Tool for Evaluating Error Resilience of Tree-Based Models Using Approximate Memory." In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2022.

[Yay+23a] Mikail Yayla, Cecilia Latotzke, Robert Huber, Somar Iskif, Tobias Gemmeke, and Jian-Jia Chen. "DAEBI: A Tool for Data Flow and Architecture Explorations of Binary Neural Network Accelerators." In: *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 2023.

[Yay+23b] Mikail Yayla, Simon Thomann, Ming-Liang Wei, Chia-Lin Yang, Jian-Jia Chen, and Hussam Amrouch. "HW/SW Codesign for Robust and Efficient Binarized SNNs by Capacitor Minimization." In: *arXiv:2309.02111* (2023).

[Yay+23c] Mikail Yayla, Fabio Frustaci, Fanny Spagnolo, Jian-Jia Chen, and Hussam Amrouch. "Unlocking Efficiency in BNNs: Global by Local Thresholding for Analog-based HW Accelerators." In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* (2023).

[Yin+20] Shihui Yin, Zhewei Jiang, Jae-Sun Seo, and Mingoo Seok. "XNOR-SRAM: In-Memory Computing SRAM Macro for Binary/Ternary Deep Neural Networks." In: *IEEE Journal of Solid-State Circuits* (2020).

[Yoo+19] Insik Yoon, Matthew Jerry, Suman Datta, and Arijit Raychowdhury. "Design space exploration of Ferroelectric FET based Processing-in-Memory DNN Accelerator." In: *arXiv:1908.07942* (2019).

[Yu+16] S. Yu, Z. Li, P. Chen, H. Wu, B. Gao, D. Wang, W. Wu, and H. Qian. "Binary neural network with 16 Mb RRAM macro chip for classification and online training." In: *International Electron Devices Meeting (IEDM)*. 2016.

[Yu+21] Shimeng Yu, Hongwu Jiang, Shanshi Huang, Xiaochen Peng, and Anni Lu. "Compute-in-Memory Chips for Deep Learning: Recent Trends and Prospects." In: *IEEE Circuits and Systems Magazine* (2021).

[YA23] C. Yuan and S.S. Agaian. "A comprehensive review of Binary Neural Network." In: *Artif Intell Rev* (2023).

[Zha+19a] Jianhao Zhang, Yingwei Pan, Ting Yao, He Zhao, and Tao Mei. "daBNN: A Super Fast Inference Framework for Binary Neural Networks on ARM devices." In: *arXiv:1908.05858* (2019).

[Zha04] Tong Zhang. "Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms." In: *International Conference on Machine Learning (ICML)*. 2004.

[ZCH19] X. Zhang, X. Chen, and Y. Han. "FeMAT: Exploring In-Memory Processing in Multifunctional FeFET-Based Memory Array." In: *International Conference on Computer Design (ICCD)*. 2019.

[Zha+20] Yizhou Zhang et al. "An Improved RRAM-Based Binarized Neural Network With High Variation-Tolerated Forward/Backward Propagation Module." In: *IEEE Transactions on Electron Devices* (2020).

[Zha+23a] Yu Zhang, Gang Chen, Tao He, Qian Huang, and Kai Huang. "ViraEye: An Energy-Efficient Stereo Vision Accelerator with Binary Neural Network in 55 nm CMOS." In: *Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2023.

[Zha+23b] Junwei Zhao, Shiliang Zhang, Zhaofei Yu, and Tiejun Huang. "SpiReco: Fast and Efficient Recognition of High-Speed Moving Objects with Spike Cameras." In: *IEEE Transactions on Circuits and Systems for Video Technology* (2023).

[Zha+17] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs." In: *FPGA*. 2017.

[Zha+19b] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Christopher De Sa, and Zhang Zhiru. "Improving Neural Network Quantization without Retraining using Outlier Channel Splitting." In: *International Conference on Machine Learning (ICML)*. 2019.

[Zha+18] Tianli Zhao, Xiangyu He, Jian Cheng, and Jing Hu. "BitStream: Efficient Computing Architecture for Real-Time Low-Power Inference of Binary Neural Networks on CPUs." In: *International Conference On Multimedia*. 2018.

[Zha+16] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. "F-CNN: An FPGA-based framework for training Convolutional Neural Networks." In: *International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 2016.

[Zha+21] Wenyu Zhao, Teli Ma, Xuan Gong, Baochang Zhang, and David Doermann. "A Review of Recent Advances of Binary Neural Networks for Edge Computing." In: *IEEE Journal on Miniaturization for Air and Space Systems* (2021).

[Syn] *https://www.synopsys.com/silicon/tcad.html*.

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

AD	Accuracy Drop. 66, 68, 69, 139
Adam	Adaptive Moment Estimation. 14, 20, 105, 111
ADC	Analog-to-Digital Converter. 8, 28, 29, 71–73, 77–80, 83–87, 89, 90, 119
AFO	Absolute Frequency of Occurrences. 94, 95
ALU	Arithmetic Logic Unit. 54
AP	Analog Path. 8, 27, 71–73, 77–79, 84, 137
ASIC	Application-Specific Integrated Circuit. iii, 5, 22, 24–27, 103, 137

BER Bit Error Rate. 3, 41, 42, 44
BERA Bit Error Rate Assignment. 54–56, 58, 59, 117, 138
BET Bit Error Tolerance. 53–56
BN Batch Normalization. 17–19, 32, 53, 54, 141
BNN Binarized Neural Network. iii, iv, 1–9, 11, 15–29, 32, 33, 35, 36, 38, 40–42, 44, 45, 47, 50–62, 64–75, 81, 86–90, 98–100, 103–106, 108, 110, 117–119, 137, 139, 141
Bop Binary Optimizer. iv, 20, 103–106, 108–111, 115, 118, 142
BOX buried oxide. 99

C Convolution. 32, 33, 53, 54, 141
CEL Cross Entropy Loss. 33, 35, 36, 41, 42, 44, 45, 55
CMOS Complementary Metal–Oxide–Semiconductor. 24, 28, 29, 47, 48, 59, 60, 64
CPU Central Processing Unit. 2, 22, 24, 25, 34, 52, 56, 137
CTF Charge Trap Flash. 23, 47

DP Digital Path. 8, 27, 71–73, 77–79, 84, 137
DRAM Dynamic Random Access Memory. 23, 24, 52, 138

ECC Error Correction Code. 54
EDA Electronic Design Automation. 26

FC Fully Connected. 31–33, 141
FD-SOI Fully Depleted Silicon-On-Insulator. 84, 87, 99
FeFET Ferroelectric FET. iv, 7, 8, 23, 24, 26, 27, 29, 47–57, 59–64, 68, 70, 83, 85, 117, 118, 137–139, 141
FeFinFET Ferroelectric FinFET. 60–62, 138
FeRAM Ferroelectric RAM. 47
FET Field Effect Transistor. 47, 48, 143
FF Flip Flop. 91–94, 99, 140
FIFO First In First Out. 76, 88, 139
FinFET Fin Field Effect Transistor. 48, 49, 143
FP Floating Point. iv, 7, 9, 103–115, 118, 119, 140, 142

FPGA	Field-Programmable Gate Array. iii, 2, 5, 16, 22, 24–26, 103, 114, 137
FPU	Floating Point Unit. 112–115, 140
GPU	Graphics Processing Unit. 2, 5, 22, 24, 25, 34, 52, 137
HDL	Hardware Description Language. 26
HLS	High-Level Synthesis. 26
HW	hardware. iv, 6, 8, 26, 28, 71, 90, 91, 101, 118
IF	Integrate-and-Fire. iv, 90
IF-SNN	Integrate-and-Fire Spiking Neural Network. iv, 8, 71, 90–95, 97, 98, 101, 118, 140
iFIFO	Input FIFO. 76, 77, 139
LBW	Less Buffer Writes. 50, 53–57, 141
LiM	Logic in Memory. iv, 8, 47, 59, 60, 62, 63, 67, 68, 70, 117, 138, 139
LTA	Local Thresholding Approximation. iv, 8, 72–89, 117–119, 139, 141
LTA-MU	LTA Maximum Utilization. 80, 83, 84
LUT	Lookup Table. 114
MAC	Multiply–accumulate. 1, 2, 4, 11, 16, 17, 22, 28, 33, 34, 65, 67, 81, 90–92, 94, 95, 98–101, 118, 139, 140
MC	Monte Carlo. 50, 51, 62
MHL	Modified Hinge Loss. iv, 7, 35, 36, 40–42, 44, 45, 65, 66, 81, 98, 117
ML	Match Line. 61, 99
MP	Maxpool. 32, 33, 53, 54, 141
MRAM	Magnetoresistive RAM. 23, 24, 47
NAS	Neural Architecture Search. 4, 20
NN	Neural Network. iii, iv, 1–7, 11–16, 18–24, 26–28, 35, 41, 47, 49, 50, 59, 66, 90, 103, 104, 119, 137
NVM	Non-Volatile Memory. 23, 47, 48, 50–52, 59, 78, 138

OS	Output Stationary. 26
PE	Processing Element. 52
PV	Process Variation. 49, 51, 60
QNN	Quantized Neural Network. 3, 137
ReLU	Rectified Linear Unit. 20, 21
RRAM	Resistive Random Access Memory. 23, 29, 47, 69
SGD	Stochastic Gradient Descent. 14, 40, 41
SNN	Spiking Neural Network. iv, 90, 91
SOTA	State Of The Art. 73, 78–81, 83–88, 139, 141
SPICE	Simulation with Integrated Circuit Emphasis. 27, 62, 98, 99
SRAM	Static Random Access Memory. 23, 24, 29, 48, 59, 60, 99
STE	straight-through-estimator. 18
STT-RAM	Spin-Transfer Torque RAM. 23, 47
SVM	Support Vector Machine. 7, 35, 36, 45, 117
SW	software. iv, 6, 8, 71, 90, 101, 118
TCAD	Technology CAD. 49, 50
WS	Weight Stationary. 26
WSAD	Weighted Speed Accuracy Drop. 67, 69

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Final Version as of March 22, 2024 (classicthesis v4.6).