Copyright
by
Zachary Susskind
2024

The Dissertation Committee for Zachary Susskind
certifies that this is the approved version of the following dissertation:

Weightless Neural Networks for Fast, Low-Energy Inference

Committee:

Lizy Kurian John, Supervisor

Derek Chiou

Mattan Erez

Felipe Maia Galvão França

Diana Marculescu

Mike O'Connor

Weightless Neural Networks for Fast, Low-Energy Inference

by

Zachary Susskind

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2024

Acknowledgments

I have an immense amount of gratitude to express and a finite number of words in which to do so. Writing these acknowledgments is therefore, as the cliché goes, *highly nontrivial*.

To begin, I would like to express my thanks to my advisor, Dr. Lizy John, for her unwavering support throughout this entire journey. Under her guidance, I have developed from an enthusiastic but naive undergraduate into a confident researcher. I am grateful for the freedom she gave me to explore new ideas, even when they didn't pan out, and for her invaluable advice that kept me on the right track.

Special thanks go to Dr. Mauricio Breternitz Jr., who first introduced me to weightless neural networks. Were it not for this bit of serendipity, I likely would have never discovered my interest in this topic and this dissertation would not exist.

I would also like to thank my committee members, Drs. Derek Chiou, Mattan Erez, Diana Marculescu, Felipe França (IT Porto), and Mike O'Connor (NVIDIA), for their feedback, suggestions, and scheduling flexibility. I have taken classes from or worked closely with many of them, and have grown academically and personally from their teaching and mentorship.

I'm grateful to all of my collaborators, both students and professors. I would like to particularly acknowledge Alan Bacellar and now-Dr. Aman Arora. Aman was hugely helpful with assisting me in preparing my first few papers when I still wasn't quite sure what I was doing, and was eternally willing to wrestle with buggy Xilinx tools. Alan consistently came at problems with a fresh perspective, which enabled him to sometimes see opportunities that I had overlooked.

Of course, I also have to thank all members of the WNN research working group for their direct and indirect contributions towards this dissertation. It's been fantastic working with you for these last several years.

I also want to shout out the members of the Laboratory for Computer Architecture, including Mugdha Jadhao and Shashank Nag for their continuation of this work, Bagus Hanindhito and Ruihao Li for their DGX server wrangling and weekly taco deliveries, and Allison Seigler, Siyuan Ma, and Zhigang Wei for their support.

I'd be remiss if I didn't mention our fantastic ECE department and Cockrell School staff, including Tom Atchity, Barbara Heine, Barry Levitch, David Korts, Lisa Contes, Leticia Lira, Melanie Gulick, and Melody Singleton. The work y'all do is critical and often overlooked.

I'm also grateful to the Semiconductor Research Corporation for funding my research, as well as the Cockrell Foundation and the Graduate School for providing supporting fellowships.

Lastly, I want to thank my family, and especially my mother, for supporting and believing in me over these past five years (and indeed these past twenty-six). It's been a long journey—now on to what's next!

Abstract

Weightless Neural Networks for Fast, Low-Energy Inference

Zachary Susskind, PhD
The University of Texas at Austin, 2024

SUPERVISOR: Lizy Kurian John

Despite significant advancements in efficient machine learning, deploying models such as deep neural networks (DNNs) on resource-constrained edge devices remains a major challenge. Conventional approaches transform pre-trained models using methods such as pruning and quantization to make better use of limited memory and compute resources. However, these approaches are insufficient when scaling to ultra-low-power "extreme edge" devices, particularly when high throughput and low latency are also desired. This domain demands approaches to machine learning which are designed from first principles to be more efficient in hardware. While some leading approaches, such as binary neural networks (BNNs), are structurally similar to DNNs, others are much more divergent in form. Weightless neural networks (WNNs), a class of machine learning model which perform computation using lookup tables, are interesting candidates in this space due to their inherent nonlinearity, efficiency of operation, and simplicity of construction.

In this dissertation, I explore the potential of WNNs to enable fast, efficient inference on the extreme edge. I first discuss BTHOWeN, which combines insights from recent WNN literature with additional algorithmic improvements to create a state-of-the-art weightless model with an accompanying FPGA-based accelerator architecture.

I next propose ULEEN, which introduces strategies to further improve the accuracy of WNNs as well as their efficiency in hardware, including a novel multi-pass learning rule and a lookup table pruning strategy. Lastly, I introduce the DWN model architecture, which enables models composed of multiple layers of small, directly-connected lookup tables to be trained using a gradient-based flow. In aggregate, these contributions yield WNNs which are fast, efficient, and readily implemented on low-end microcontrollers or as custom hardware accelerators, achieving for instance $>2000\times$ reduction in energy-delay product versus fully-connected BNNs in an FPGA. Overall, this work positions WNNs as a leading approach for tiny devices.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Artificial intelligence (AI) and mobile computing are advancing at blistering paces. Thanks in large part to the popularization and maturation of deep neural networks (DNNs) in the last fifteen years, AI has developed from a field once infamously summarized as "increasingly disappointing" [96] to a $240B market [136]. Concurrently, there are forecast to be 40 billion Internet-of-Things (IoT) devices worldwide by 2029, up from 15.7 billion today [141]. It is unsurprising, then, that the investigation of ways to use AI on these devices is an area of very active research.

Mobile and IoT devices have some unique constraints that make them challenging targets for AI models. Very often, they are battery-operated, which means that the energy used to run models must not be excessive. Furthermore, in order to keep them portable and inexpensive, they have smaller and slower processors and less memory than workstation or data-center-scale computers. By contrast, traditional DNNs require slow and energy-hungry floating-point arithmetic operations and have large numbers of learned "weight" parameters. The conventional way to resolve this incompatibility is to transmit data from a mobile device to a remote "cloud" server, which performs the actual task of running the DNN model. However, this approach has several limitations [38]: it introduces latency (response time) due to the need to transmit requests and responses over the network, it introduces security and privacy concerns if the central server can not be fully trusted or is compromised, it risks overloading this server if too many devices make requests, and it presupposes that a reliable network connection is available in the first place. This has prompted the emergence of the field of **edge computing**, which seeks to move computation to base stations and end devices through a combination of model and hardware optimizations.

Edge devices frequently include one or more **hardware accelerators**: specialized processors which are far faster and more efficient than CPUs but not suitable for general applications [165]. AI models intended for edge devices often incorporate

modifications which reduce their memory requirements and make them easier to run, usually at the cost of a small amount of accuracy. Restricting models to prediction (**inference**) rather than training on edge devices enables additional optimizations, since it fixes the values of weights to constants. For instance, values can be approximated using data formats that require less memory and are easier for computers to perform arithmetic with (**quantization** [62]), or the least-important parameters can be excised from the model entirely (**pruning** [40]). If the nature of these optimizations is known in advance, a custom hardware accelerator can be designed to take advantage of them. When we also choose specific modifications for a model based on their impacts on the efficiency of accelerators, this creates a feedback-driven process known as **algorithm-hardware co-design**.

1.1 Problem Description and Motivation

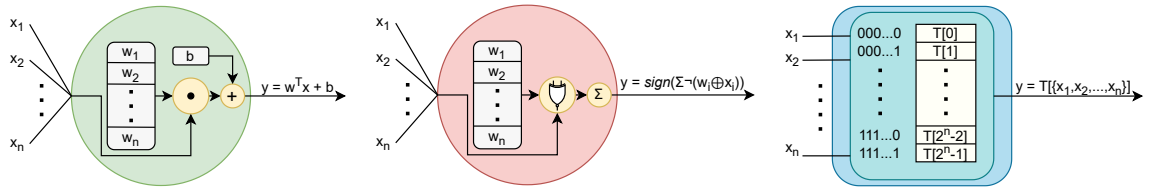
Standardized approaches to optimizing DNNs for edge devices can yield large improvements in efficiency. For example, quantizing a DNN from single-precision floating-point (FP32) to the commonly-used 8-bit integer (INT8) format reduces its memory footprint by a factor of 4 and the energy for arithmetic operations by a factor of 20 [70]. However, these techniques have limits to their scalability, and are insufficient to target extremely low-energy devices, particularly when high inference throughput, low latency, or a tiny circuit area are also needed. This has lead to a wide variety of approaches [13, 17, 18, 37, 51, 73, 100, 120, 145, 146, 152] which, rather than optimizing a DNN for deployment on edge devices, propose new types of AI models which are designed from first principles to be more efficient in hardware.

This avenue of research has become increasingly important with the emergence of "extreme edge" or "mist" devices [122], which put computation directly next to physical sensors and actuators. These devices generally include an ultra-low-power processor, a short-range radio transceiver, and possibly a small battery—or they may forego the battery and instead rely on harvesting energy from their surroundings or

from the physical phenomena they monitor. Examples include undersea submarine detectors powered by microbial activity [104], energy-harvesting image sensors [93], low-latency wearable biomedical monitoring devices [44], and the DARPA N-ZERO program [117], which developed sensors for passive monitoring. Other applications for extreme edge devices include "smart dust" projects, such as battery-free sensors designed to be dispersed by the wind [74], and the "Industry 4.0" initiative, which focuses on incorporating edge sensors with ultra-low latencies in manufacturing environments [45]. An overarching requirement for extreme edge devices is that they must be able to operate for years or decades without maintenance due to the difficulty of accessing them and the impracticality of replacing billions of tiny batteries.

This dissertation concerns a class of AI models called **weightless neural networks** (WNNs) and their potential to enable fast, accurate, and energy-efficient inference on extreme edge devices. The defining characteristic of WNNs is that they perform the vast majority of their computation through table lookup operations [12]. In DNNs, neurons operate by computing dot products between real-valued vectors of input activations and learned weights (Figure 1.1a). This process can be reduced into a series of multiply-accumulate (MAC) operations. Binary neural networks (BNNs), a related class of model, instead use Boolean values for weights and activations. The equivalent to the dot product in a BNN is a bitwise XNOR operation, followed by counting the number of '1' bits (a population count, or popcount, operation) (Figure 1.1b). This eliminates the need for multiplication, which is costly in hardware, but still requires addition for the popcount. Like BNNs, the neurons in WNNs have binary inputs and outputs. However, rather than applying parameterized arithmetic or logical functions to their inputs, WNNs concatenate them to form integer addresses, and use these addresses to index lookup tables (LUTs) (Figure 1.1c). Hence, a neuron in a WNN with n inputs will form an n-bit address, index a LUT with 2^n entries (one for each combination of inputs), and can represent any one of 2^{2^n} unique Boolean functions (one for each combination of LUT entries).

WNNs can represent complex behaviors using shallow models and few neurons.



(a) DNNs use multiply-accumulate operations.

(b) BNNs use XNOR-popcount operations.

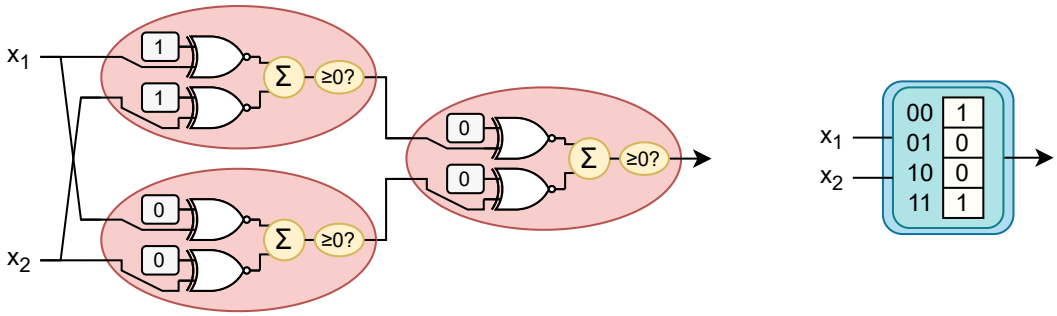
(c) WNNs use concatenate-lookup operations.

Figure 1.1: Illustrations of neurons in DNNs, BNNs, and WNNs.

For instance, as shown in Figure 1.2a, the smallest possible BNN to implement a two-input Boolean XNOR function requires a total of three two-input neurons forming two distinct layers.[1] On the other hand, for the WNN in Figure 1.2b to implement the same function, all that is needed is a single neuron composed of a two-input LUT. The ability to use shallower models makes WNNs attractive for ultra-low-latency applications, since the critical path of functional units that data must flow through is shortened. Lookup tables are also efficient and straightforward to implement in hardware—in fact, field-programmable gate arrays (FPGAs), which are commonly used to prototype and implement accelerators, are largely composed of reconfigurable LUTs. Furthermore, the inherent non-linearity of the LUTs which compose WNNs gives them an interesting parallel with biological neurons. While the dendritic trees of cortical neurons were once believed to serve little computational function, they are now known to have complex non-linear behaviors, with individual dendrites observed implementing the XOR function [4]. Biological neurons are extraordinarily energy-efficient [121], so this similarity makes WNNs worth exploring.

Unfortunately, there are some very significant drawbacks to existing WNNs that have so far relegated them to niche use cases. Since the number of entries in a LUT grows exponentially with the number of inputs, constructing large, densely-

[1]By convention, BNNs treat '0' inputs as -1 during the popcount, which therefore computes a majority function treating ties as '1'. In Figure 1.2a, the two neurons on the first layer compute $\text{maj}(x_1, x_2) = x_1 \vee x_2$ and $\text{maj}(\neg x_1, \neg x_2) = \neg(x_1 \wedge x_2)$. The neuron on the second layer computes $\text{maj}(\neg(x_1 \vee x_2), \neg(\neg(x_1 \wedge x_2)) = \neg(x_1 \vee x_2) \vee (x_1 \wedge x_2) = \neg(x_1 \oplus x_2)$.



(a) A BNN for $\neg(x_1 \oplus x_2)$.

(b) A WNN for $\neg(x_1 \oplus x_2)$.

Figure 1.2: DNNs and BNNs require multiple neurons and layers of computation to represent most Boolean functions. As shown here, a BNN implementing the XNOR of two inputs requires three neurons and two layers, while a WNN is trivially constructed using a single LUT-2 neuron.

connected WNNs is impractical. For instance, a LUT which could be indexed by an entire sample from the MNIST [92] dataset (a 28×28 image) using one bit per pixel would require a capacity of $2^{28\times28}$ bits, or about 10^{217} exabytes. Therefore, WNNs must use very sparse connectivity and/or data compression schemes to be feasible. An additional issue is that their accuracies tend to lag far behind those of DNNs and BNNs. Overall, while weightless model architectures seem well-suited for extreme edge devices in concept, there would need to be a great deal of improvement over prior work in both their parameter sizes and their accuracies for this to be realistic. The objective of this dissertation is therefore to achieve these improvements through structural changes, new training strategies, and efficient accelerator architectures.

1.2 Thesis Statement

By leveraging the nonlinearity of lookup tables, weightless neural networks can be used to enable high-throughput, low-latency, low-energy inference on edge devices. This is made possible through significantly improving on every aspect of the prior work using a combination of new model architectures, improved training strategies, and efficient hardware accelerator and microcontroller implementations.

1.3 Contributions of this Dissertation

This dissertation proposes novel techniques for the composition, training, and deployment of weightless neural networks. These contributions can be divided under the following three main topics:

1. **Improved Compression and Encoding for Weightless Neural Networks [140]:** I first propose a weightless model which resolves an incompatibility between leading techniques from prior work and introduces novel algorithmic improvements which allow for more accurate training and more efficient inference. In addition, I introduce an FPGA-based inference accelerator architecture for deploying models. This work, which I call **BTHOWeN**, outperforms the state-of-the-art Bloom WiSARD [130] WNN in accuracy and parameter sizes and yields more efficient hardware than quantized DNNs based on `hls4ml` [50].

2. **Multi-Pass Learning with Weightless Ensembles [137, 138, 139]:** I next introduce a model which leverages a novel gradient-based multi-pass WNN learning rule, a submodel ensembling technique, and a weightless pruning strategy to achieve further improvements in accuracy and parameter footprint. By using a sparsity-aware accelerator architecture with input data compression, this work, **ULEEN**, outperforms fully-connected FINN [145] BNNs in latency, throughput, and energy efficiency, with equal or better accuracy.

3. **Multilayer Weightless Neural Networks [22]:** Lastly, I radically rethink how WNNs are structured by introducing a model which incorporates multiple layers of directly-chained lookup tables, and explore learning rules that enable propagation of gradients through table indexing operations. This restructuring eliminates costly hash computation, granting order-of-magnitude improvements in energy efficiency and accelerator hardware area. These models, **DWNs**, have parameter sizes so minuscule that they can be deployed on a microcontroller

with just 2 KB of RAM, demonstrating their suitability for TinyML [2] scenarios which focus on using inexpensive commercial off-the-shelf hardware.

1.4 Organization of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides relevant background information on WNNs, methods to optimize DNNs for the edge, and some other prior approaches to efficient machine learning. Chapter 3 describes the tools, libraries, and processes I used to develop software models for training and edge implementations for inference, as well as the metrics and datasets I used for evaluation. Chapter 4 presents the BTHOWeN model and its associated inference accelerator. Chapter 5 describes the ULEEN model, which introduces further techniques to improve the efficiency of WNNs in both software and hardware. Chapter 6 introduces the DWN model, which uses multiple layers of LUTs to enable ultra-efficient inference on both FPGAs and microcontrollers. Finally, Chapter 7 concludes the dissertation and discusses some potential avenues for future work.

Chapter 2: Background and Related Work

This chapter begins with an introduction to weightless neural networks, including both the field as a whole and several specific models that are relevant to understanding my contributions in this dissertation. I next discuss some of the approaches that are used to optimize deep neural networks for efficient inference, as well as closely-related models such as binary neural networks. Lastly, I briefly summarize some approaches to efficient machine learning that are not based on DNNs or WNNs.

2.1 Weightless Neural Networks

Weightless neural networks (WNNs) are a class of neural model which use lookup tables (LUTs) as their fundamental computational units. WNNs operate using the **concatenate-lookup operation**: a vector of inputs is concatenated to form an integer address, and this address is used to index into a LUT, yielding a response. Both inputs and responses are usually binary. While the entries of the LUTs are learned parameters, these models are "weightless" in the sense that the parameters do not act as multiplicative scalars on input activations, which is the defining characteristic of weights in perceptrons [60] and their descendant models including deep neural networks (DNNs) and binary neural networks (BNNs).

WNNs are of theoretical interest due to their ability to represent complex behaviors with small, shallow models. The Vapnik–Chervonenkis (VC) dimension of an n-input LUT is 2^n, while that of an n-input neuron in a DNN or BNN is just $n+1$. This indicates that LUTs are superior in their ability to represent complex functions,[1] particularly when those functions are not linearly separable. For instance, the XNOR function can not be modeled using a single neuron in a BNN, but is

[1] More formally, the VC dimension of a model is equal to the cardinality of the largest set of points that can be correctly classified by the model regardless of their labels.

trivially implementable using a two-input LUT in a WNN (see Figure 1.2).

2.1.1 Origins of Weightless Neural Networks

The first weightless models emerged from early experiments in optical character recognition. Bledsoe and Browning proposed the original "n-tuple pattern recognition method" for categorizing handwritten or typed alphanumeric characters by projecting them onto a 10×15 grid of photoresistors [29]. Photoresistors were randomly combined into 75 pairs, and their outputs were binarized to form 2-bit values. The values of these pairs were then recorded for a reference sample of each character. When a new, unknown character was presented to the model, its pair values were computed and compared against those of each of the reference characters. The character with the most pair values in common was then taken as the prediction. This method proved very effective for scanning typed text (where the variation between multiple instances of the same character was minimal), but struggled with recognizing handwriting, where letters can vary more widely in size, shape, and position.

Variants of this experiment changed the number of photoresistors in each tuple from n=1 to 5, rather than just using pairs (n=2), and explored training using multiple sets of characters. When using multiple sets, the model determined whether the value of each tuple was shared with *any* instance of a particular character in the training data, which was accomplished by using LUTs to store the observed values of tuples. An important observation in this work was that the optimal value of n tended to increase with the size of the training set. For very small n, model performance was found to *degrade* as the amount of training data increased. This was because too many unique values were observed for each tuple during training, meaning that during inference, a sample would appear to be very similar to all characters. This "saturation" phenomenon therefore made determining the correct class impossible. Using a large tuple size with little training data resulted in the opposite problem: the values of most tuples during inference were not associated with *any* output class.

The hardware limitations of the computers of this era restricted the scope and complexity of the experiments that could be performed with early WNNs. Nonetheless, subsequent works explored extensions such as real-valued LUTs which weighted tuple values by their frequency [28] and tuples of up to n=24 [144]. Notably, the association between training set size and optimal n was found to eventually flatten out, suggesting that an excessively large n results in the memorization of dataset noise rather than useful generalization.

2.1.2 The WiSARD Classifier

The next major advancements in WNNs were driven by improvements in computer memory capacity, processing speed, and availability of training data. This progression was epitomized by the WiSARD (Wilkie, Stonham, and Aleksander's Recognition Device) [10] classifier. WiSARD is a canonical weightless model that serves as the foundation for a large volume of subsequent work. A WiSARD model is composed of n-input, 2^n-entry lookup tables with learned 1-bit entries, which are also referred to as **RAM nodes**. As Figure 2.1 demonstrates, binary model inputs are randomly assigned to RAM nodes, which is similar to the construction of tuples in the earlier work. One set of RAM nodes is constructed for each output class in the dataset. These class-specific sets of RAM nodes are known as **discriminators**, and typically share the same random mapping of inputs (i.e., the RAM node at index i in discriminator d_1 will have the same inputs as the RAM node at index i in discriminator d_2).

WiSARD is trained using a single-pass learning rule, meaning each training sample is only presented to the model once. As shown in Figure 2.2a, all RAM node entries are first initialized to 0. Next, training samples are sequentially presented to the discriminators corresponding to their labeled classes; for instance, Figure 2.2b shows an input image containing the digit '0' being presented to Discriminator 0, and Figure 2.2c shows an image containing '1' being presented to Discriminator 1. The binarized values of the input pixels are combined to form RAM node addresses

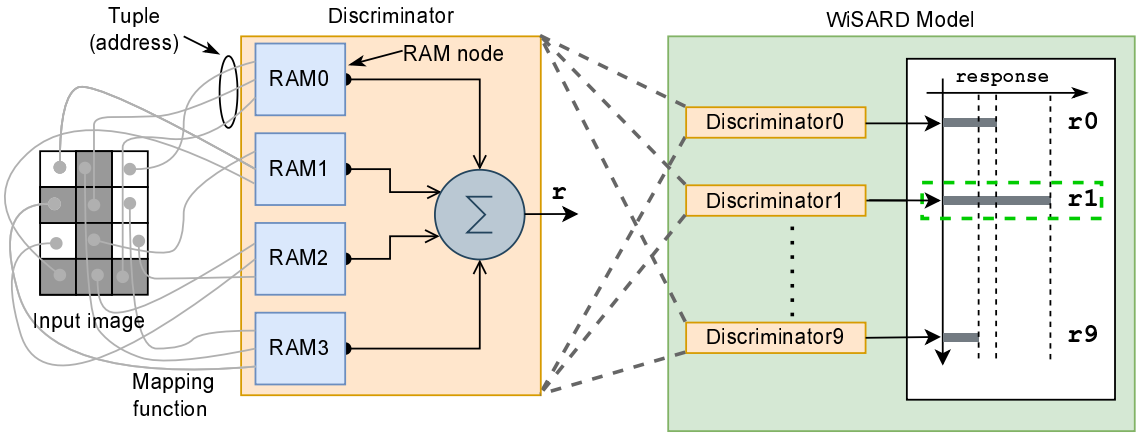


Figure 2.1: WiSARD, a simple WNN model for classification tasks. WiSARD contains separate sets of RAM nodes, known as discriminators, for each output class.

according to the model's mapping function. Next, each RAM node in the target discriminator is indexed using its computed address, and the entry thus accessed is set to 1. Unlike the simplified depiction in Figure 2.2, a real WiSARD model repeats this process using many training samples for each class. However, presenting the same training sample to a discriminator multiple times has no additional effect, since after the first exposure, all accessed RAM node entries are already set to 1. Thanks to the simplicity of this single-pass learning rule, WiSARD has been observed to train up to four orders of magnitude faster than DNNs and support vector machines [33].

To perform inference, WiSARD presents a sample to all discriminators. The values of the accessed RAM node locations within each discriminator are then summed, producing a set of **response scores**. The index of the discriminator with the largest response score is then taken to be the predicted class. Since RAM nodes only output a 1 during inference when they are presented with a stimulus that was seen during training, one might expect WiSARD to generalize poorly to new data. Indeed, individual RAM nodes have no ability to generalize, and a WiSARD model consisting of a single enormous LUT for each class would be useless. However, since each RAM node in a typical WiSARD model is only sensitive to a small subset of inputs, it is

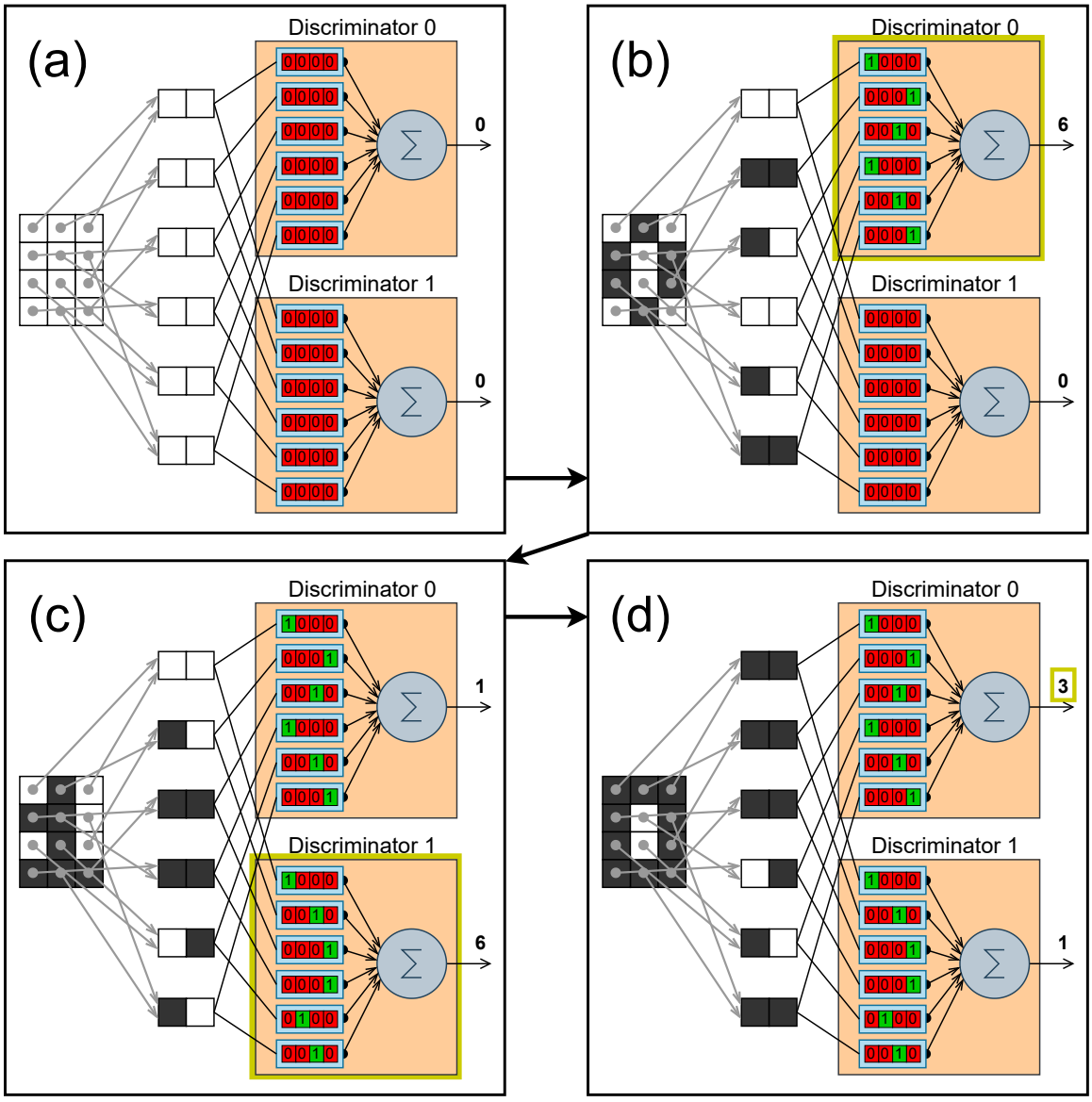


Figure 2.2: Training a WiSARD model. (a) All RAM node entries are initially 0. (b,c) Training samples are presented sequentially to the discriminators corresponding to their labeled classes, and accessed RAM node entries are set to 1. (d) For inference, a sample is presented to all discriminators, which sum their RAM node outputs. The index of the discriminator with the greatest sum is taken as the predicted class.

highly likely that at least some tuple values seen by individual RAM nodes during inference will be identical to ones seen during training. So long as more RAM tuples are shared with the discriminator corresponding to the correct output class than with any other discriminator, the model will still output a correct prediction. For instance, in Figure 2.2d, a new sample of the digit '0' is presented to the model for inference. Although just 3/6 tuple values are shared with the training data in Discriminator 0 (giving a response score of 3), only a single value is shared in Discriminator 1.

The number of inputs to each RAM node (i.e., the tuple size), n, is a crucial hyperparameter for WiSARD models. Small values of n limit the complexities of the behaviors each RAM node can learn. As observed in earlier work, this helps to prevent overfitting when available training data is limited, but can result in lower accuracy otherwise. Excessively low values of n also make WiSARD vulnerable to saturation. Larger values of n improve accuracy (with diminishing returns) so long as the dataset is of sufficient size and diversity to avoid overfitting, but exponentially increase the size of the model: a WiSARD model with d classes and N inputs has a parameter size of $d\lceil \frac{N}{n} \rceil 2^n$ bits.

WiSARD can learn sophisticated behaviors despite only having a single layer of learnable parameters due to the nonlinearity of its LUT-based RAM nodes, which can each represent any of the 2^{2^n} Boolean functions of their inputs. WiSARD is reasonably straightforward to implement in hardware: since computation is performed using lookup tables, the majority of the model can be mapped to commodity memory modules. This approach was used for the development of the WISARD/CRS1000 image processing neurocomputer, which was sold commercially in the mid-1980s for applications such as automated defect detection.

2.1.3 Improving WiSARD

While the structural simplicity and rich representational capability of WiSARD make it attractive in theory, it is unfortunately not competitive with DNNs

in accuracy or model parameter sizes. However, subsequent works have explored methods to improve on both of these aspects.

2.1.3.1 Bleaching

As discussed earlier, saturation is a major problem for smaller WiSARD models. With a sufficiently large or noisy dataset, most RAM node entries are set to 1 during training, and the model loses its ability to effectively discriminate. Some early (pre-WiSARD) WNNs attempted to address this issue by storing how often specific tuple values occurred in the training data, rather than solely *whether* they occurred, and used this information to weight RAM node outputs during inference. However, this approach was found to perform poorly for all but very simple models [144].

Bleaching [35] is a simple but effective strategy to avoid saturation in WiSARD. The bleaching strategy replaces the single-bit RAM node entries with counters, which are incremented each time a tuple value is seen during training. During inference, a bleaching threshold b is determined to binarize the RAM, with counter values greater than or equal to b interpreted as 1 and values less than b interpreted as 0. Figure 2.3 shows a simple example of this: when b is chosen to be equal to 15, all patterns seen fewer than 15 times during training are discarded.

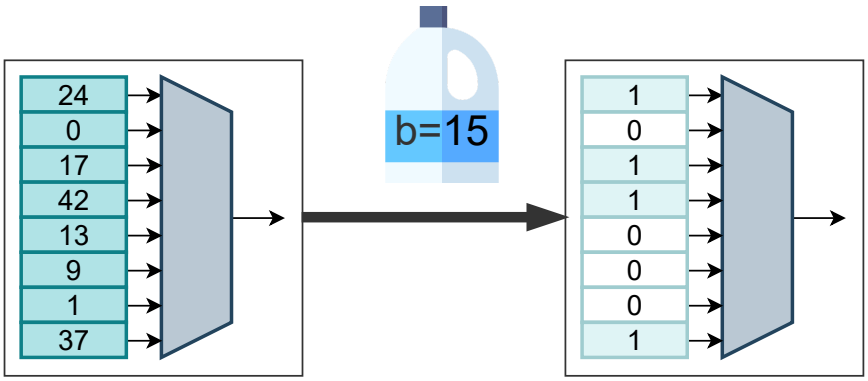


Figure 2.3: The bleaching process replaces binary RAM node entries with counters, which are incremented each time they are accessed during training. After training, a threshold b is used to binarize counter values.

Several strategies have been explored for choosing b. Bleaching can be used dynamically to break ties during inference by increasing b whenever the top two discriminator responses are close to equal, or it can be applied statically. A static choice of b avoids the potential need to run inference multiple times for a single sample. It also allows the counter tables to be permanently replaced with binary RAMs, which reduces the model's memory footprint. However, a static choice of b which is too small will not eliminate saturation, while a choice which is too large will result in poor performance since only the most common patterns will be retained. Therefore, many candidate values of b should be evaluated for static bleaching.

2.1.3.2 Thermometer Encoding

WiSARD conventionally binarizes inputs by comparing them against their mean values in the training data. However, a great deal of information is lost with this approach. Multi-bit binary integer encodings, like those used for quantized DNNs, are poor choices for WNNs since each bit is treated independently when forming addresses. For instance, the least significant bit of an 8-bit integer, taken in isolation, does not provide any meaningful insight into the integer's value, and should not be used in addressing. Multi-bit unary "thermometer" encodings [80] are instead the preferred approach for WNNs. A thermometer encoding compares a value against a series of increasing thresholds, setting input bits from least to most significant as progressively more thresholds are surpassed. As shown in Figure 2.4, this technique is conceptually similar to (and named after) mercury passing the lines on an analogue thermometer.

Nonlinear Thermometer Encoding: Thermometer encoding thresholds are usually picked to split the value range for an input into equal intervals. However, this may not be ideal if an input follows a known, nonuniform distribution. For instance, many processes in particle physics follow distributions that are approximately Gaussian. A

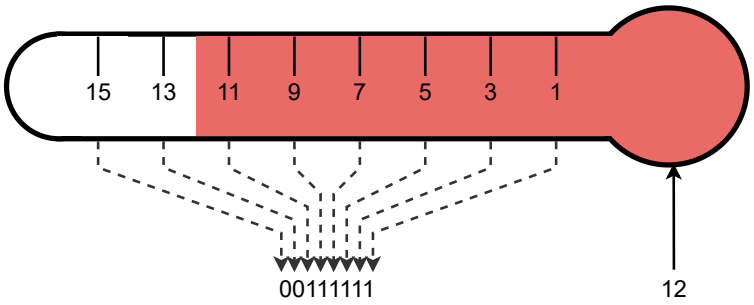


Figure 2.4: In a unary thermometer encoding, an input is compared against a series of increasing thresholds, with result bits set from least to most significant.

prior work [157] exploring WiSARD in this domain achieved significant accuracy benefits by using nonlinear separations between thresholds.

2.1.3.3 Bloom WiSARD

While increasing the tuple size n can improve the accuracy of WiSARD models, it also exponentially increases their memory footprint. However, large RAM nodes tend to be highly sparse, meaning most entries are still 0 after training. Therefore, rather than implementing RAM nodes using LUTs, Bloom WiSARD [130] proposes using hash-based data structures such as Bloom filters as RAM nodes. An example discriminator with RAM nodes implemented in this way is shown in Figure 2.5.

A Bloom filter [30] is composed of a small lookup table and a set of independent hash functions. During training, inputs to the filter are hashed using each of these functions to form a set of addresses, and all indicated locations in the LUT are set to 1. During inference, the Boolean AND of the accessed locations is taken as the filter output. Bloom filters are approximate data structures, meaning they will sometimes produce erroneous output. They will never produce false negatives, so they always output 1 in response to patterns seen during training, but they are susceptible to false positives. The false positive rate for a Bloom filter is a function of the size of the RAM, the number of hash functions, and the number of patterns stored [63]. This creates a tradeoff between Bloom filter LUT size and model accuracy. In practice,

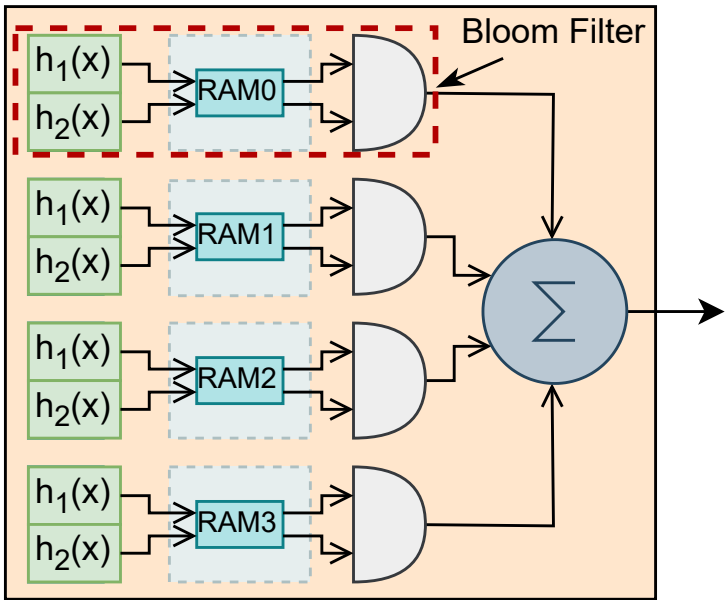


Figure 2.5: Bloom WiSARD uses Bloom Filters, rather than LUTs, as RAM nodes. This introduces additional complexity from hash computation and risks RAM nodes emitting false positives, but can greatly reduce the model's parameter size.

Bloom WiSARD can frequently decrease the parameter size of a WiSARD model by 2000× with minimal impact on accuracy.

2.1.3.4 Results for Prior Work

Figure 2.6 compares the baseline WiSARD model and weightless models using the three enhancements discussed against DNNs and BNNs on the MNIST [92] handwritten digit recognition dataset. As this figure shows, while prior work has been quite effective in reducing the parameter size of WiSARD and somewhat effective in improving its accuracy, there is still a wide gap from the weighted models. Therefore, there is a clear need for additional work in this domain.

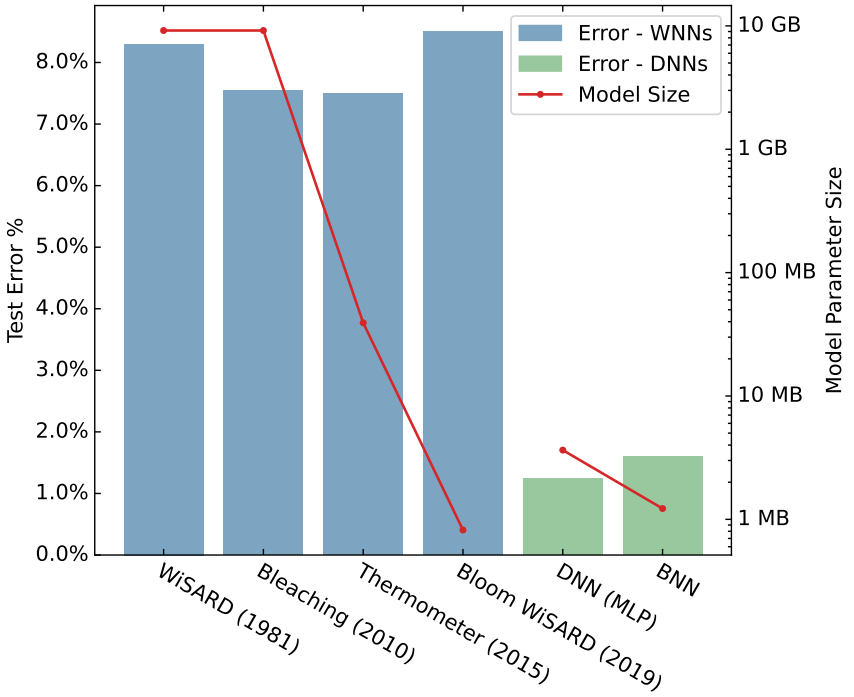


Figure 2.6: Performance of prior WNNs versus fully-connected DNN and BNN models on the MNIST [92] dataset.

2.1.4 Other Weightless Neural Models

So far, this section has focused primarily on WiSARD and related models. While these are the most common types of WNNs, there are many other approaches that have been explored [12, 101], several of which are relevant to this dissertation.

2.1.4.1 Probabilistic Logic Nodes

The Probabilistic Logic Node (PLN) [6, 9] introduces a new "unknown" u state for each RAM node entry, which is interpreted as a 0 or a 1 with probability 0.5 during training. All RAM nodes are initially filled with u. The goal of training is to replace all instances of u with a 0 or a 1, making inference deterministic.

A key structural feature of PLNs is the **pyramid**, shown in Figure 2.7. A pyramid consists of a hierarchy of RAM nodes connected as a tree. This helps to

avoid state explosion while still enabling the pyramid to be sensitive to all inputs. For instance, implementing a 64-input LUT requires 2,097,152 TiB of storage. On the other hand, a three-layer pyramid of LUT-4s with 64 inputs only requires 42 bytes of storage. Although a pyramid clearly can not represent any function of its inputs, multiple pyramids can be used with different connectivities to compensate for this.

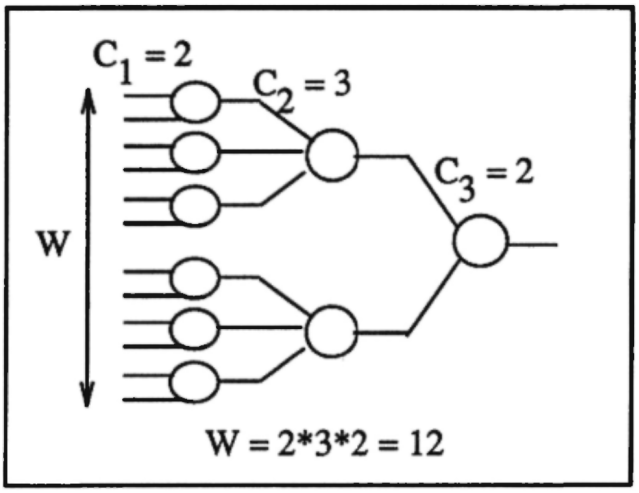


Figure 2.7: A pyramidal structure of RAM nodes, which is used to reduce parameter sizes in models such as PLNs and MPLNs. Figure from [6].

Multi-Level PLNs: The Multi-level PLN (MPLN) [108] extends the PLN by allowing for multiple "unknown" states with different probabilities during training. As with PLNs, inference is deterministic.

2.1.4.2 Goal-Seeking Neurons:

Unlike the PLN and MPLN, the Goal-Seeking Neuron (GSN) [55] allows u values to be directly output by RAM nodes during training. A u in an address is treated as both a 0 and a 1; for instance, the address $0u1u$ would expand to {0010, 0011, 0110, 0111}. RAM nodes in a GSN model output a 0 or a 1 if all addressed entries are equal to that value, and a u otherwise.

2.1.4.3 Training

Multiple training strategies have been proposed for multi-layer models such as PLNs, MPLNs, and GSNs. Many are based on backwards depth-first search strategies [6, 55] which seek to iteratively replace u values with 0 or 1 to maximize model accuracy. Other approaches, such as generalizing RAMs [11], seek to extrapolate to patterns that were not seen during training by looking at the behavior of the model in response to other patterns at short Hamming distances. Unlike the single-pass learning rule used for WiSARD, these strategies often make multiple passes through the data, adjusting values in the RAM nodes to satisfy predicates until some termination condition is met (such as all u values being eliminated). Note that these are very different from the multi-pass gradient-based learning rules which are used to train models such as DNNs, as they make discrete rather than continuous updates.

Unfortunately, these approaches to training are often very complex and have never been fully developed to reasonable accuracies. Most works in this domain are only evaluated on toy problems, and others underperform WiSARD on real datasets. For instance, one work [112] based on MPLNs achieved just 84% accuracy on a simplified, four-class variant of MNIST.

2.1.5 Weightless Neural Networks on Edge Devices

There have been several prior works which explored WNNs in edge contexts [7, 112, 142]. While many of these works were targeted at narrow scopes, one work [53] explored the feasibility of implementing an FPGA-based hardware accelerator for training and inference with WiSARD. This design used hash tables to reduce the memory footprint of the model, though unlike Bloom WiSARD it included mechanisms for hash collision detection. This accelerator achieved 90.73% accuracy on the MNIST dataset, slightly lower than Bloom WiSARD.

2.2 Deep Neural Networks

Deep neural networks (DNNs) span a diverse range of model architectures, including multi-layer perceptrons (MLP) [125], convolutional neural networks [91], and transformers [149]. DNNs are the dominant approach to machine learning today, and there are many strategies which have been explored to optimize them.

2.2.1 Optimizing DNNs for Edge Inference

Neurons in DNNs compute dot products between a data-dependent input activation vector and a learned weight vector. Therefore, the fundamental mathematical operation in a DNN is the **multiply-accumulate** (MAC), $c \mathrel{+}= a \cdot b$. Neurons are composed into layers, which are stacked interspersed with other layers such as nonlinear activations to form full models. Reducing the cost of a single MAC, the number of MACs per neuron, the number of neurons per layer, and the number of layers in a model are all viable ways to optimize DNNs for more efficient inference.

2.2.1.1 Quantization

DNNs were historically implemented using 32-bit floating point (FP32) weights and activations. However, more recent work has demonstrated that the range and precision of this data type are greater than what is actually needed. For instance, the high-throughput tensor cores of the NVIDIA H100 GPU natively support the 19-bit TF32, 16-bit FP16 and BF16, and 8-bit E3M4 and E5M2 floating-point data formats; notably, they do *not* support FP32 [114]. While training with FP8 weights and activations requires some modifications to the backpropagation algorithm [126, 154] normally used to train DNNs in order to limit the dynamic range of gradients [115], it yields a 4× reduction in memory footprint and 4× increase in peak throughput versus TF32.

Quantization is a technique by which a DNN is modified to use simpler data types for weights and activations after training in order to reduce the memory foot-

print and computational cost of inference [62]. Generally, inference with DNNs needs less numerical precision than training, since it is no longer necessary to represent gradient-based updates in the backpropagation phase [124]. 8-bit integers (INT8) are a common choice [75, 77], and reduce hardware area and energy by ∼50% versus FP8 [148]. Cutting-edge techniques enable INT4 [41, 66] and even INT2 [36] quantization. These smaller data types further reduce hardware area and inference energy. However, quantizing this aggressively can significantly reduce accuracy.

2.2.1.2 Pruning

Pruning is a process which eliminates connections within a DNN by forcing weight or activation values to 0 [8, 103, 113, 163]. Since the product of any number with 0 is 0, and any number plus 0 is itself, a MAC operation can be skipped if one of its inputs is known to have been pruned. Specialized hardware accelerators can exploit this data sparsity to improve efficiency [79, 114, 164]. More aggressive approaches to pruning DNNs aim to eliminate convolutional filters [69], or even entire layers [32, 52, 128]. With these approaches, a single, large model can be trained once, then progressively cut down to target different design points for inference.

Pruning is usually performed after training, and potentially followed by an additional fine-tuning pass. However, some works have explored dynamic pruning during training [89, 97], or even at initialization time [46, 81, 151].

2.2.2 Binary Neural Networks

Binary neural networks (BNNs) [43, 58, 123] are the limit case of quantization, as they use single-bit values for weights and activations. Binarizing weights and activations to {-1, +1} simplifies multiplication to an XNOR gate. BNNs take the XNOR of their input activations with a learned binary vector, sum this vector, and compare the result against a learned or fixed threshold. These "XNOR-popcount" operations are much more efficient than MAC operations in both energy and hardware

area. However, BNNs generally require 2–11× more parameters and operations to reach the same accuracy as a full-precision DNN [145].

Ternary Weight Networks: Ternary Weight Networks (TWNs) [13, 94, 99, 166] are a closely related category of model that constrain weights and activations to {-1, 0, +1}. The accuracy of TWNs has been shown in some cases to be only slightly worse than full-precision DNNs.

2.2.2.1 Training BNNs with Straight-Through Estimators

When developing a DNN with very low precision, it is difficult to apply quantization as a post-training operation without causing a large drop in accuracy. Instead, it is often better to use quantized weights and activations during training as well. However, training a neural network using backpropagation requires weights to be continuous-valued to enable gradient-based updates. To circumvent this issue, BNNs store weights internally as floating-point values between -1 and 1, and binarize them using the sign function (Equation 2.1). However, this introduces a new problem: the derivative of the sign function (Equation 2.2) is poorly behaved, as it is 0 almost everywhere and infinite at $x = 0$. This means that, during backpropagation, gradients that are passed backward through the sign function to the weights will either be cancelled to 0 or explode to $\pm\infty$.

$$\text{sign}(x) = \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases} \quad (2.1) \qquad\qquad \text{sign}'(x) = \begin{cases} +\infty & x = 0 \\ 0 & x \neq 0 \end{cases} \quad (2.2)$$

To resolve this issue, BNNs employ an approach known as the **straight-through estimator** (STE) [161]. The STE function behaves identically to the sign function during the forward training pass (Equation 2.3). However, its derivative (Equation 2.4) is different, being instead equal to that of the HardTanh function. This "proxy gradient" approach, though mathematically inaccurate, prevents gradi-

ent cancellation, and has been shown both empirically and theoretically to enable BNN convergence.

$$\mathrm{STE}(x) = \begin{cases} -1 & x < 0 \\ 1 & x \geq 0 \end{cases} \quad (2.3) \qquad \mathrm{STE}'(x) = \begin{cases} 1 & |x| \leq 1 \\ 0 & |x| > 1 \end{cases} \quad (2.4)$$

2.2.2.2 Hardware Implementations of BNNs

Thanks to the computational simplicity and small parameter sizes of BNNs, they are appealing models for targeting energy-efficient hardware implementations. Both FPGA and ASIC implementations exist [16, 95]. Notably, Xilinx has developed the Brevitas [118] library for training low-precision quantized and binary neural networks, and the FPGA-based FINN [145] platform to deploy BNNs for inference.

2.2.2.3 Other Approaches to Low-Precision Training

While the STE is a common approach for training ultra-low-precision models such as BNNs, it is not used universally. For instance, alpha-blending [100] represents weights as an affine combination of the unquantized (full-precision) and quantized versions of underlying learned parameters. During training, the quantized representations of weights are progressively favored by gradually increasing the non-learned parameter α from 0 to 1. For instance, when training a BNN using alpha-blending, a weight is given by $(1-\alpha)\, w + \alpha \,\mathrm{sign}(w)$, where w is the trained parameter. This allows a fraction $(1-\alpha)$ of the gradient to be backpropagated to w.

2.2.3 Tabularization of DNNs

Other approaches to optimizing DNNs replace arithmetic operations with table lookups. After training a model, partial results can be precomputed and stored in LUTs to be used during inference. A variety of methods have been proposed.

2.2.3.1 LogicNets

Xilinx's LogicNets [146] model is a sparsely-connected DNN constructed out of neuron equivalent (NEQ) units. An NEQ has γ inputs of β-bit precision and a single β-bit output. The NEQ has learned internal weight and bias values with full FP32 precision, and quantizes its output using the STE.

As each input has 2^β possible values, each NEQ can be represented using a lookup table (LUT) with a total size of $\beta 2^{\beta\gamma}$ bits. The converted NEQs can then be combined into a netlist and passed to an FPGA synthesis tool. Logic optimizations performed during synthesis allow the final hardware area to be much smaller than a naive implementation of the LUTs.

Some subsequent works [17, 18] have expanded on this model by using more sophisticated structures (such as multi-layer DNNs) within NEQs, while maintaining their sparse, low-precision connectivity externally. LogicNets also bears strong similarities to some much earlier work [31, 54] on tabularizing sparse DNNs.

2.2.3.2 LUTNet

LUTNet [152] proposes a "logic expansion" scheme which transforms the XNOR operators in a ternary weight network into LUTs. The first input to a LUT is chosen to be the original input activation for the XNOR gate it replaced, while the rest are picked randomly. These LUTs are then fine-tuned in an additional training pass, with their entries treated as the coefficients of a Lagrange interpolating polynomial. This process was found to give FPGA area and accuracy superior to conventional BNNs.

2.2.3.3 Approximate Matrix Multiplication

Approximate matrix multiplication (AMM) techniques convert the dot product between a weight and activation vector into a series of table lookups. Drawing from the Product Quantization (PQ) [78] method for approximate nearest neigh-

bor search, AMM techniques split an input activation vector into disjoint subspaces, identify the nearest neighbor from a set of "prototype" vectors in each subspace, and then sum the partial dot products for each prototype. If this technique is used for inference, the weight vector is fixed, and therefore the partial dot products can be precomputed and stored in lookup tables.

The MADDNESS [27] algorithm for AMM uses K-means clustering on input activations across the training data to identify prototype vectors. Earlier techniques, such as PQ, used Euclidean distance to identify the nearest prototype to the activation vector within each subspace. However, this introduces a significant number of multiplications. Instead, MADDNESS uses a locality-sensitive hashing technique which uses learned parameters but is arithmetic-free at inference time. It also introduces a lightweight technique for optimizing prototypes based on ridge regression. Overall, the only arithmetic operations MADDNESS requires during inference are the summations of partial dot products across subspaces. Recent work [162] refined this technique for multi-layer and transformer-based DNN architectures, and used it to develop a hardware prefetcher that outperformed state-of-the-art approaches.

2.2.3.4 Direct Conversion

Quantized DNNs with very few inputs can be converted directly into lookup tables. However, this is not a general-purpose technique since its memory footprint scales exponentially with the number of inputs to the model. This technique was applied to image upscaling by restricting the receptive field of the upscaling kernel to 2–4 pixels, resulting in a final model size of <1.3 MB [76]. The resultant model outperformed traditional methods of image upscaling such as bilinear filtering with comparable latency, though it could not match the performance of (much slower) large neural models.

2.3 Other Approaches to Efficient Machine Learning

There have been countless approaches proposed for efficient machine learning which are not based on DNNs or WNNs. This section lists a few examples.

- **XGBoost** [39] is based on ensembles of decision trees. It is notable for its scalability and high performance, and gives excellent results for many applications based on tabular data.

- **Hyperdimensional computing** [61] represents data using high-dimensional random "hypervectors", so that the cosine similarity between any two unrelated hypervectors is approximately 0. Transformations are applied to hypervectors to manipulate and combine data points. Classification can be performed by learning class hypervectors, then determining which of them is most similar to a given inference sample.

- **DiffLogicNet** [120] learns networks of two-input logic gates. It uses a gradient-based learning rule where each functional unit maintain weights for each of the 16 possible two-input Boolean functions it could represent, which are converted into probabilities using a softmax function. During inference, functional units are converted into the gate with the highest associated probability.

- **Tiny Classifier circuits** [73] are networks composed of a small number (50–300) of two-input NAND gates. The connectivity between gates is learned via graph-based genetic programming. Despite their tiny sizes, these models are competitive with XGBoost on many datasets.

- **Tsetlin Machines** [64] are networks composed of Tsetlin automata, which learn conjunctive clauses of arbitrarily many variables. Clauses are assigned positive or negative polarity, with majority voting used to determine outputs. Unlike RAM nodes in WiSARD, groups of Tsetlin automata can learn Boolean functions of arbitrarily many inputs without encountering state explosion. However, each automaton is only capable of learning a single conjunctive clause.

Chapter 3: Methodology

This chapter provides an overview of the specific processes I used to develop, deploy, and evaluate weightless models and accelerator architectures. I first discuss the software tools and techniques I used to train models and convert them to more efficient forms for inference. Next, I discuss the FPGA and microcontroller deployment scenarios that I explored for these models. Lastly, I discuss the metrics and datasets which I used for evaluation.

3.1 Model Development Methodology

I wrote the code for training models primarily in Python in order to leverage its excellent ecosystem support for scientific computing and machine learning applications. This enabled me to iterate rapidly on model prototypes. Once models were trained, I converted them into more efficient forms for edge inference. To accomplish this, I used a templated metaprogramming strategy to semi-automatically instantiate SystemVerilog (for FPGA deployment) or C++ (for microcontroller deployment) code which implemented the trained model without external dependencies.

3.1.1 Training Weightless Models

My first weightless model, BTHOWeN (Chapter 4), uses a single-pass learning rule related to the method used by WiSARD. I implemented training for BTHOWeN using the NumPy [67] library for scientific computation. NumPy provides a high-level API based on multidimensional "ndarray" objects, which invokes a C backend to perform logical and arithmetic operations. This backend has been extensively optimized to take advantage of features such as x86 SIMD extensions, which makes NumPy highly performant; in fact, a prior work observed that benchmarks rewritten in Python with NumPy were sometimes faster than their native C versions [3].

I used the Numba [90] library to further optimize key kernels. Numba is a JIT compiler for Python which focuses on supporting only a narrow subset of language features—in particular, array-oriented computation using NumPy ndarrays. It transforms Python bytecode into an LLVM intermediate representation, which is then used to produce machine code. The advantage of Numba is that it completely eliminates the overhead of the Python interpreter for compiled kernels, which can otherwise be a bottleneck for NumPy applications. However, it has a very restrictive syntax which often requires significant restructuring of code, which may not be worth the effort for non-critical functions.

My subsequent models, ULEEN (Chapter 5) and DWN (Chapter 6), use backpropagation-based learning rules. Therefore, I developed the code for training these models by extending the PyTorch [19] library. PyTorch is a highly performant library for machine learning with GPU acceleration. It provides a powerful "autograd" engine for automatic calculation of derivatives of functions. However, since PyTorch is primarily intended for use with DNNs, it lacks the WNN-specific functionality that I needed for my research. Thankfully, PyTorch provides multiple ways to write and incorporate custom extensions.

The most straightforward way to extend PyTorch is to create a custom module as a subclass of `torch.nn.Module`. This allows for the specification of custom trainable parameters and mathematical operations, and hooks directly into the autograd engine, which eliminates the need to write code for the backward training pass. However, the autograd engine is not able to handle functions such as straight-through estimators. In these cases, it is necessary to extend autograd itself through a custom `torch.autograd.Function`, in which both the forward and backward training passes must be explicitly implemented.

Custom extensions to PyTorch can be bottlenecked by the overhead of the Python interpreter. To resolve this, PyTorch allows functions to be written in C++ using the LibTorch library and hooked into a Python wrapper using the Ninja build

system with `torch.utils.cpp_extension`. Additionally, LibTorch can be interfaced with handwritten CUDA kernels, which enables optimizations such as kernel fusion. I used these approaches for the most performance-critical pieces of my training code.

3.1.2 Converting Models for Inference

The end result of the training process is a Python object representing the complete, trained model. The next step is to convert this object into equivalent SystemVerilog or C++ source code for inference on an edge platform. One approach would be to convert the entire model into a header file representing its parameters and hyperparameters, and then write the code for inference in a way which used this file for initialization (for instance, through heavy use of SystemVerilog `generate for` loops). This is difficult, however, since functional units which vary in behavior based on the model would have to be carefully parameterized within the constraints of the target language's preprocessor. Therefore, I chose to instead implement *templated* source files, which were specialized to produce inference source code for specific models. I created these templates using the Mako [24] Python library.

Mako enables the embedding of Python snippets into code written in arbitrary languages, which are then used to produce source code in the target language as a "pre-preprocessing" step. This style of metaprogramming is particularly useful for my work since my models are already represented as Python objects, and can therefore be directly manipulated by Mako.

3.2 Model Deployment and Evaluation Methodology

When deploying a machine learning model on the edge, there are effectively two options: design a custom accelerator, or write software for a commercial off-the-shelf platform. For most of my analysis, I used the former approach, since hardware accelerators are generally much faster and more efficient than software solutions. I focused primarily on FPGA-based solutions to aid in comparison with prior work.

However, using a custom accelerator in a commercial product is expensive. FPGAs have a high unit cost, which makes them undesirable for mass-produced devices, while ASICs have very high non-recurring engineering costs due to the expense of producing a custom set of masks. Therefore, I also performed some experiments using a low-cost commodity microcontroller.

3.2.1 Area, Power, and Performance Evaluation on FPGAs

I targeted several Xilinx FPGAs with my experiments to aid in comparison against different prior works. For BTHOWeN and ULEEN, I used Xilinx Vivado 2019.2 for all experiments, while for DWN I used Vivado 2022.2. After synthesis and implementation, I used Vivado's included power model and utilization reports to gather the dynamic power, static power, and hardware area of designs. In a few cases, prior works that I compared against did not publish power results. For these models, I used Xilinx Power Estimator [160], a spreadsheet-based analytical modeling tool, to estimate device power.

To check the functional correctness of designs, and to measure latency and throughput, I simulated several testbenches using Synopsys VCS and analyzed waveforms with DVE. In addition to simple unit-level tests, I also created a testbench which could run inference on a simulated accelerator using a complete dataset. I used a Python script to convert test data into a binary format, which was then loaded into the testbench using a `$fread` directive. By comparing the predictions of the software model to the simulated accelerator over thousands of samples, I could attain a high degree of confidence that the design was functionally correct.

3.2.2 Performance Evaluation on Microcontrollers

I also performed some experiments using an Elegoo Nano, which is a direct clone of the open-source Arduino Nano, a low-cost commodity microcontroller.[1] The Nano is based on the ATMega328P, which at time of writing retails for $1.52 in volume. By contrast, even low-end FPGAs can cost hundreds of dollars.

The ATMega is an 8-bit in-order processor for the AVR RISC ISA with a two-stage pipeline and no instruction or data caches. It provides 2 KB of SRAM, 30 KB of Flash memory, and runs (in the Nano) at a frequency of 16 MHz. Obviously, the throughput and energy efficiency of this device can not come anywhere close to a custom hardware accelerator. However, models such as DNNs and BNNs are generally impractical on such a limited device. Therefore, my goal with these experiments was to demonstrate the viability of WNNs in ultra-low-cost deployment scenarios.

All models which I ran on the Nano (both my work and comparison models) communicated with a host PC over a 1 Mbps serial connection in order to read in samples and write back predictions. This is an atypical baud rate for a serial connection. I chose this specific value based on the ATMega's datasheet, as it is the highest feasible value which is an exact divisor of the Nano's clock frequency. In theory, the Nano supports 2 Mbps transfer using "double speed USART" mode, but I found that this was unstable and resulted in frequent bit errors.

3.3 Evaluation Metrics

I compared the efficiency of my models against prior work in both software and hardware implementations. When examining the efficiency of the models themselves, I focused on the accuracy and parameter sizes of designs. When comparing hardware efficiency, I considered model throughput, latency, area, and energy. Directly com-

[1] This should not be confused with the similarly-named Nano 33 BLE, which has a much more powerful processor and far more memory.

paring the areas of designs on FPGAs is difficult since different functional units (e.g., LUTs, flip-flops, DSP slices, and block RAMs) have different sizes. Unfortunately, Xilinx does not provide the relative component areas which would be necessary to derive a normalized total circuit area. Therefore, I used LUT count as a proxy for circuit area. This metric is generally favorable to prior work, since the accelerator architectures I designed for WNNs do not use DSPs or BRAMs.

One factor I did *not* consider in my evaluations was the training time of models. This information is not available for many prior works, and was regardless not the focus of my experiments. The memory access patterns of WNNs are structured in a way that is not cache-friendly for CPUs or GPUs, so they were usually memory-bound during training. I was also generally willing to explore optimizations that increased the training time of models if they also increased the accuracy or inference-time efficiency of the final result. Regardless, the largest WNN models I discuss in this dissertation can be trained in less than a day on a single NVIDIA A100 GPU.

3.4 List of Datasets

Table 3.1 provides a listing of and citations for the datasets I used in this dissertation. I give additional brief notes where relevant.

Dataset Name	Source	Notes
MNIST	[92]	Classic image classification dataset
FashionMNIST	[158]	Same size as MNIST but more difficult.
KWS	[153]	Subset of dataset; part of MLPerf Tiny [23].
ToyADMOS/car	[84]	Subset of dataset; part of MLPerf Tiny [23].
VWW	[98]	Subset of dataset; part of MLPerf Tiny [23].
CIFAR10	[87]	Part of MLPerf Tiny [23].
Ecoli	[110]	Assorted tabular datasets of varying sizes and complexities
Iris	[15, 56]	
Letter	[132]	
Satimage	[134]	
Shuttle	[111]	
Vehicle	[107]	
Vowel	[47]	
Wine	[5]	
Phoneme	[14]	
Skin-seg	[26]	
Higgs	[155]	
UNSW-NB15	[106]	Network intrusion detection dataset.
BoT-IoT	[86]	Network intrusion detection dataset.
JSC	[50]	Particle physics; same paper proposed `hls4ml` platform
Madeline	[1]	
Fraud Detection	[147]	

Table 3.1: List of all datasets used in this dissertation.

Chapter 4: Improved Compression and Encoding for Weightless Neural Networks[1]

The proliferation of IoT devices and embedded sensors, combined with the desire for private, scalable, and low-latency machine learning, motivates a need for efficient edge inference. WNN models such as WiSARD in theory seem well-suited to this domain. Table lookups are computationally simple and energy-efficient operations, and their nonlinearity enables WNNs to represent complex behaviors with shallow models, reducing inference latency. However, two major issues that have so far prevented the widespread adoption of WNNs are their large parameter sizes, which limit their usefulness on memory-constrained devices, and their poor accuracy compared to state-of-the-art approaches. Therefore, edge inference today frequently leverages DNNs optimized using techniques such as those discussed in §2.2.1 [156].

Prior techniques to enhance and optimize WiSARD (§2.1.3) suggest that the gaps in accuracy and parameter size are not intractable. In particular, Bloom WiSARD [130] was frequently able to compress WiSARD models by three to four orders of magnitude without substantially harming their accuracy. However, some of these techniques have incompatibilities that make them challenging to integrate into a single model. In addition, prior evaluations of WNNs in edge contexts have generally been limited in scope. Therefore, an approach which combined these enhancements into a single model, along with an edge hardware platform that could be easily adapted to different applications, would be helpful to better understand the potential of WNNs.

This chapter describes BTHOWeN (**B**leached **T**hermometer-encoded **H**ashed-

[1]Zachary Susskind, Aman Arora, Igor D. S. Miranda, Luis A. Q. Villon, Rafael F. Katopodis, Leandro S. de Araújo, Diego L. C. Dutra, Priscila M. V. Lima, Felipe M. G. França, Mauricio Breternitz, and Lizy K. John. Weightless neural networks for efficient edge inference. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 279–290, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450398688. doi: 10.1145/3559009.3569680. URL `https://doi.org/10.1145/3559009.3569680`

input **O**ptimized **We**ightless Neural **N**etwork; pronounced as *Beethoven*), my initial exploration into improving WNNs in edge contexts. BTHOWeN consists of two components: a weightless model which incorporates multiple prior approaches to enhancing the WiSARD classifier and introduces further novel improvements, and an FPGA-based accelerator architecture for fast, energy-efficient inference. Specifically, with BTHOWeN, I make the following contributions:

1. A new model design for small and accurate WNNs. BTHOWeN incorporates bleaching [35], thermometer encodings [80], and Bloom filter-based model compression [130]. I demonstrate that a counter-based variant of the Bloom filter can be used to enable bleaching in compressed models, and furthermore that the hashing procedure in Bloom filters can be modified to improve hardware implementation efficiency. I introduce a generalized Gaussian thermometer encoding strategy which improves model accuracy independent of the underlying data distribution.

2. A comparison of this model against Bloom WiSARD, the prior state-of-the-art for memory-efficient WNNs, across nine multi-class classification datasets. BTHOWeN is shown to provide a geometric average 62% reduction in test error and 56% reduction in model parameter size.

3. An FPGA-based inference accelerator architecture for BTHOWeN models. Compared against quantized MLPs of similar accuracy, BTHOWeN achieves a mean 91% reduction in latency and 82% reduction in dynamic energy. Compared against tiny CNNs with similar accuracy, BTHOWeN's energy and latency reductions exceed 99%.

4. A toolchain for generating BTHOWeN models, including automated hyperparameter sweeping and bleaching value selection. A second toolchain for converting trained BTHOWeN models to RTL for the accelerator architecture. The code for this is publicly available at `https://github.com/ZSusskind/BTHOWeN`.

4.1 The BTHOWeN Model

My objective in designing BTHOWeN was to create a hardware-aware, high-accuracy, high-throughput WNN architecture. To accomplish this goal, I enhanced the techniques previously used to improve WiSARD with novel algorithmic and architectural improvements.

4.1.1 Efficient, Hardware-Friendly Hashing

Bloom filters require multiple independent hash functions, but do not constrain what these hash functions must be. Bloom WiSARD [130] used a double-hashing technique based on the MurmurHash [20] algorithm. Double-hashing only requires the evaluation of two hash functions, regardless of the number of functions needed by the Bloom filter. It accomplishes this by generating a set of independent values from the two hashed results. However, this technique introduces a multiplication and an addition for each hash function. Multiplication is a relatively costly operation in FPGAs, especially when many computations must be performed in parallel. Therefore, BTHOWeN does not use double-hashing, but rather evaluates all hash functions separately.

The MurmurHash algorithm itself introduces many arithmetic operations (e.g., 5 multiplications to hash a 32-bit value), which is similarly undesirable. Therefore, I use hash functions drawn from a strongly universal hash family, which are mutually independent and therefore minimize collision probabilities. I first considered using the Multiply-Shift family [49] of hash functions. For an n-bit input size and an m-bit output size, these implement the function $h(x) = (ax + b) \gg (n - m)$, where a is an odd n-bit integer, and b is an $(n - m)$-bit integer. Multiply-shift functions can be implemented using only a few machine instructions, so they are inexpensive in software, which is beneficial for fast training. Unfortunately, the overhead from the single multiplication operation was still excessive in hardware.

Instead, BTHOWeN uses the H3 family of hash functions [34]. As shown in

Figure 4.1, functions in H3 require no arithmetic operations, only XOR operations and multi-bit 2:1 MUXes (optimizable to AND gates). For an n-bit input x and m-bit output, hash functions in the H3 family take the form:

$$h(x) = x[0]p_0 \oplus x[1]p_1 \oplus \ldots \oplus x[n-1]p_{n-1}$$

Here, $x[i]$ is the ith bit of x, and $P = \{p_0 \ldots p_{n-1}\}$ consists of n random m-bit values. The drawback of the H3 family is that its functions require substantially more storage for parameters when compared to the Multiply-Shift family: nm bits versus just $2n - m$.

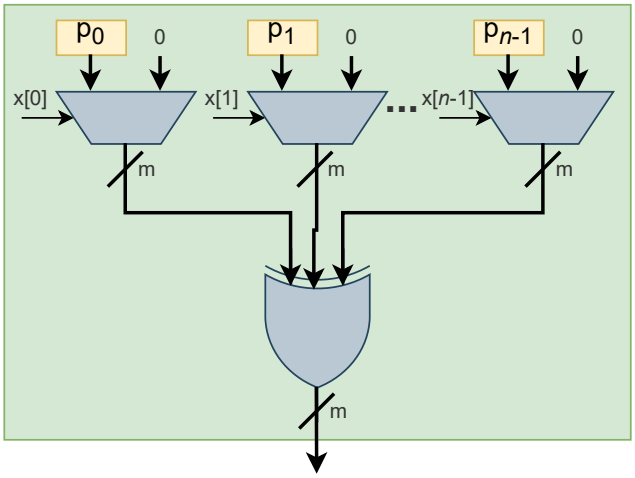


Figure 4.1: Hardware implementation of an H3 hash function. These arithmetic-free functions are fast and energy-efficient, but require a large set of random parameters (shown here as $\{p_0, \ldots, p_{n-1}\}$).

Since a Bloom WiSARD model is composed of many Bloom filters (one per RAM node), and Bloom filters themselves require multiple hash functions, the storage requirements for the H3 hash parameters become problematic if implemented naively. Additionally, since many parameters need to be accessed simultaneously in order for hash computations to be performed in parallel, it is difficult to map these parameter sets to FPGA block RAMs, meaning less area-efficient solutions such as large register files must be used instead. However, there is no disadvantage to sharing the same

set of hash functions between all Bloom filters in a BTHOWeN design. Therefore, all Bloom filters can retrieve their hash parameters from a single central register file, with one nm-bit entry for each of the shared hash functions.

This reuse of hash parameters enables an additional optimization. Recall from §2.1.2 that WiSARD models typically use the same assignment of inputs to tuples for all discriminators. This means that Bloom-filter-based RAM nodes at the same index in different discriminators will receive the same inputs, and therefore calculate the same hashed indices into their internal LUTs. By separating the hashing component of the Bloom filter from the table lookup component, BTHOWeN calculates hash values only once for all discriminators, which reduces computations by a factor equal to the number of classes.

Note that cryptographically-secure hash functions such as SHA and MD5 are a poor choice for hardware-friendly Bloom filters, as their security features introduce substantial computational overhead and are unnecessary in this context.

4.1.2 Counting Bloom Filters

Bloom filters are data structures for approximate set membership: given an input, they indicate whether the input is *definitely not* an element of the set represented by the filter or *possibly* an element of the stored set. On the other hand, bleaching requires a record of *how many times* each input pattern was seen during training. Therefore, there at first appears to be a fundamental incompatibility between bleaching and Bloom WiSARD.

To resolve this issue, I introduce the use of *counting* Bloom filters for training BTHOWeN. As depicted in Figure 4.2, counting Bloom filters replace the binary entries of a conventional Bloom filter with a set of multi-bit counters. When training a BTHOWeN model, each time an input is presented to a filter, counter values are read from the locations indicated by the hashed results. The counter with the smallest value (or counter**s**, in the event of a tie) is then incremented. At inference time, the

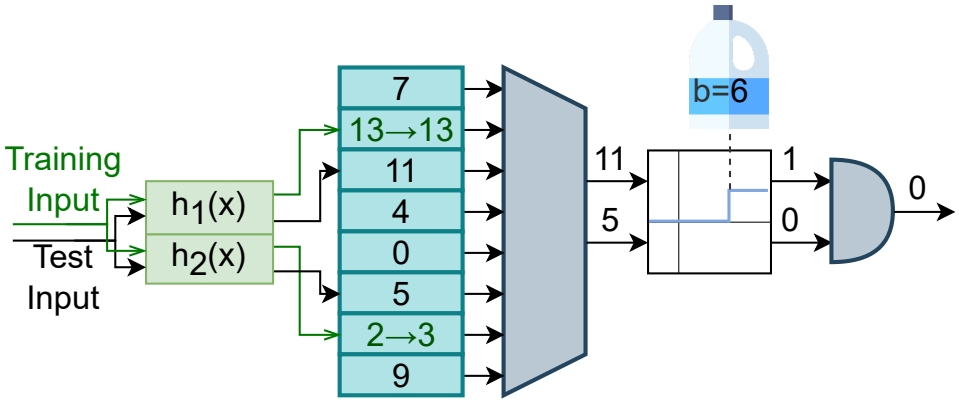


Figure 4.2: Implementation of a counting Bloom filter used in training BTHOWeN. During training, when an input is presented to the filter, the smallest of the counter values accessed by the hash function outputs is incremented. During inference, the smallest accessed counter value is compared against the bleaching threshold to determine the filter response.

smallest of the accessed counter values is compared against the bleaching threshold b to determine the filter response. As with conventional Bloom filters, false negatives are impossible: if a pattern was seen i times during training, then the smallest of its associated counter values must be at least i. Therefore, the possible responses of a counting Bloom filter with threshold b are *possibly seen at least b times* and *definitely not seen b times*.

The construction of the counting Bloom filter in BTHOWeN is different from the typical form. Classically [82], counting Bloom filters increment *all* indicated counter values when an element is added. The advantage of this approach is that it allows for elements to be removed by decrementing counters, though doing so introduces the possibility of false negatives. However, BTHOWeN does not need the ability to remove elements from a filter, and incrementing only the smallest counter provides a tighter upper bound on how many times patterns were seen. The implementation of counting Bloom filters in BTHOWeN is conceptually similar to count-min sketches under the conservative update rule [25]. However, count-min sketches use a separate data array for each hash function, while counting Bloom filters use a unified data

array. This results in a tradeoff between false positive rate and memory footprint.

4.1.3 General Nonlinear Thermometer Encoding

Most prior works which used multi-bit unary thermometer encodings for WNNs used equal increments between thresholds. The thresholds for an input with minimum value m and maximum value M using a k-bit encoding are given by Equation 4.1:

$$\left\{ m + i\frac{M-m}{k+1} \,\middle|\, i \in \{1 \dots k\} \right\} \tag{4.1}$$

This "linear" approach works well if an input feature is uniformly distributed, and therefore equal importance should be given to all parts of the input space. However, it works less well for features which are more likely to assume values closer to their mean. For instance, Figure 4.3 shows a 7-bit encoding for a feature that follows a truncated normal distribution. If a linear encoding is used to represent this feature, then 4/7 bits (57%) are used to encode just 24% of inputs. Recognizing this issue, one prior work [157] explored using unequal increments between thresholds when features followed a known, non-uniform distribution, and found that this significantly improved accuracy.

In BTHOWeN, I use a nonlinear thermometer encoding that assumes that the distributions of input features are approximately Gaussian. This approach increases the resolution of the encoding near the mean value of an input feature, while sacrificing some fidelity near the extrema. For an input feature with mean μ and standard deviation σ, the thresholds for a k-bit encoding are derived from the quantile function of the normal distribution [133], as shown in Equation 4.2.

$$\left\{ \sqrt{2}\sigma \cdot \mathrm{erf}^{-1}\left(\frac{2i}{k+1} - 1\right) + \mu \,\middle|\, i \in \{1 \dots k\} \right\} \tag{4.2}$$

As Figure 4.3 demonstrates, this approach is beneficial when the values of input features are concentrated at their means, as it avoids dedicating excessive en-

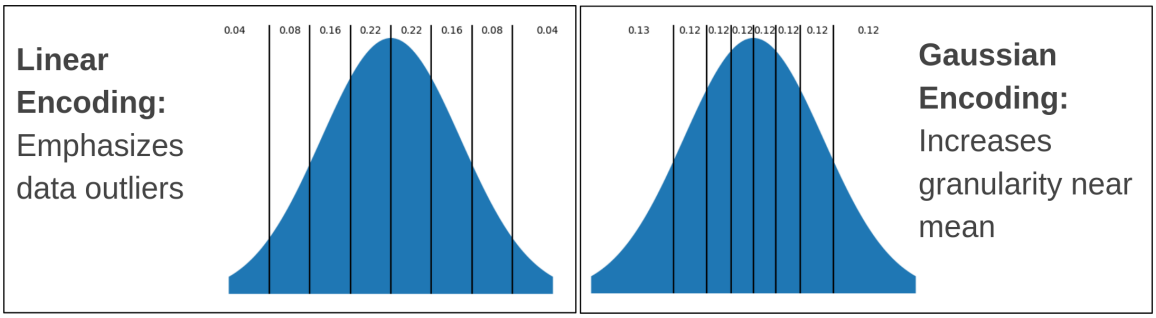


Figure 4.3: Comparison of the conventional "linear" thermometer encoding with the Gaussian strategy used in BTHOWeN. Gaussian encoding avoids assigning disproportionately many encoding bits to represent data outliers when the distribution of an input is concentrated at its mean.

coding bits to representing outlying values. BTHOWeN uses this strategy for all input features, without requiring any information about the actual underlying distribution.

4.2 BTHOWeN Software Model

Figure 4.4 summarizes the process of creating a BTHOWeN model. The model is initialized based on dataset parameters (inputs and classes) and configurable hyperparameters (encoding bits per model input and inputs, table entries, and hash functions per Bloom filter). Tuple mappings and H3 hash function parameters are selected at random, and all counters are set to 0.

During training, samples are sequentially presented to the discriminators corresponding to their output labels. Inputs are encoded, mapped to tuples, and passed to the counting Bloom filters, which update their entries using the approach described in §4.1.2. As with WiSARD, this is a single-pass process, since additional training passes would effectively just multiply counter values by a constant.

After training, the bleaching threshold b is chosen to maximize the model's ability to generalize to new data. This is typically evaluated using a small subset of the dataset which was not used during training. For very small datasets, this process may be too noisy due to the tiny size of the bleaching/validation set, in which case

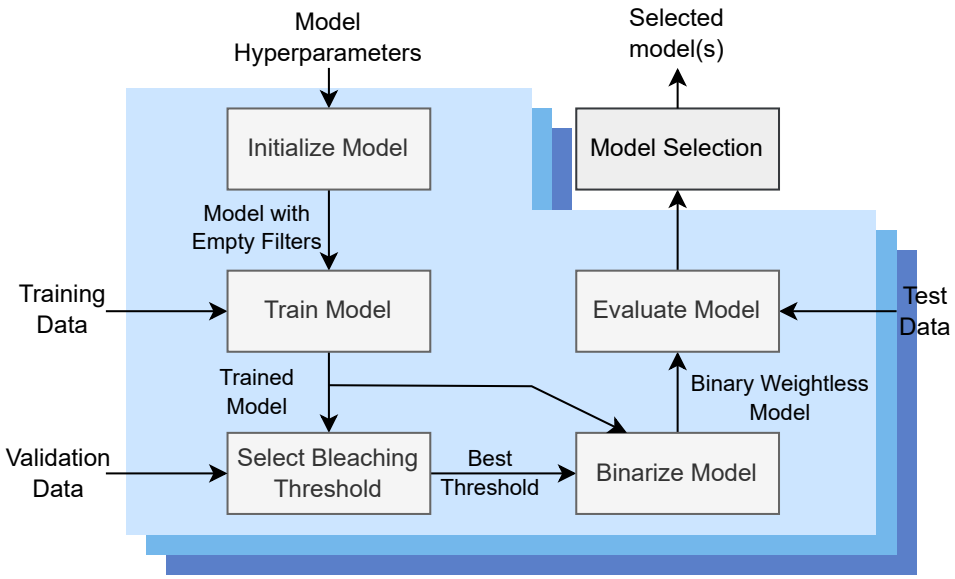


Figure 4.4: A flowchart depicting the process of training BTHOWeN models and preparing them for inference. After model initialization, BTHOWeN is trained using a single-pass learning rule. A validation set is used to select a bleaching threshold, and Bloom filters are binarized based on this threshold. Hyperparameter sweeping is used to identify models which strike a good balance between parameter size and accuracy.

the training data is used as well. The selection of b is performed using a binary search strategy between 1 and the largest value seen in any counter, detailed in Algorithm 1. Once bleaching has been performed, the model can be binarized by replacing counter values less than b with 0 and counter values of at least b with 1. This allows the counting Bloom filters to be replaced with conventional (binary) Bloom filters, which reduces the memory footprint of the model for inference.

BTHOWeN models have several configurable hyperparameters. Increasing the size of the Bloom filters' lookup tables decreases the frequency of false positives, which can improve accuracy, but increases the model's parameter size and provides diminishing returns. Using more inputs per Bloom filter broadens the space of Boolean functions the filters can learn to approximate, and makes the model size smaller as fewer Bloom filters are needed in total. However, it also increases the likelihood of

false positives, and can lead to overfitting. Using more hash functions per Bloom filter has complex impacts on false positive rates [63] and therefore accuracy, though I have empirically found that there is almost never a benefit to using more than four hash functions for BTHOWeN, particularly since hash computation increases the cost of inference. Lastly, increasing the number of bits in the thermometer encoding improves input resolution and therefore accuracy at the cost of model size.

Algorithm 1 BTHOWeN bleaching threshold selection

Input: Trained model M; bleaching dataset (X, Y)
Output: Bleaching threshold b

$max\_val \leftarrow \max_{d \in M.discriminators} (\max_{f \in d.filters} (\max_{c \in f.counters} (c.value)))$
$b \leftarrow \lfloor max\_val/2 \rfloor$
$step \leftarrow \max(\lfloor max\_val/4 \rfloor, 1)$
$known \leftarrow \text{Dict}(\{\})$
while True **do**
    $values \leftarrow [b - step, b, b + step]$
    $accuracies \leftarrow [0, 0, 0]$
    for $i \in \text{range}(3)$ **do**
        if $values[i] \leq 0$ **then**
            continue ▷ Value out of range.
        end if
        if $values[i] \in known$ **then**
            $accuracies[i] \leftarrow known[values[i]]$
            continue ▷ Use cached result.
        end if
        $predicted \leftarrow M(X, values[i])$ ▷ Run inference with trial bleaching value.
        $accuracies[i] \leftarrow \text{sum}(predicted == Y)$ ▷ Count correct inferences.
        $known[values[i]] \leftarrow accuracies[i]$
    end for
    $best \leftarrow values[\text{argmax}(accuracies)]$
    if ($best$ is b) and ($step$ is 1) **then**
        return b
    end if
    $b \leftarrow best$
    $step \leftarrow \max(\lfloor step/2 \rfloor, 1)$
end while

In order to identify Pareto-optimal combinations of model hyperparameters,

I developed an automated sweeping methodology for BTHOWeN. The single-pass learning rule for BTHOWeN makes it efficient to train, generally running in a few minutes on a single CPU core, which allowed me to explore a large search space. An example hyperparameter sweep configuration is shown in Table 4.1. All 1,008 combinations of these hyperparameters were used to train separate models.

Hyperparameter	**Values**
Encoding Bits per Input	1, 2, 3, 4, 5, 6, 7, 8
Input Bits per Bloom Filter	28, 49, 56
Entries per Bloom Filter	128, 256, 512, 1024, 2048, 4096, 8192
Hash Functions per Bloom Filter	1, 2, 3, 4, 5, 6

Table 4.1: Example hyperparameter sweep values for BTHOWeN. These specific values were used for the MNIST dataset. Other datasets had minor variations due to differences in input dimensionality.

4.3 BTHOWeN Inference Accelerator

Figure 4.5 shows a block diagram for the FPGA-based inference accelerator I developed for BTHOWeN. To simplify control logic, functional units in the accelerator operate in lockstep to the greatest extent possible. This means that an entire input sample must be read in before computation can begin. This deserialization is performed using a double-buffered bus interface (not shown). Bloom filters are divided into separate "Bloom Hasher" and "Bloom Lookup" blocks, which avoids the redundant computation of hashes for each discriminator. Each hasher computes H3 hash functions of mapped model inputs, reading the hash parameters, which are shared between all Bloom filters, from a single shared register file. Since the off-chip bus usually has insufficient bandwidth to load a full sample each cycle, hashers are time-multiplexed to reduce model area. Hashers evaluate a single hash function at a time for all the Bloom filters in the model, storing partial results in an intermediate buffer. Once a computation has been finished for all filters, the results are passed

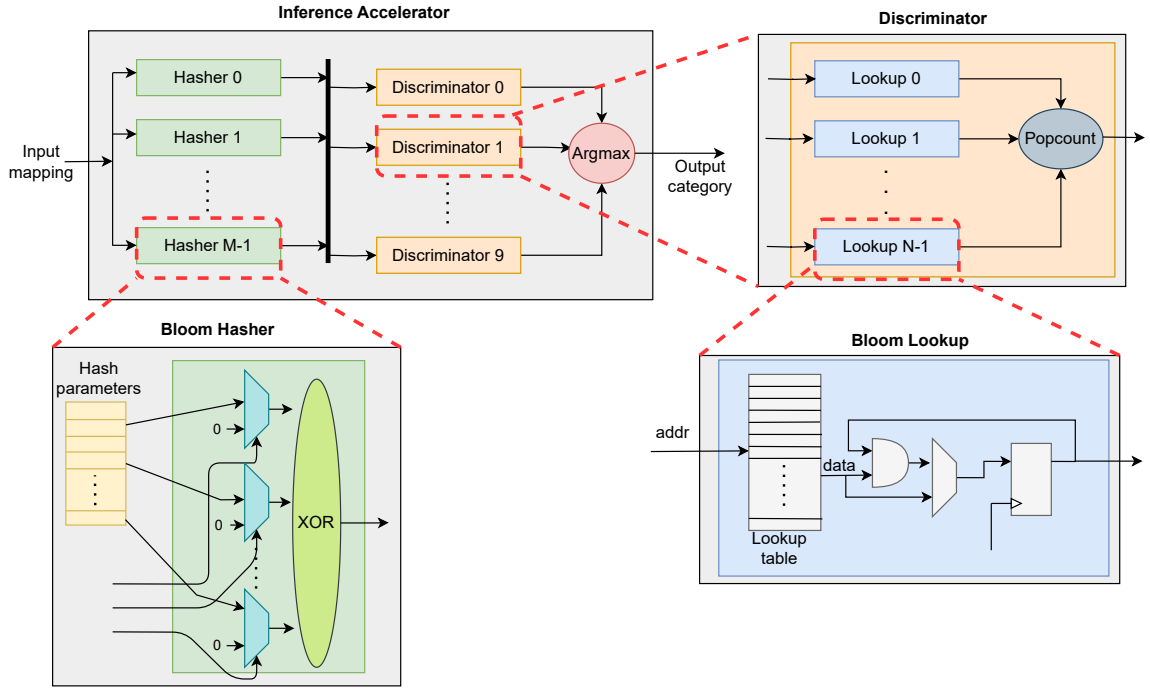


Figure 4.5: Inference accelerator architecture for BTHOWeN. Bloom filters are divided into dedicated Hasher and Lookup blocks. The Hasher blocks compute the H3 hash function on the input data, reading from a shared set of random hash parameters. The Discriminator block takes hashed data as input, passes it through Lookup blocks, and performs a popcount on the result, returning a response. The Lookup block contains a LUT, which is accessed using the addresses produced by the hashers, and performs an AND reduction on the results of multiple accesses.

to the Bloom lookup blocks, and the hashers move on to evaluating the next hash function.

The lookup blocks accept the hash results as inputs. They use these results to index their internal LUT, and either store this value in an internal register or compute the AND of it with the current value of the register. This ultimately computes a serial AND reduction across all hash function results. A small internal state machine ensures that the correct number of hashed inputs are combined and that all lookup blocks output their final values in lockstep.

The popcount modules counts the number of 1s in the outputs of the lookup

blocks within each discriminator, producing the response scores. Lastly, the argmax module determines the index of the discriminator with the strongest response. These modules are unchanged from the conventional WiSARD model.

Training with bleaching requires multi-bit counters for each entry in each Bloom filter, which introduces a large memory overhead. For instance, some models have optimal bleaching values greater than 400. If the accelerator included saturating counters large enough to represent this value, it would increase the memory usage of the design by a factor of 9. There would also be additional hardware complexity introduced by the counter increment mechanisms and comparators. Since BTHOWeN is intended for inference on edge devices, I determined that the costs of supporting on-chip training in this accelerator outweighed the advantages.

4.4 Evaluation Methodology

I evaluate BTHOWeN on the same nine multi-class classification datasets that were used for Bloom WiSARD [130]. Most of these datasets are tabular in nature, meaning features do not carry positional information. I compare BTHOWeN against quantized, fully-connected (multilayer perceptron / MLP) DNNs on these datasets.

A neural architecture search was used to identify models that were as small as possible while still having accuracy comparable to BTHOWeN. The trained models were then quantized to 8-bit precision, and hardware was generated for them using the `hls4ml` tool [50]. The training and data collection for these MLP models was performed by my co-authors in the main paper [140], particularly A. Arora, L. Villon, and R. Katopodis.

All MLP models and most BTHOWeN models target the Zybo Z7 development board,[2] which was used for a prior FPGA-based WNN accelerator [53]. For MNIST, which was the largest evaluated dataset in terms of number of input features, I also

[2]Part number `xc7z020clg400-1`

created a larger design which targeted a Kintex UltraScale FPGA[3] in order to explore the scalability of BTHOWeN. All models were implemented at a clock frequency of 100 MHz. Most models used a data bus width of 64b; for the BTHOWeN model on the larger FPGA, I also explored an implementation with a 256b bus due to the larger number of available I/O pins.

Since MNIST is an image dataset, it can also be approached using convolutional neural networks. CNNs perform very well for large image datasets, but I was interested in exploring their scalability to tiny devices. However, `hls4ml` struggled to scale down to the small Zybo FPGA and produced inefficient designs for the convolutional layers (this issue did not impact the MLPs, which only had fully-connected layers). To make a more fair comparison with tiny CNNs, I used the latency and resource utilization values for optimized implementations reported by Arish et. al. [127]. Xilinx Power Estimator (XPE) [160] was then used to estimate energy for this design.

4.5 Results

4.5.1 Selected BTHOWeN Models

After hyperparameter sweeping, I picked models for each dataset which struck a good balance between model parameter size and accuracy. Sweeping results showed clear points of diminishing returns as model sizes grew larger. For most datasets, I only chose a single model for inference. However, for MNIST, I picked three models at different design points in order to better illustrate the tradeoff between model size and accuracy. Hyperparameters for the selected models, along with the iso-accuracy MLPs used for comparison, are shown in Table 4.2.

[3]Part number `xcku035-ffva1156-1-c`

Model Name	Bits /Input	Bloom Filter Inputs	Bloom Filter Entries	Bloom Filter Hashes	Size (KiB)	Test Acc.	MLP Layers
MNIST-S	2	28	1024	2	70.0	0.934	—
MNIST-M	3	28	2048	2	210	0.943	784–16–10
MNIST-L	6	49	8192	4	960	0.952	—
Ecoli	10	10	128	2	0.875	0.875	7–8–8
Iris	3	2	128	1	0.281	0.980	4–4–3
Letter	15	20	2048	4	78.0	0.900	16–40–26
Satimage	8	12	512	4	9.00	0.880	36–16–16–6
Shuttle	9	27	1024	2	2.63	0.999	9–4–7
Vehicle	16	16	256	3	2.25	0.762	18–16–4
Vowel	15	15	256	4	3.44	0.900	10–18–11
Wine	9	13	128	3	0.422	0.983	13–10–3

Table 4.2: Hyperparameters for selected BTHOWeN and MLP models.

4.5.2 Comparison with Iso-Accuracy DNN Models

Table 4.3 shows FPGA implementation results for the selected BTHOWeN models, iso-accuracy DNNs, tiny CNNs, and a prior WNN FPGA implementation. All models were implemented on the Zybo Z7 FPGA except for BTHOWeN-L, which was implemented on the larger Kintex board. "BTHOWEN-L*" additionally increased the data bus width from the 64b used for all other models to 256b in order to reduce the data movement bottleneck associated with reading samples from off-chip. The winning results between BTHOWeN and the MLP models are highlighted for power, energy, and area on each dataset. In all cases, the difference in accuracy between the two models is no more than 0.4%.

For the MNIST dataset, the medium BTHOWeN model is only 0.3% less accurate than the MLP, reduces dynamic energy by 42%, and reduces latency by almost 96%. The MLP uses fewer LUTs and FFs than the medium BTHOWeN

Dataset	Model	Cycles /Inference	Dyn. Energy (nJ/Inf.)	LUTs	FFs	BRAMs (36Kb)	DSPs	Test Acc.
MNIST	BTHOWeN-S	25	48.75	15,756	3,522	0	0	0.934
	BTHOWeN-M	*37*	*142.8*	38,912	6,577	*0*	*0*	0.943
	BTHOWeN-L	74	2,225	151,704	18,796	0	0	0.952
	BTHOWeN-L*	19	600.0	158,367	25,905	0	0	0.952
	MLP	846	245	*2,163*	*3,007*	8	28	0.946
	CNN 1	33,615	19,497	5,753	3,115	7	18	0.947
	CNN 2	33,555	14,429	3,718	2,208	5	10	0.920
	Hashed WNN	28	118.4	9,636	4,568	128.5	5	0.907
Ecoli	BTHOWeN	*2*	*0.24*	*353*	*223*	0	0	0.875
	MLP	14	4.2	1,596	1,615	0	0	0.875
Iris	BTHOWeN	*1*	*0.05*	*57*	*90*	0	0	0.980
	MLP	10	0.8	427	488	0	0	0.980
Letter	BTHOWeN	*4*	*24.92*	21,603	*2,715*	0	0	0.900
	MLP	26	39.52	*17,305*	15,738	0	0	0.904
Satimage	BTHOWeN	*5*	*4.2*	*3,771*	*1,131*	0	0	0.880
	MLP	25	9.75	7,007	7,558	0	0	0.878
Shuttle	BTHOWeN	*2*	*0.36*	*593*	*121*	0	0	0.999
	MLP	14	1.82	693	711	0	0	0.999
Vehicle	BTHOWeN	*5*	*1.9*	*1,781*	*597*	0	0	0.762
	MLP	15	3.6	2,824	3,035	0	0	0.766
Vowel	BTHOWeN	*2*	*0.8*	*1,559*	*756*	0	0	0.900
	MLP	18	12.6	5,743	4,663	0	0	0.903
Wine	BTHOWeN	*3*	*0.36*	*585*	*239*	0	0	0.983
	MLP	14	3.64	1,836	1,832	0	0	0.983

Table 4.3: FPGA implementation results for BTHOWeN and comparison models. "Hashed WNN" from [53]. Models are grouped by dataset, with the winning metrics italicized for each dataset (for MNIST, BTHOWeN-M is compared against the MLP model).

model, but also requires FPGA DSP and BRAM resources. BTHOWeN also compares favorably against the tiny CNNs. CNN-1 has an accuracy of 94.7%, just slightly better

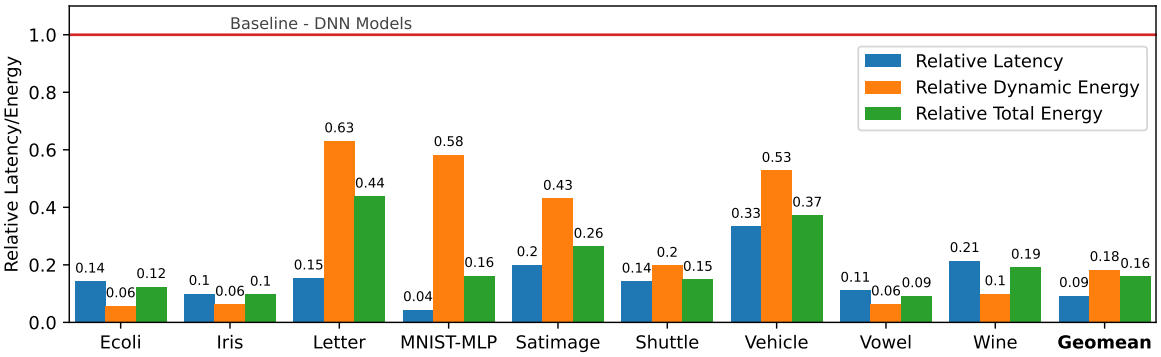


Figure 4.6: Relative latencies and energies of BTHOWeN models versus iso-accuracy MLPs. All models were implemented on a Zybo Z7 FPGA at 100 MHz, with power and performance metrics derived from Vivado reports.

than BTHOWeN-M. Despite this, BTHOWeN consumes 0.74% of the energy of the CNN, while reducing latency from 33.6k cycles to 37.

As Table 4.3 illustrates, BTHOWeN's hardware implementation consumes fewer hardware resources (LUTs and FFs) than its MLP counterpart for all datasets except MNIST and Letter. BTHOWeN's reduction in dynamic energy ranges from 37% on Letter to 94% on Ecoli, Iris, and Vowel. Figure 4.6 summarizes the performance of BTHOWeN versus the MLP models. It also includes total (static + dynamic) energy values. BTHOWeN's advantage in total energy is often even larger than in dynamic energy because its higher throughput decreases leakage energy per inference. Overall, BTHOWeN models are significantly faster and more energy efficient than DNNs of comparable accuracy.

4.5.3 Comparison with Bloom WiSARD

Bloom WiSARD [130] is a state-of-the-art memory-efficient WNN model. However, it does not have an associated hardware implementation. Therefore, in comparing it with BTHOWeN, I focused on model parameter sizes and accuracies. Figure 4.7 displays the results of this comparison. In all cases, BTHOWeN achieves superior accuracy with a smaller parameter size than the prior work. On average, inference

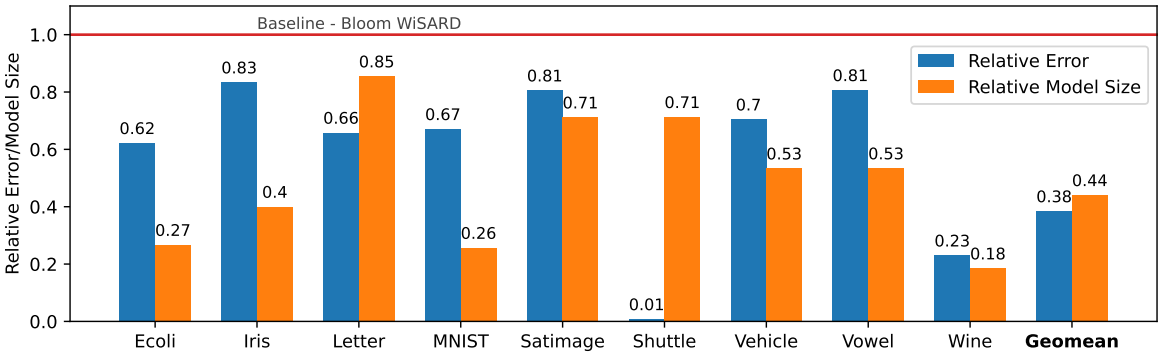


Figure 4.7: Relative model accuracies and parameter sizes for BTHOWeN versus Bloom WiSARD [130]. BTHOWeN outperforms the prior WNN on all nine datasets in both metrics. For MNIST, the BTHOWeN-M model was used for comparison.

error rate is reduced by 62% and parameter size is decreased by 56%. Although no hardware implementation of Bloom WiSARD exists, BTHOWeN would presumably also be more efficient there due to its use of a less expensive hash function.

The Shuttle dataset is a notable outlier in Figure 4.7, as BTHOWeN reduces error by ∼99% versus Bloom WiSARD. Shuttle is an anomaly-detection dataset in which 80% of the training data belongs to the "normal" class [111]. Since Bloom WiSARD does not incorporate bleaching, the discriminator corresponding to this class likely became saturated during training, causing it to always output a near-maximal response during inference.

4.5.4 Comparison with Prior FPGA-based WNN

A prior WNN accelerator [53] for MNIST was implemented on the same Zybo FPGA used for most results in Table 4.3, and at the same frequency of 100 MHz. This prior work is shown in the row labeled "Hashed WNN". The latency and energy consumption for this design lie between the BTHOWeN-S and BTHOWeN-M models, but its accuracy is far worse than even BTHOWeN-S.

While the exact latency for this design was not published, it suffers from slow memory access speeds. The accelerator reads in one 28-bit filter input per cycle, and

uses a single-bit input encoding, so it takes 28 cycles to read in a 784-bit sample. Therefore, the 28 cycles given for this design in Table 4.3 is a lower bound. Energy is also a lower bound based on this cycle count and published power values.

BTHOWeN has significant architectural differences which contribute to its superior accuracy and efficiency:

1. The prior accelerator used a simple hash-table-based encoding scheme, which had explicit hardware for collision detection. BTHOWeN uses an approach based on Bloom filters, which is tolerant of occasional false positives and does not detect or explicitly mitigate them.

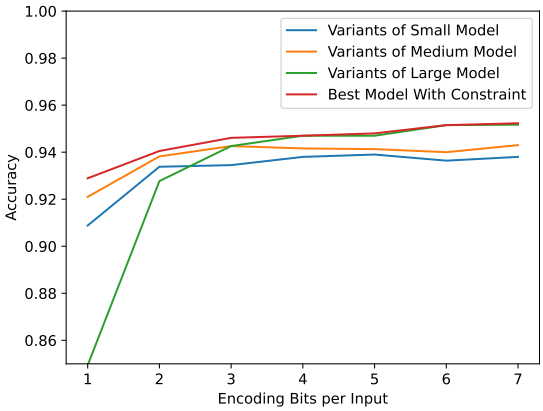
2. The prior accelerator did not support bleaching or thermometer encoding, constraining it to single-bit encodings and risking saturation issues. BTHOWeN uses counting Bloom filters and the multi-bit Gaussian thermometer encoding to avoid these limitations.

3. The prior accelerator supported on-chip training. While this is potentially useful for online learning applications, and did not require multi-bit counters in their design due to its lack of bleaching support, it introduces significant hardware complexity (evidenced in their use of large numbers of BRAMs), which harms the efficiency of inference.

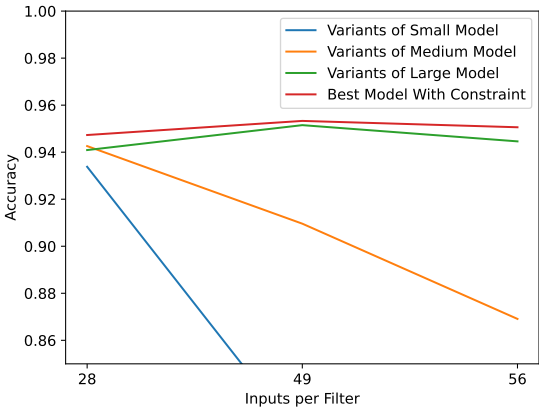
4.5.5 Model Sweeping Analysis

Figure 4.8 uses the hyperparameter sweep results for MNIST as an illustrative example of the tradeoffs and points of diminishing return that I observed in BTHOWeN model creation. The first four plots show results which vary a single hyperparameter from the selected BTHOWeN configurations in Table 4.2.

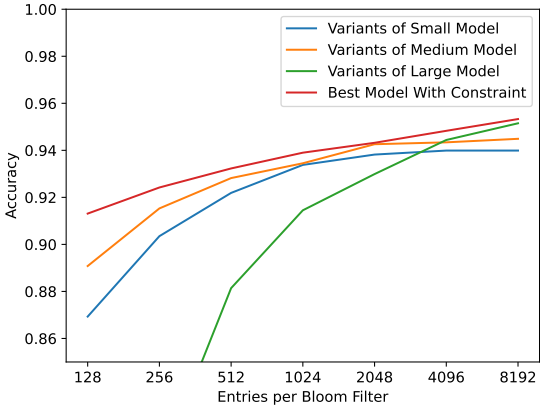
From Figure 4.8a, it is clear that using more thermometer encoding bits provides clear but diminishing improvements. Variants of the large model underperform the smaller models when restricted to very few encoding bits due to the use of a wider



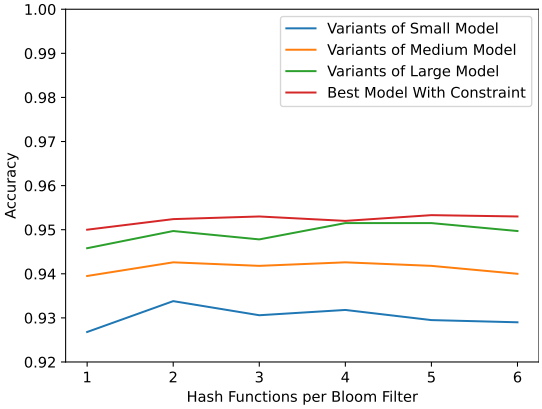
(a) Accuracy vs. thermometer bits



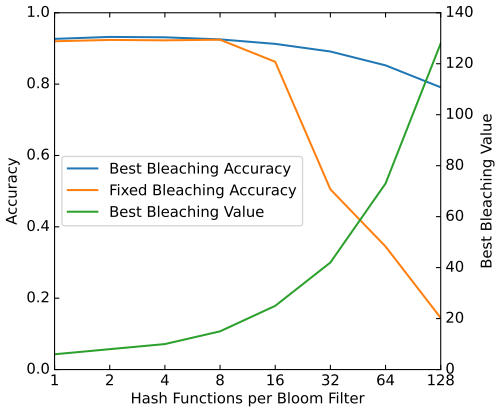
(b) Accuracy vs. Bloom filter inputs



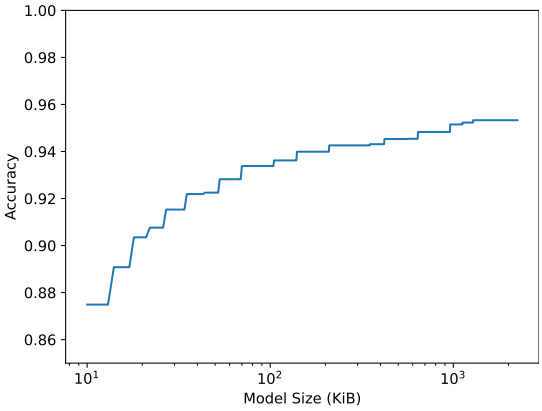
(c) Accuracy vs. Bloom filter entries



(d) Accuracy vs. Bloom filter hash fns.



(e) Accuracy vs. hash functions per filter with fixed (b=16) and variable bleaching



(f) Most accurate models not larger than given size constraint

Figure 4.8: Sweeping BTHOWeN across 1,008 configurations (Table 4.1) on MNIST.

Bloom filter input width (49 versus 28 bits), resulting in too few Bloom filters per discriminator. In 4.8b, increasing the number of inputs per filter from 28 to 49 hurts accuracy for the small and medium models, as their Bloom filters do not have enough entries to handle the larger variety of patterns that are seen. In 4.8c, increasing the number of entries per filter is seen to uniformly improve accuracy, though again with diminishing returns. Restricting the entries per filter for the large model below 4096 causes rapid collapse in accuracy, underscoring the importance of having a large LUT capacity when a filter has many inputs. Lastly, 4.8d shows the consequences of varying the number of hash functions per Bloom filter. While going from one hash function to two gives a small but clear improvement, the effects of more than two hash functions are less clear in most cases.

I expected that using progressively more hash functions per Bloom filter would eventually harm accuracy, since this is known to increase their false positive rate [63]. However, from Figure 4.8d, this does not appear to be happening. To understand this effect, I explored variants of the small MNIST model with up to 128 hash functions per Bloom filter, shown in Figure 4.8e. When the bleaching threshold is fixed at the optimal value for the original model (b=16), accuracy degrades with more hash functions per filter, particularly once more than 8 functions are used. However, when b is allowed to vary according to the binary search strategy in Algorithm 1, the impact on accuracy is much smaller, and instead b increases. From this, it appears that bleaching can help to compensate for a poor choice of the number of hash functions per filter, since it mitigates the high false positive rate that would otherwise occur.

Finally, Figure 4.8f shows the most accurate BTHOWeN model which could be obtained under a given maximum model size. Exponentially increasing the model size provides diminishing returns in accuracy. This suggests that further algorithmic improvements are needed to achieve higher accuracies with reasonable model sizes.

4.6 Summary

In this chapter, I presented BTHOWeN, my first weightless model and hardware architecture co-designed for efficient inference. BTHOWeN combines and enhances prior WNNs with additional algorithmic improvements, including counting Bloom filters, optimized hashing, and general Gaussian thermometer encoding. It outperforms Bloom WiSARD by 62% in inference error and 56% in parameter size, thereby establishing a new state-of-the-art for WNNs. BTHOWeN's FPGA accelerator platform enables fast, energy-efficient inference, reducing latency by 91%, dynamic energy by 82%, and total energy by 84% versus edge-optimized 8-bit quantized MLP models.

Chapter 5: Multi-Pass Learning with Weightless Ensembles[1][2][3]

Binary neural networks (BNNs) [42, 43, 71, 123] have received considerable interest as an approach to inference on the extreme edge. In particular, the Xilinx Brevitas [118] library and accompanying FINN [145] platform provide a toolchain for the entire process of developing BNNs, from training models to deploying them for inference on edge FPGAs. While BTHOWeN is considerably more efficient and accurate than prior WNNs and compares favorably against quantized DNNs implemented using `hls4ml`, it can fall short of FINN in accuracy, throughput, and efficiency. For instance, the largest BTHOWeN model for MNIST achieved 95.2% accuracy, required 158k LUTs, and could process one sample every 19 cycles. By contrast, the *smallest* published FINN model achieves 95.8% accuracy, requires 91k LUTs, and can process one sample every 16 cycles. BTHOWeN is unable to match the accuracy of the larger FINN models.

While BNNs are efficient in hardware, they require activations to be propagated through multiple layers of computation, increasing inference latency. Therefore,

[1] Zachary Susskind, Aman Arora, Igor D. S. Miranda, Alan T. L. Bacellar, Luis A. Q. Villon, Rafael F. Katopodis, Leandro S. de Araújo, Diego L. C. Dutra, Priscila M. V. Lima, Felipe M. G. França, Mauricio Breternitz Jr., and Lizy K. John. ULEEN: A novel architecture for ultra-low-energy edge neural networks. *ACM Trans. Archit. Code Optim.*, 20(4), dec 2023. ISSN 1544-3566. doi: 10.1145/3629522. URL `https://doi.org/10.1145/3629522`

[2] Zachary Susskind, Alan T.L. Bacellar, Aman Arora, Luis A.Q. Villon, Renan Mendanha, Leandro S. De Araújo, Diego L.C. Dutra, Priscila M.V. Lima, Felipe M.G. França, Igor D.S. Miranda, Mauricio Breternitz, and Lizy K. John. Pruning weightless neural networks. In *ESANN 2022 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 37–42, 2022. doi: http://dx.doi.org/10.14428/esann/2022.ES2022-55

[3] (Poster presentation) Zachary Susskind, Aman Arora, Alan T. L. Bacellar, Diego L. C. Dutra, Igor D. S. Miranda, Mauricio Breternitz, Priscila M. V. Lima, Felipe M. G. França, and Lizy K. John. An FPGA-based weightless neural network for edge network intrusion detection. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23, page 232, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394178. doi: 10.1145/3543622.3573140. URL `https://doi.org/10.1145/3543622.3573140`

there is still merit to exploring WNNs as an alternative to BNNs for edge inference. However, additional improvements are clearly needed on top of what I proposed in BTHOWeN to bridge the gap in model performance. Therefore, I introduce a set of novel WNN optimizations, including a multi-pass learning rule, additive submodel ensembles, and RAM node pruning, which expand the viability of WNNs in extreme edge inference scenarios. Collectively, these contribute to a weightless neural model and hardware architecture I call ULEEN (**U**ltra **L**ow **E**nergy **E**dge **N**etworks). In particular, I make the following contributions:

1. ULEEN, a novel weightless model that enhances prior WNNs such as BTHOWeN by introducing a multi-pass gradient-based learning rule, additive submodel ensembles, and RAM node pruning. I compare ULEEN to fully-connected binary neural networks on the four MLPerf Tiny datasets and MNIST. ULEEN reduces parameter size by a geometric average of 1.9× compared to comparably accurate BNNs trained using the Brevitas library, and increases accuracy by an average of 3.1% over similarly-sized BNNs.

2. A fast, energy-efficient FPGA-based inference accelerator architecture for ULEEN. I compare this accelerator against the Xilinx FINN [145] platform for optimized BNN deployment on a Xilinx Zynq Z-7045 FPGA. The FPGA implementation demonstrates superior performance versus similarly accurate BNNs, including a 6.1× decrease in area-delay product (ADP) with an 8.9× increase in energy efficiency on the KWS dataset, and a 5.0× decrease in ADP with a 3.8× increase in efficiency on ToyADMOS/car.

3. A comparison of ULEEN against two prior memory-efficient WNNs, Bloom WiSARD [130] and BTHOWeN. I show that my optimizations in ULEEN can reduce inference error by up to 5.3× and model parameter size by up to 5.5× compared to Bloom WiSARD across the KWS, ToyADMOS, and MNIST datasets. I also demonstrate that my multi-pass learning and ensemble techniques provide

significant accuracy improvements over BTHOWeN, reducing inference error by an average of 2.3×. Pruning enables a 27% reduction in model size with minimal accuracy impact, giving a final result comparable in parameter size to BTHOWeN but with far superior accuracy.

The code I developed for this work is publicly available at `https://github.com/ZSusskind/ULEEN`.

5.1 The ULEEN Model

ULEEN incorporates multiple algorithmic improvements on top of BTHOWeN, including structural changes and a novel multi-pass WNN training strategy. These enable it to outperform BTHOWeN in both accuracy and efficiency.

5.1.1 Multi-Pass, Gradient-Based Learning for WNNs

Most WNNs, including BTHOWeN, are trained using single-pass learning rules. While multi-pass WNN learning rules have been explored in prior work, they used discrete complicated search-based strategies (as opposed to gradient-based updates) and empirically resulted in models with poor accuracy (see §2.1.4.3). Despite these shortcomings, I believed this topic was worth revisiting due to the limitations of single-pass WNN learning rules such as the one used for BTHOWeN. When training BTHOWeN, patterns are presented only to the discriminator corresponding to the correct output class, without any mechanism for feedback between discriminators.[4] This means that while excitatory behaviors can be learned, inhibitory interactions between discriminators are not generally possible. However, it is well-understood that inhibition plays a crucial role in biological learning [72]. Therefore, I developed

[4]Arguably, bleaching provides a limited degree of feedback since it can account for the case where a pattern is common in one class and rare (but not nonexistent) in another. However, bleaching is not able to serve as a feedback mechanism when a pattern is common in multiple classes.

a gradient-based learning rule for ULEEN which allows training samples to be passed to all discriminators, and uses backpropagation to update RAM node entries.

5.1.1.1 Continuous Bloom Filters

The key innovation I introduce in ULEEN to enable effective gradient-based multi-pass learning is the *continuous* Bloom filter, an example of which is shown in Figure 5.1. In a continuous Bloom filter, table entries are continuous (floating-point) values in the range [-1, +1]. During a forward pass of the model, table entries are accessed using addresses generated by multiple hash functions, as with a conventional Bloom filter. The accessed table entries are then binarized using the straight-through estimator (STE) function (see §2.2.2.1). Lastly, the AND of these binarized values is output as the filter's response.[5]

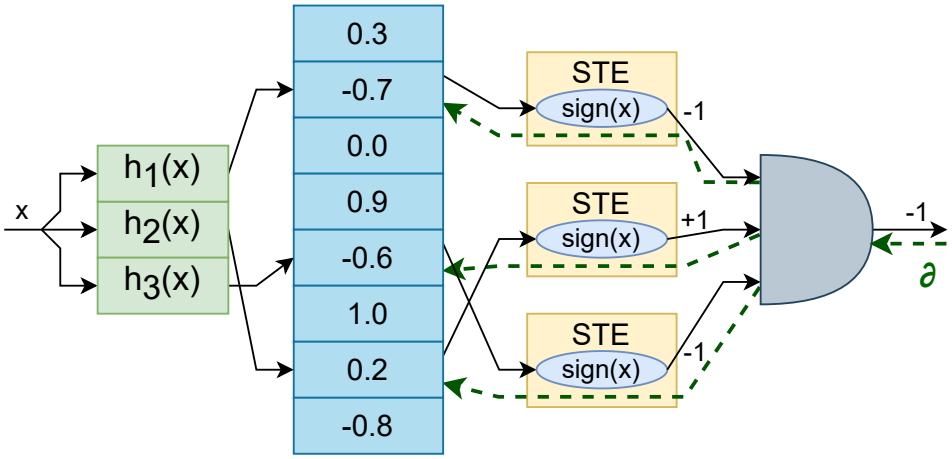


Figure 5.1: A continuous Bloom filter, a partially differentiable relaxation of the Bloom filter which is used for training ULEEN. Continuous Bloom filters have continuous-valued table entries, which are binarized using the straight-through estimator function. Entries are updated using a gradient-based multi-pass learning rule.

[5]Following the convention used by BNNs, I describe the two logic levels for Boolean values as $\{-1, +1\}$ instead of the more familiar $\{0, 1\}$. This is done for mathematical convenience during training and does not have any impact on the final generated logic.

During backpropagation, the AND gate at the end of the continuous Bloom filter receives a gradient value which indicates how its output impacts the loss function of the model. This gradient is then distributed between the inputs to the AND gate. If the output of the AND gate was +1 during the forward pass, indicating that all of its inputs were also +1, then each input receives the gradient divided by the total number of inputs (equivalently, the number of hash functions). If the output was -1, then inputs that were -1 receive the gradient divided by the number of inputs equal to -1, and the inputs equal to +1 receive no gradient. These gradients are then passed backward through the STE (which behaves as an identity function in the backwards pass since table entries are clamped to [-1, +1]) and used to update the corresponding entries in the continuous Bloom filter's table.

Continuous Bloom filters are complex to implement in hardware due to their use of floating-point values. However, like the counting Bloom filters in BTHOWeN, they can be binarized and replaced with conventional Bloom filters before inference. ULEEN does not use bleaching; instead, table entries that are less than 0 are replaced with 0 and entries that are at least 0 are replaced with 1.

5.1.2 Additive Submodel Ensembles

Ensembles, like the one shown in Figure 5.2, combine multiple small, weak classifiers into a single stronger model. Ensembles have been extensively studied in other areas of machine learning, and are the driving concept behind techniques such as Bayesian model averaging, boosting, and bagging [48]. ULEEN leverages ensembles by independently training several WNN submodels on the same training data. During inference, the response scores for each discriminator are summed across the submodels before the final argmax prediction. In other words, if a submodel i produces a response score R_{ij} for class j, then the final prediction of the model will be $\text{argmax}_{\forall j} \left(\sum_i R_{ij}\right)$.

This additive ensemble technique is similar to but distinct from bagging. In

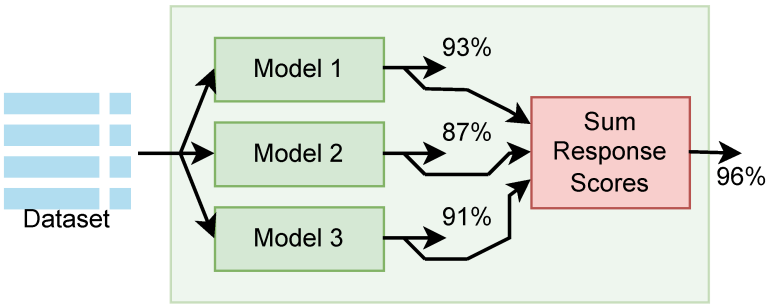


Figure 5.2: ULEEN uses "additive ensembles" of weightless submodels, where the response scores of each discriminator are summed across the submodels before the output class is predicted. Ensembles are more accurate than any of the individual submodels which compose them, and an ensemble of small submodels is usually more efficient than a single large model.

bagging, submodels are trained using random subsets of the training data to influence them to learn different patterns and behaviors. On the other hand, in ULEEN, all submodels see the same training data, but the connections from model inputs to RAM nodes are different. This sparse connectivity forces RAM nodes in different submodels to learn distinct patterns, even if their hyperparameters are otherwise identical.

One might reasonably expect that using ensembles of submodels would increase the size of a ULEEN model, since there are more RAM nodes in total. However, I have found that in practice this is usually not the case. The individual submodels of an ensemble can be made much smaller (and therefore individually less accurate) than a monolithic model without significantly degrading ensemble accuracy. Since the amount of hash computation required for inference increases with the number of submodels, I avoid using ensembles with excessively many submodels. ULEEN ensembles of numerous tiny submodels can give excellent accuracy, but they are impractical to implement in hardware due to the area overhead of hashing.

The idea of building ensembles out of WNNs has been explored previously [102]. However, the prior work which examined this subject did not see much benefit from the technique: the ensembles were far larger than the monolithic models, and only achieved a marginal improvement in accuracy. It appears that multi-pass learning

may be a critical factor to making ensembling effective for WNNs.

5.1.3 RAM Node Pruning

ULEEN also introduces a technique for pruning WNNs. In the context of DNNs, "pruning" most commonly refers to a process of identifying and eliminating the weights which contribute the least to overall model accuracy. However, there is no direct analogue to this in WNNs, since even if there were a reliable way to identify which table entries were unimportant, it is unclear how they could be eliminated without radically restructuring the table lookup. Therefore, when pruning ULEEN models, I focus on identifying and eliminating entire RAM nodes that are irrelevant or harmful to model performance.

After training a ULEEN model, I evaluate how useful each RAM node is as a predictor of the model output by calculating a "utility score". In particular, for each RAM node N_{ij} in discriminator d_i, the pruning algorithm observes whether N_{ij} outputs a 1 when the correct class label is i. This is then used to compute true and false positive and negative rates for the RAM node. For a dataset with M classes, the utility score for a RAM node is given by $(M-1)(TPR-FNR)+(TNR-FPR)$. Next, a fixed fraction of the RAM nodes in each discriminator with the lowest utility scores are removed from the model.

Eliminating RAM nodes reduces the maximum possible response score of each discriminator. The impact that this has can vary between discriminators. For instance, a RAM node that always outputs 0 and a RAM node that always outputs 1 are equally useless, but removing them will have different impacts on a discriminator's average response score. To compensate for this, I next learn an integer bias for each discriminator, which is added to its output. Finally, I perform a fine-tuning training pass to restore some of the accuracy that was lost due to pruning.

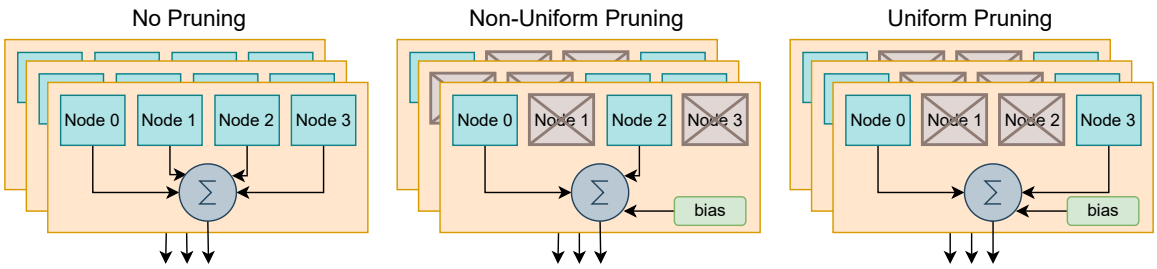


Figure 5.3: Two strategies for pruning ULEEN models. Non-uniform pruning eliminates the same number of RAM nodes in each discriminator, but allows those RAM nodes to be picked at arbitrary indices. Uniform pruning adds the additional constrain that the pruned RAM nodes be at the same indices in all discriminators. Uniform pruning can have a larger impact on accuracy, but enables additional optimizations in hardware.

5.1.3.1 Non-Uniform and Uniform Pruning

As shown in Figure 5.3, I explored two variations of the RAM node pruning strategy. "Non-uniform" pruning requires the same number of RAM nodes to be eliminated in each discriminator, but does not otherwise constrain which RAM nodes are removed. This approach uses per-discriminator metrics to determine which RAM nodes are least useful. On the other hand, the "uniform" pruning strategy averages utility scores across the discriminators, meaning RAM nodes are removed at common indices in all discriminators.

The non-uniform pruning strategy implicitly recognizes that a particular input tuple may be useful for predicting some classes but not others. By choosing the most important RAM nodes in each discriminator individually, the impact on accuracy from pruning can be reduced. However, eliminating RAM nodes at the same index in all discriminators is more efficient in hardware. This is because ULEEN, like BTHOWeN, shares the same input-to-tuple mapping for all discriminators, meaning Bloom filter hash computation only needs to be performed once for all discriminators. If the RAM node at a given index is eliminated in some discriminators but preserved in others, then the hash functions for this RAM node must still be computed once for the remaining discriminators. However, if a RAM node is eliminated in *all* discriminators,

Dataset	Unpruned		≤1% Drop		≤3% Drop	
	Size (KiB)	Test Acc.%	Size (KiB)	Test Acc.%	Size (KiB)	Test Acc.%
MNIST	373	98.49%	112	97.91%	38.1	95.99%
FashionMNIST	2786	89.20%	1533	88.62%	412	86.61%
Letter	30.9	93.13%	23.6	92.27%	15.8	90.28%
Satimage	5.53	88.30%	3.66	87.30%	1.17	85.35%

Table 5.1: Pruning sensitivity study results for additional models and datasets. Pruning rations were chosen for ≤1% and ≤3% drops in inference accuracy.

which is guaranteed with uniform pruning, then the hash computation can be skipped. This can potentially save hardware area, even if more RAM nodes must be preserved overall to maintain model accuracy.

To understand which strategy was superior in practice, I explored pruning up to 98% of RAM nodes from ULEEN using both the uniform and non-uniform techniques. A result from this sweep is shown in Figure 5.4. In this example, the two techniques have almost identical impacts on model accuracy up to a ∼70% pruning ratio. While non-uniform pruning can eliminate hash computations if a RAM node is independently pruned in all discriminators, even at a 70% pruning ratio, the reduction in hash computation from this is only 7.1%, versus 70% for uniform pruning. While non-uniform pruning is significantly more accurate at very high (≥90%) pruning ratios, this can be offset by just using a slightly lower ratio with uniform pruning. Therefore, the uniform strategy is generally the superior approach for pruning ULEEN models.

Table 5.1 shows sweeping results with uniform pruning for some additional ULEEN models on other datasets. On average, 40% of RAM nodes can be pruned with ≤1% loss in accuracy, and 74% can be pruned with ≤ 3% loss in accuracy. Although this does not approach the degree of weight pruning that is frequently possible for DNNs, it is still a notable reduction in memory requirements and circuit area. When used in the context of an ensemble, the bias terms that are introduced

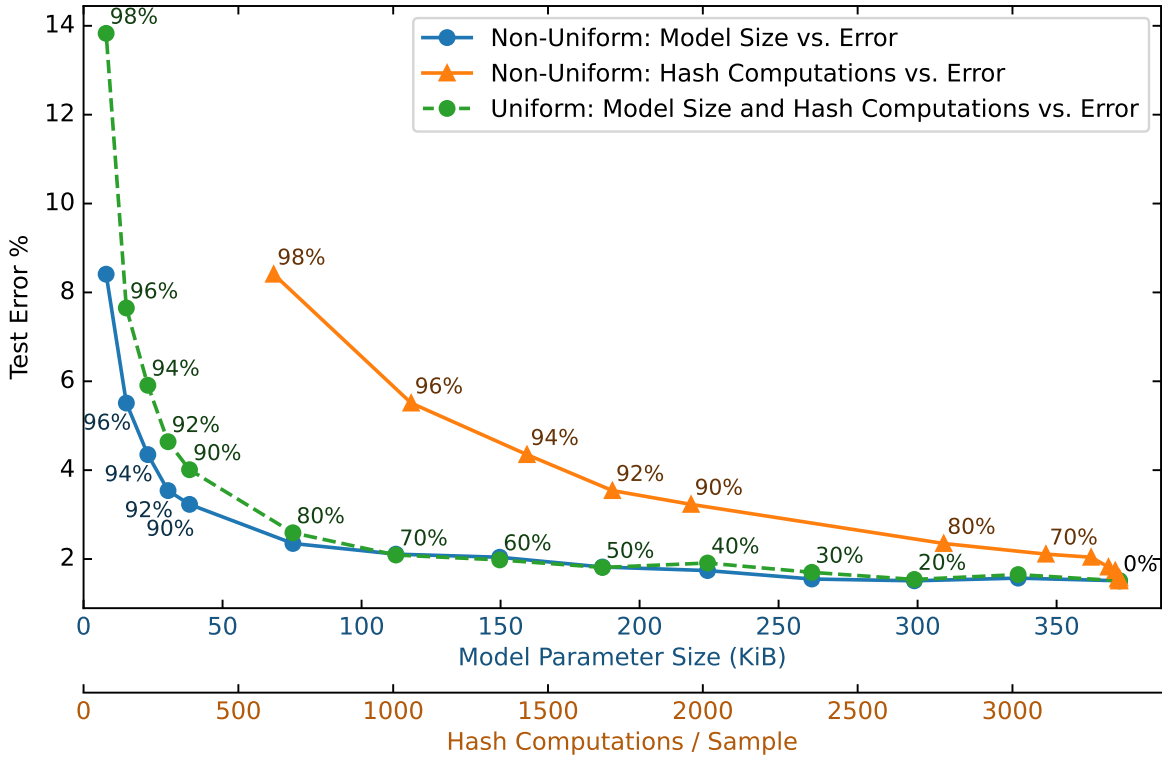


Figure 5.4: Pruning sensitivity study for a ULEEN model for the MNIST dataset. The uniform and non-uniform pruning strategies have almost identical performance with low-to-moderate pruning ratios, but the uniform strategy is much more effective at eliminating hash computations.

by pruning can be summed across submodels, meaning the only inference overhead incurred by pruning is a single bias addition for each output class.

5.2 ULEEN Software Model

Figure 5.5 shows what a ULEEN model looks like during the initial training phase. Submodels are trained separately, meaning they each perform a separate softmax and cross-entropy loss calculation, and do not share gradients. The summation of discriminator responses across classes is used exclusively for inference and is not shown in this figure. This figure also does not show RAM node pruning, which is not performed until after initial training.

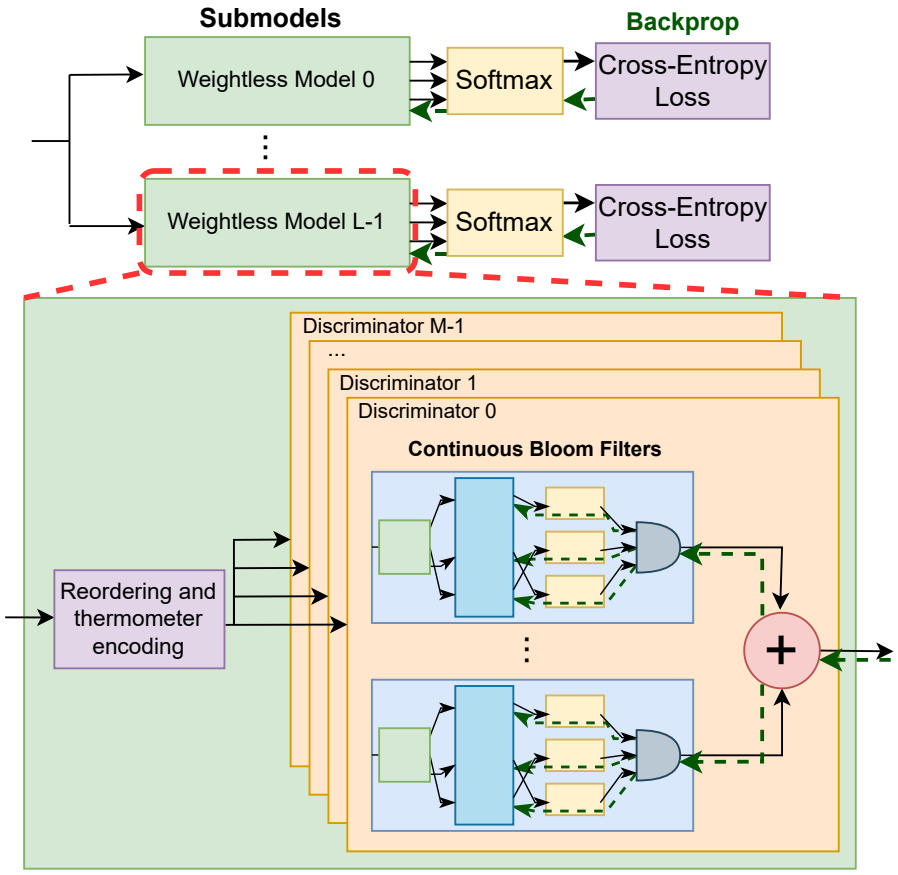


Figure 5.5: An overview of a ULEEN model during training. ULEEN is composed of an ensemble of submodels, each of which is a itself a WNN. Submodels are composed of discriminators, which use continuous Bloom filters during training to enable gradient-based weight updates. Submodels are trained independently, with their losses computed separately.

Figure 5.6 summarizes the multi-pass training flow for ULEEN. Inputs are encoded with the same Gaussian thermometer strategy I used in BTHOWeN. The entries of the continuous Bloom filters are randomly initialized between -1 and 1 and iteratively updated using backpropagation with the straight-through estimator. Note that gradients are not backpropagated through the hash functions, and thermometer encoding thresholds therefore are not updated during training.

To provide regularization and prevent overfitting, I apply dropout with prob-

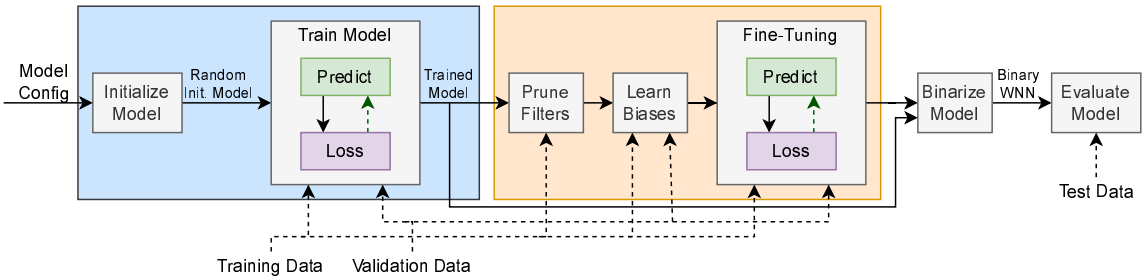


Figure 5.6: Training procedure for ULEEN models. After an initial training process, the least useful RAM nodes are pruned, followed by additional passes to learn discriminator biases and fine-tune the pruned model. Lastly, the finished model is binarized, eliminating floating-point arithmetic and preparing it for inference.

ability p=0.5 to the outputs of the RAM nodes of the larger ULEEN models during training. Without this regularization, I observed that models would frequently perfectly memorize their training data. Dropout randomly cancels some values and rescales the rest. This results in the possible continuous Bloom filter outputs becoming {-2, 0, +2} during training. This strategy improves generalization and does not result in any overhead during inference.

I train ULEEN models using the Adam [83] optimizer with a base learning rate of 10^{-3}. Initial filter entry values are uniformly sampled (i.e., drawn from $\mathcal{U}(-1, 1)$). I also use a simple form of data augmentation for the MNIST dataset: each image in the training data is copied 9 times, with copies shifted between -1 and 1 pixels horizontally and vertically. After training, models are pruned and fine-tuned as described previously. Lastly, the continuous Bloom filters are statically binarized by applying the sign function and then rescaled from {-1, 1} to {0, 1}, which transforms them into conventional Bloom filters. Discriminator output biases are also adjusted to account for this transformation.

5.3 ULEEN Inference Accelerator

Figure 5.7 shows the block diagram for my pipelined ULEEN inference accelerator architecture. During the evaluation of BTHOWeN, I noticed that the bandwidth

of the off-chip input bus was usually the throughput bottleneck for models. Therefore, ULEEN introduces an optional compression scheme for input data. This scheme replaces unary thermometer-encoded values with binary integers representing how many bits are set. Since thermometer bits are always set from least to most significant, this process is easily reversed to recover the original encoding. Compression reduces the amount of data movement for a k-bit thermometer encoding by a factor of $\frac{k}{\lfloor \log_2(k) \rfloor + 1}$, but requires hardware support for decompression (shown in the top left of the figure). The decompression block is composed of functional units which compare inputs against a series of increasing constant values. The hardware for these functional units is simple, and they can be time-multiplexed like the hash units, so I always use compression for ULEEN models with three or more thermometer bits per input.

The discriminators in Figure 5.7 have several refinements over their predecessors in the BTHOWeN accelerator architecture. Most notably, they have been improved to take advantage of the sparsity which is created by pruning. Bloom lookup units corresponding to RAM nodes which have been pruned can be eliminated entirely. Additionally, the design automatically detects and eliminates unnecessary hash function computations. The number of hash functional units to generate is determined by the dataset sample inputs, thermometer encoding width, compression scheme, hash functions per Bloom filter, and non-pruned Bloom filters per discriminator, and is chosen to be as small as possible without bottlenecking the design.

Each submodel in an ensemble must compute its own hashes since input orders, hash input and output widths, and random hash parameters vary. Since different submodels have different table contents, sizes, and prunings, they also have their own sets of filter units. Bloom filter outputs for the discriminators in each submodel are concatenated and a popcount is performed, which combines the popcount step of WiSARD with the additive ensemble aggregation. The aggregated pruning bias values are added to the resultant response scores, and the index of the strongest response is computed and used as the final prediction.

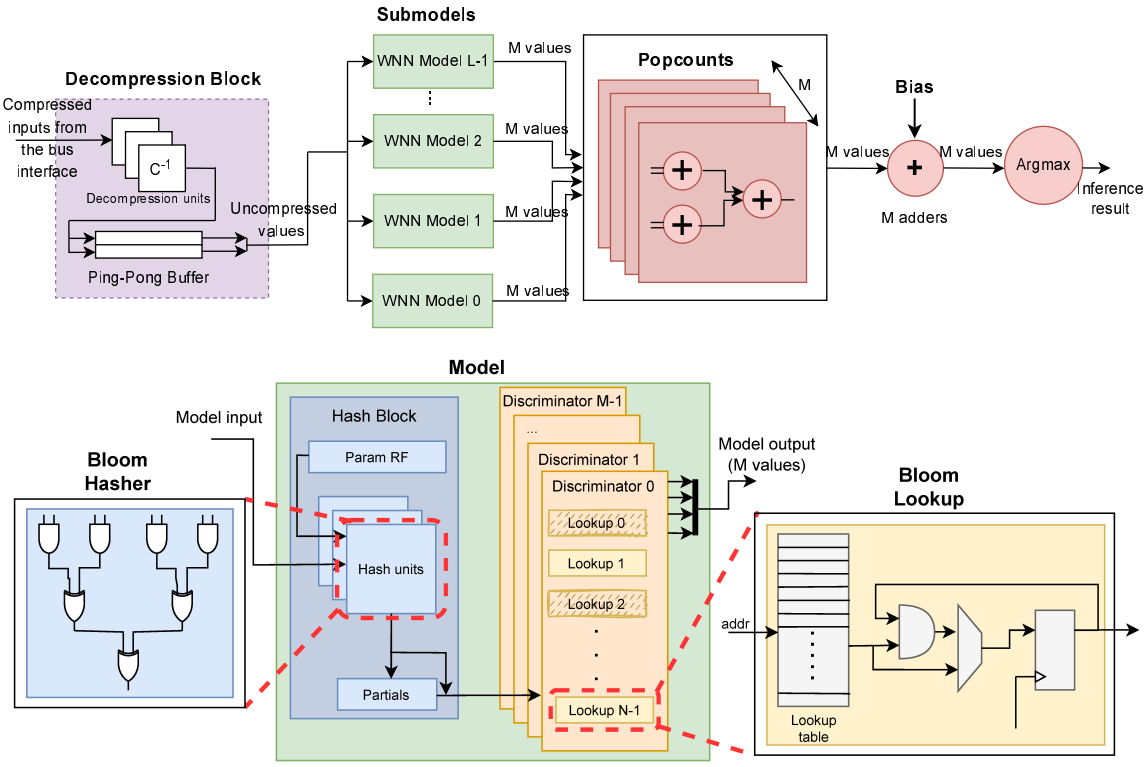


Figure 5.7: Inference accelerator architecture for ULEEN. Input is deserialized and, if needed, decompressed, before being passed to an ensemble of submodels. Each submodel contains a hardware block for computing hash functions and a set of hardware units for performing lookups. These units have been improved from BTHOWeN to take advantage of pruning sparsity. The outputs of the submodels are summed and biased to get per-class response scores. The index of the strongest response score is taken as the predicted class.

5.4 Evaluation Methodology

I compare my accelerator architecture for ULEEN against Xilinx's FINN [145] framework for BNN inference on FPGAs. FINN does not itself propose a novel BNN algorithm, but is instead a tool for generating hardware accelerators for pretrained BNNs. Therefore, comparing ULEEN against FINN also gives a sense of its performance relative to the broader domain of BNN literature. I use fully-connected, MLP-style FINN models for this comparison. While FINN supports generating hardware for both fully-connected and convolutional BNNs, I focus solely on the former

since ULEEN is not a convolutional architecture.

FINN only provides fully-connected model results for MNIST. They propose three network topologies, SFC, MFC, and LFC, which each contain three hidden layers with 256, 512, and 1024 neurons per layer, respectively. They also propose throughput-optimized "max" and area-optimized "fix" FPGA implementations for each of these models. I compared against the "max" implementations in this work, since the ULEEN accelerator is also optimized for high peak throughput. For the MLPerf Tiny datasets, I trained BNN models for FINN using the Xilinx Brevitas [118] low-precision machine learning library. I used three hidden layers for these models as well, with differing numbers of neurons per hidden layer.

5.4.1 Datasets

In addition to MNIST, I use the four datasets in MLPerf Tiny [23] for comparisons against FINN in this work. These datasets are:

1. **Keyword spotting (KWS):** This dataset is extracted from Speech Commands v2 [153], which consists of 105,829 utterances from 2,618 speakers. It consists of spectrograms representing ten different keywords, plus an "unknown word" category.

2. **ToyADMOS/car:** This dataset consists of audio recordings of seven different toy cars [84]. The objective is to identify "anomalous" samples collected from deliberately damaged cars.

3. **Visual Wake Words (VWW):** This dataset consists of 96×96 grayscale images extracted from the MSCOCO 2014 dataset [98]. The objective is to determine whether an image contains at least one person.

4. **CIFAR-10:** An image classification dataset consisting of 32×32 RGB images in 10 classes [87].

5. **MNIST:** Image classification, with 28×28 grayscale images of the digits 0–9 [92].

5.4.1.1 A note on positional independence

Not all of these datasets are well-suited to MLPs or ULEEN. In particular, CIFAR-10 and VWW exhibit high degrees of positional independence, where image features may be present in different locations and at different scales. However, these datasets are still interesting to explore with ULEEN because CNNs are difficult to implement on extreme edge devices. For instance, FINN's convolutional designs are orders of magnitude slower and less efficient than their fully-connected models.

5.4.2 Implementation

I implement the software for training ULEEN using custom modules and LibTorch C++ extensions for the PyTorch machine learning library. This allows me to leverage GPU acceleration during training. The forward and backward passes for Bloom filters are implemented as a single multidimensional gather/scatter operation, which enables efficient memory-parallel computation, but is still bottlenecked by the GPU's memory bandwidth.

FINN is designed to leverage Brevitas [118], an open-source library by Xilinx for creating low-precision and binary neural networks. I used Brevitas to train new binary MLP models for MNIST and the four datasets in MLPerf Tiny [23]. My co-author [139] A. Arora then passed these models through the FINN HLS flow to create optimized FPGA implementations for the BNNs. The FINN compiler applies a series of transformations to convert the network's nodes to layers, which invoke functions in a highly optimized `finn-hls` library, using time-multiplexing (referred to in the FINN documentation as "folding") to reduce execution resources. The resultant C++ code is then passed through Xilinx Vivado HLS to generate the RTL. We also attempted to replicate the hardware generation for the SFC, MFC, and LFC models. However, our

FINN implementations were significantly less efficient than Xilinx's published results. This suggests that these models may have been originally hand-tuned to an extent that we were not able to replicate. Therefore, I use the original, superior values from Xilinx for these three models in our comparison with ULEEN.

For the FPGA implementation and comparison, I target the Zynq Z7045 SoC platform, which was also used to report results for FINN. I also use the same I/O interface width as FINN (112 bits), and target the same frequency of operation (200 MHz). However, in some cases I was unable to achieve this frequency due to routing congestion.

5.5 Results

5.5.1 Software Model Comparison of ULEEN with BNNs

Table 5.2 shows the seven ULEEN models I created for comparison with FINN: three for the MNIST dataset with varying model sizes and accuracies and four for the MLPerf Tiny datasets. I chose the three MNIST models to specifically compare against FINN's SFC, MFC, and LFC models. The other four models were chosen as configurations with a good balance between accuracy and parameter size. The table includes two FINN models for each ULEEN model: one intended to match ULEEN in accuracy ("FINN BNN Iso-Accuracy" in the header), and one to match ULEEN in parameter size ("FINN BNN Iso-Size"). The iso-accuracy FINN models for MNIST (SFC, MFC, and LFC) are from the original FINN paper; I created the other eleven models using the Xilinx Brevitas [118] library. The FINN models I created all have three hidden layers of equal size, since this was the approach that was used in the original paper for SFC, MFC, and LFC. FINN was unable to match ULEEN's accuracy on VWW and CIFAR-10 even when I made the parameter sizes of the BNN models more than an order of magnitude larger.

Overall, ULEEN is consistently able to match the accuracies of binary MLPs that were trained using Brevitas while maintaining a smaller parameter size, or match

Dataset	ULEEN Size (KiB)	ULEEN Test Acc.%	FINN Iso-Accuracy Size (KiB)	FINN Iso-Accuracy Test Acc.%	FINN Iso-Accuracy Hidden Layers	FINN Iso-Size Size (KiB)	FINN Iso-Size Test Acc.%	FINN Iso-Size Hidden Layers
MNIST-S	16.9	96.2	40.8	95.8	256×3 (SFC)	16.4	95.2	128×3
MNIST-M	101	97.8	114	97.7	512×3 (MFC)	103	97.7	480×3
MNIST-L	262	98.5	355	98.4	1024×3 (LFC)	283	98.0	896×3
KWS	101	70.3	324	70.6	1024×3	101	67.0	524×3
ToyADMOS	16.6	86.3	36.1	86.1	256×3	16.4	85.5	144×3
VWW	251	61.8	3329*	57.1*	2048×3*	264	55.7	224×3
CIFAR-10	1379	54.2	19466*	45.7*	8192×3*	1345	44.4	1700×3

Table 5.2: Comparison between ULEEN and prior work (FINN) for MNIST and MLPerf Tiny. ULEEN is smaller than similar-accuracy FINN models, and more accurate than similar-size FINN models. (*FINN is unable to achieve ULEEN's accuracy for VWW or CIFAR-10.)

their parameter sizes while achieving a higher accuracy. In particular, compared to iso-accuracy FINN models, ULEEN has 1.1× to 3.2× fewer parameters, with a geometric mean reduction of 1.9×[6]. Compared to iso-parameter-size FINN models, ULEEN is 0.1% to 9.8% more accurate, with a mean improvement of 3.1%.

Detailed information on the seven ULEEN models I selected is presented in Table 5.3. All submodels use two hash functions per Bloom filter; using more than this typically does not provide any benefit for ULEEN, particularly when considering the additional hardware area for hashing it entails. I include accuracies for both the full ULEEN ensembles and all of their component submodels individually. In most cases, the ensemble is more than 4% more accurate than the best single submodel. This demonstrates that the additive submodel ensemble technique I developed for ULEEN is effective across a range of applications.

As anticipated, ULEEN and FINN have poor accuracy on the VWW and

[6]These figures for parameter size reduction exclude the iso-accuracy FINN models for the VWW and CIFAR-10 datasets since they could not match the accuracy of ULEEN. If we include these two models, then the maximum improvement of ULEEN over FINN increases to 14.1× and the geometric mean to 3.4×.

Dataset	Model	Submodel	Bits/Inp	Inps/Filter	LUT Entries	Size (KiB)	Test Acc.%
MNIST	ULN-S	**Ensemble**	**2**	—	—	**16.9**	**96.20**
		SM0	”	12	64	7.19	92.91
		SM1	”	16	64	5.39	90.25
		SM2	”	20	64	4.38	86.16
	ULN-M	**Ensemble**	**3**	—	—	**101**	**97.79**
		SM0	”	12	64	10.9	83.54
		SM1	”	16	128	16.0	90.93
		SM2	”	20	256	26.0	92.92
		SM3	”	28	256	18.44	87.05
		SM4	”	36	512	29.38	80.93
	ULN-L	**Ensemble**	**7**	—	—	**262**	**98.46**
		SM0	”	12	64	25.0	88.78
		SM1	”	16	128	37.7	93.24
		SM2	”	20	128	30.2	92.44
		SM3	”	24	256	50.3	93.92
		SM4	”	28	256	43.1	90.47
		SM5	”	32	512	75.6	90.44
KWS		**Ensemble**	**12**	—	—	**101**	**70.34**
		SM0	”	5	8	9.62	56.93
		SM1	”	6	16	16.1	59.32
		SM2	”	7	32	27.5	59.94
		SM3	”	8	64	48.12	61.01
ToyADMOS		**Ensemble**	**6**	—	—	**16.6**	**86.33**
		SM0	”	7	64	6.88	83.61
		SM1	”	9	64	5.34	82.32
		SM2	”	11	64	4.38	79.85
VWW		**Ensemble**	**12**	—	—	**251**	**61.76**
		SM0	”	5	8	30.2	59.07
		SM1	”	7	16	43.2	57.78
		SM2	”	9	32	67.2	59.20
		SM3	”	11	64	110	58.96
CIFAR-10		**Ensemble**	**8**	—	—	**1379**	**54.21**
		SM0	”	6	32	112	49.12
		SM1	”	8	64	168	49.53
		SM2	”	12	128	224	46.39
		SM3	”	16	256	336	42.23
		SM4	”	20	512	538	38.27

Table 5.3: Details of the selected ULEEN models. SMi refers to the ith individual submodel comprising an ensemble.

CIFAR-10 datasets due to their high degree of positional variance (e.g., a person could appear in the top left or bottom right of an image). Since ULEEN and the FINN models I use for comparison do not incorporate convolution, they struggle to learn position-independent features. Thus, while ULEEN is well-suited for applications with little positional variance, such as tabular datasets, it is not a universally applicable machine learning model. Since both ULEEN and FINN perform poorly on these two datasets, I do not consider them in the FPGA implementation analysis.

5.5.2 FPGA Implementation Comparison of ULEEN with FINN

Detailed comparisons between my FPGA implementations of ULEEN and the iso-accuracy FINN models are shown in Table 5.4. I report dynamic energy for a single inference in isolation (batch size b=1) and steady-state inference (b=∞). The results that were reported by Xilinx for their SFC, MFC, and LFC FINN models [145] include total power but not dynamic power. Since I did not have access to their RTL to gather the data for these experiments directly, I extrapolated by assuming a constant 0.3W of static power. This value was derived from the average static power of the other FINN models, as synthesis results showed very little variation in static power between designs.[7]

Overall, ULEEN reduces energy per inference by an average of 8.3× (5.5–11.7×; geometric mean) for a single inference, and by 7.1× (3.8–9.1×) for steady-state inference. While ULEEN also has superior latency and throughput to FINN (by 5.3× and 3.7×, respectively), this is more difficult to compare directly due to the different areas of the two models. Accounting for this, ULEEN's area-delay product is on average 4.5× (1.7–7.8×) lower than FINN's, with 3.6× (1.7–6.1×)

[7]I report dynamic rather than total energy because the FINN models for KWS and ToyADMOS are much more serial than the ULEEN models. This is due to a limitation of the FINN synthesis tool: "folding" (time-multiplexing) factors must be integer divisors of the layer sizes, restricting the possible model topologies. The serial nature of these models results in a small circuit area, but their static energy consumption is high since FPGAs have limited capability to power down unused regions of the chip. Therefore, using dynamic energy is more fair and favorable to FINN.

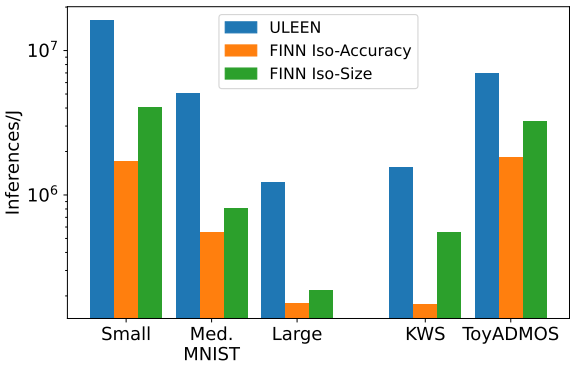


Figure 5.8: Energy efficiency (steady-state inferences per Joule) comparison between FINN and ULEEN.

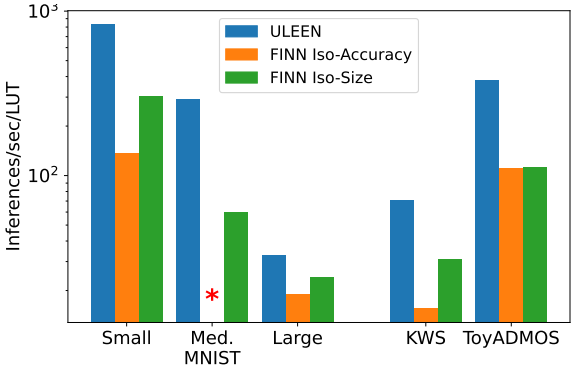


Figure 5.9: Area efficiency (inferences per second per LUT) comparison between FINN and ULEEN (*LUT usage is not provided for the medium MNIST iso-accuracy FINN model in [145]).

higher throughput per unit area.[8] Therefore, ULEEN is substantially superior to the iso-accuracy FINN models in energy efficiency, latency, and throughput, even after adjusting for differences in model areas.

Figures 5.8 and 5.9 compare ULEEN's energy efficiency and throughput per unit area against the iso-accuracy and iso-size FINN models. ULEEN's advantage over the iso-size models is smaller than with the iso-accuracy models, but it still outperforms even these less accurate BNNs. ULEEN's steady-state energy efficiency is on average 3.8× (2.2–6.2×; geometric mean) better than the iso-size FINN models, and its throughput per unit area is 2.7× (1.4–4.8×) higher.

To summarize, ULEEN is faster and more efficient than even less accurate MLP-style BNNs in an optimized FPGA implementation. While it is not suitable for all workloads (particularly large image datasets), it is a strong option for applications which place less importance on positional invariance, such as tabular datasets.

[8] I use LUT utilization as a proxy for total circuit area. Calculating area in this way is to FINN's advantage since it also uses BRAMs while ULEEN does not, but Xilinx does not publish the information necessary to estimate the LUT-equivalent area of a BRAM (see also §3.3). Using unnormalized throughputs, ULEEN is 1.2–15.0× faster, with a geometric mean of 3.7×.

Model	Latency (μs)	Xput (kIPS)	Dynamic b=1	μJ/Inf. b=∞	LUT	BRAM	Test Acc.%
ULN-S	0.21	14,286	0.191	0.062	17,319	0	96.2
FINN-SFC	0.31	12,361	2.170	0.566	91,131	4.5	95.8
ULN-M	0.29	14,286	0.823	0.199	49,445	0	97.8
FINN-MFC	—	6,238	—	1.763	—	—	97.7
ULN-L	0.94	4,048	3.137	0.823	123,102	0	98.5
FINN-LFC	2.44	1,561	20.74	5.445	82,988	396	98.4
ULN-KWS	0.39	10,000	2.536	0.642	141,074	0	70.3
FINN-KWS	7.78	668	29.72	5.716	42,847	151.5	70.6
ULN-ToyADMOS	0.34	11,111	0.549	0.143	29,404	0	86.3
FINN-ToyADMOS	3.52	1,568	3.022	0.547	14,100	34.5	86.1

Table 5.4: Comparison of ULEEN against iso-accuracy FINN models for MNIST, KWS, and ToyADMOS. Note that some results are not available for the FINN MFC model, as they were not provided in the paper the data was drawn from [145].

5.5.3 Sensitivity Analysis

Figure 5.10 shows the impacts of the optimizations introduced in BTHOWeN and ULEEN on the accuracies and parameter sizes of WNN models. Results are shown for three datasets: MNIST, KWS, and ToyADMOS. The seven results shown in each graph are for, from left to right, (1) Bloom WiSARD [130], the prior WNN state-of-the-art, (2) a variant of Bloom WiSARD that uses counting Bloom filters for bleaching, (3) a further variant that also incorporates a simple "linear" thermometer encoding, (4) BTHOWeN [140], which uses a Gaussian thermometer encoding, (5) a variant of BTHOWeN which uses my multi-pass learning rule with continuous Bloom filters, (6) an ensemble of such models, and (7) the final, pruned ULEEN [139] models.

ULEEN is significantly more accurate than my prior work, BTHOWeN, which in turn is both more accurate and smaller than the earlier Bloom WiSARD model. The BTHOWeN models shown in Figure 5.10 have on average 4.2× fewer parameters than the corresponding Bloom WiSARD models and reduce average test error by an average factor of 1.5×. The ULEEN models have approximately the same parameter

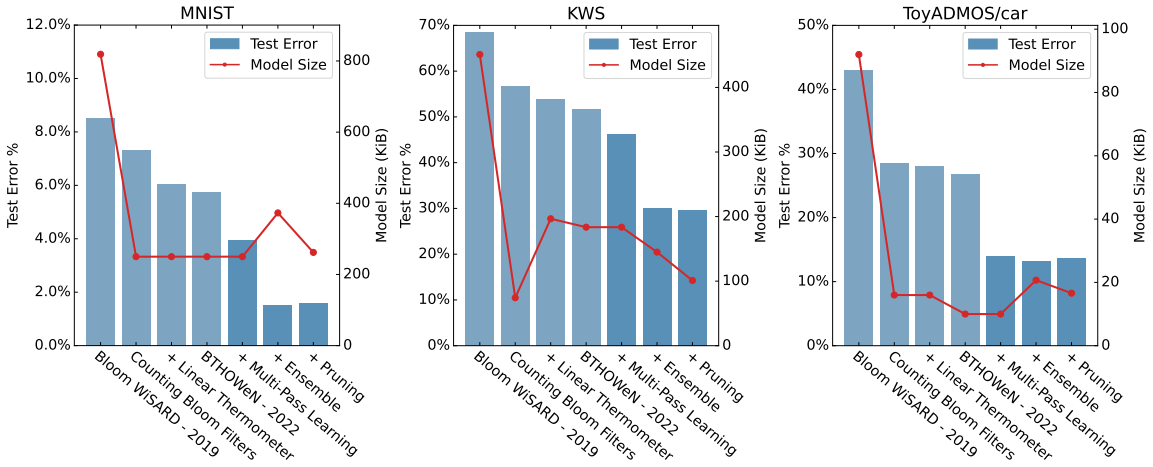


Figure 5.10: Accuracy and model size of WNNs as progressive model improvements are applied. Results are included for the MNIST, KWS, and ToyADMOS datasets. All entries except the first ("Bloom WiSARD") represent improvements which I have contributed in this dissertation.

sizes as the BTHOWeN models (a deliberate choice I made when selecting models for comparison) but reduce test error by an additional factor of 2.3×. Overall, ULEEN models are 3.1–5.5× smaller than Bloom WiSARD, with 2.3–5.3× lower error.

Comparing BTHOWeN to Bloom WiSARD, it is evident that the ability to reject rare input patterns (using counting Bloom filters in BTHOWeN, and continuous Bloom filters in ULEEN) is critical for model accuracy and efficiency. This change alone reduces test error by 1.3× and model size by 4.9× compared to the Bloom WiSARD baseline. Using a linear or Gaussian thermometer encoding provides some additional reduction in model error, with the Gaussian encoding providing a greater reduction than the linear approach. However, thermometer encoding also increases the model size. To counter this, I decrease the size of the Bloom filters, which negates some of this improvement in accuracy.

About 55% of the improvement from BTHOWeN to ULEEN is attributable to the multi-pass learning rule, with the remainder coming from the addition of submodel ensembles. Pruning reduces model sizes by an average of 27% with at most a 0.6% increase in test error. In fact, for KWS, accuracy actually *improved* due to the

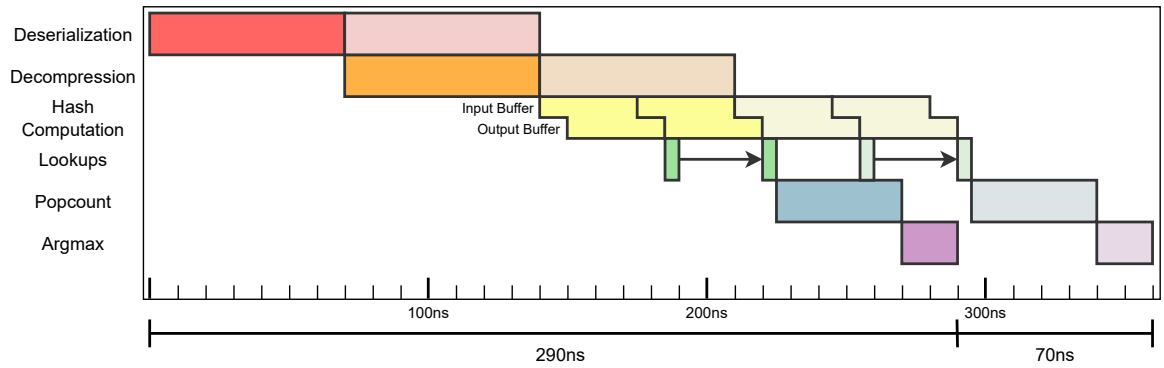


Figure 5.11: Breakdown of the time spent in each stage for inference with ULEEN, showing two inputs being processed in a pipelined fashion.

elimination of noise from low-utility RAM nodes. These results, combined with the low inference overhead of the optimizations I propose, demonstrate the superiority of ULEEN over prior WNNs for edge inference.

Figure 5.11 shows the occupancy of the ULEEN accelerator's pipeline stages during inference, using the ULN-M model as an illustrative example. The pipelined design of the accelerator enables substantial overlapping of computation, with 290ns of latency but one sample finished every 70ns in the steady state. The hash computation stage is itself internally pipelined, as its functional units have a two-cycle latency. The latencies of the decompression and hash computation stages can be tuned by instantiating different numbers of functional units. While by default the accelerator only has enough functional units for these two stages to match the throughput of the deserialization stage, increasing their counts could decrease the latency of ULN-M to a theoretical minimum of 170ns (a 41% reduction). However, this would not provide any benefit to throughput.

5.6 Comparing ULEEN with Xilinx LogicNets[9]

LogicNets [146] is another project for efficient FPGA inference by the same team at Xilinx that developed FINN. LogicNets is interesting because, like ULEEN, it uses lookup tables to perform computation during inference. However, unlike ULEEN, LogicNets is trained as a sparsely-connected mixed-precision DNN, which is then converted into a LUT-based model (see §2.2.3.1). I created ULEEN models and accelerators for the two datasets used in this paper: UNSW-NB15, a network intrusion detection (NID) dataset for identifying malicious internet traffic, and JSC, a particle physics dataset based on classifying observations from the Large Hadron Collider.

UNSW-NB15 is a particularly important dataset since NID is a major issue on edge and IoT devices. For instance, in 2016, the Mirai botnet used hundreds of thousands of compromised IoT devices to launch distributed denial-of-service attacks of sufficient magnitude to overload internet infrastructure [85]. Descendants of Mirai continue to pose a threat. While UNSW-NB15 is targeted at general NID rather than edge devices specifically, there is a more recent dataset for IoT devices by the same research group, BoT-IoT. Therefore, I gathered results for both datasets.

UNSW-NB15 is an imbalanced dataset, with approximately 22 "normal" samples for every "attack" sample. I performed random oversampling on the training data to rebalance it to 1:1, and rebalanced the test data to 2:1, which was the ratio used for LogicNets. BoT-IoT is imbalanced to a far larger degree, containing 73,370,344 "attack" samples and 9,531 "normal" samples. I used ADASYN [68] to generate synthetic training data for this dataset, and rebalanced the test data to 1:1 using random oversampling (meaning there was no synthetic data in the test set).

[9](Co-first author) Shashank Nag, Zachary Susskind, Aman Arora, Alan T. L. Bacellar, Diego L. C. Dutra, Igor D. S. Miranda, Krishnan Kailas, Eugene B. John, Mauricio Breternitz Jr., Priscila M. V. Lima, Felipe M. G. França, and Lizy K. John. LogicNets vs. ULEEN: Comparing two novel high throughput edge ML inference techniques on FPGA. In *Proceedings of the Midwest Symposium on Circuits and Systems*, MWSCAS '24, page to appear. Institute of Electrical and Electronics Engineers, 2024

I used the augmented BoT-IoT data to train models for LogicNets (using the same topology they employed for UNSW-NB15) and ULEEN.

Table 5.5 shows the ULEEN models I selected for this comparison. Unlike the models in Table 5.3, these use only one hash function per Bloom filter. This is because LogicNets performs their evaluation assuming their accelerator is part of a larger FPGA-based design, and uses an arbitrarily high input bus width in order to maximize throughput. Since the ULEEN accelerator can perform at most one full hash function per cycle, using two per Bloom filter would halve its potential throughput. This was not a concern for the previous designs because they were bottlenecked by the width of the off-chip data bus.

Dataset	**Submodel**	**Bits/ Input**	**Inputs/ Filter**	**Entries/ Filter**	**Size (KiB)**	**Test Acc.%**	**LogicNets Acc.%**
UNSW-NB15	Monolithic	4	10	64	0.19	98.92	91.3
BoT-IoT	Monolithic	8	12	128	0.53	99.38	88.6
JSC	**Ensemble**	**32**	—	—	**19.2**	**71.25**	71.8
	SM0	”	8	128	1.25	64.97	
	SM1	”	10	256	2.03	65.06	
	SM2	”	12	512	3.44	63.49	
	SM3	”	14	512	3.12	63.82	
	SM4	”	16	1024	5.00	63.20	
	SM5	”	20	1024	4.38	63.19	

Table 5.5: Selected ULEEN models for LogicNets comparison.

ULEEN is much more accurate on the UNSW-NB15 and BoT-IoT datasets, but is somewhat less accurate for JSC. Figure 5.12 compares the parameter sizes of the ULEEN models with LogicNets. Before post-training LUT conversion of the LogicNets models, ULEEN again wins on UNSW-NB15 and BoT-IoT, while the JSC ULEEN model is somewhat larger than LogicNets. After LUT conversion, the parameter footprints of the LogicNets models increase by orders of magnitude, and ULEEN is far superior on all datasets.

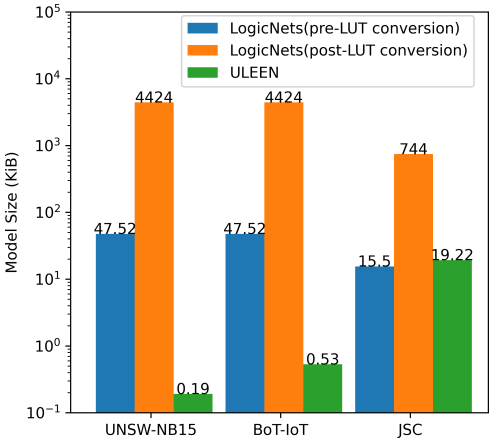


Figure 5.12: Comparison of model parameter sizes between LogicNets and ULEEN. LogicNets results are shown before and after converting neurons into LUTs.

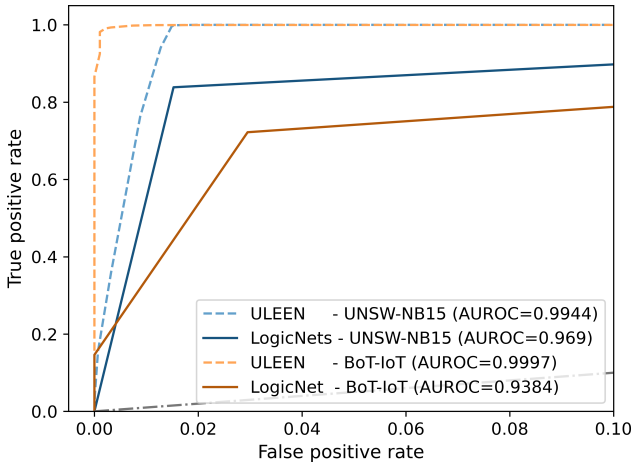


Figure 5.13: Receiver operating characteristic (ROC) curves for LogicNets and ULEEN. ULEEN has considerably higher area under the ROC, indicating superior performance.

Figure 5.13 shows the receiver operating characteristic (ROC) curves for ULEEN and LogicNets on the UNSW-NB15 and BoT-IoT datasets. ROC curves are a useful figure of merit for binary classifiers, since they evaluate performance at all possible classification thresholds. The area under the curve is significantly larger for ULEEN on both datasets, indicating superior classification performance. (Data for LogicNets in this figure was collected by S. Nag.)

Table 5.6 shows FPGA implementation results for ULEEN compared against those for LogicNets. LogicNets targets a high-end Xilinx Virtex UltraScale+ FPGA[10] for their designs, runs synthesis in out-of-context mode, and uses as high of a clock speed as possible. I use the same approach for ULEEN. Clock frequencies are equal to throughputs for all designs since they have an initiation interval of one sample per cycle, enabled by very wide input buses (for this reason, the ULEEN models also do not use input compression). While I lack LogicNets synthesis results for BoT-IoT, they would likely be close to those for UNSW-NB15 since the model topology

[10]Part number `xcvu9p-flgb2104-2-i`

Dataset	Model	Clock (MHz)	Xput (kIPS)	Dyn.Energy (pJ/Inference)	LUTs	FFs
UNSW-NB15	LogicNets	471	471,000	654	15,949	1,274
	ULEEN	**740**	**740,000**	**73**	**269**	**538**
JSC	LogicNets	427	427,000	6,222	37,931	**810**
	ULEEN	**773**	**773,000**	**1,222**	**4,774**	5,541
BoT-IoT	ULEEN	920	920,000	156	530	1,079

Table 5.6: FPGA implementation results for ULEEN and LogicNets.

was unchanged. Areas for LogicNets are much smaller than would be expected from the naive parameter sizes due to synthesis-time optimizations. ULEEN is superior in throughput, energy efficiency, and LUT usage on both UNSW-NB15 and JSC, though the JSC ULEEN model uses more LUTs than its LogicNets counterpart.

My co-first-author, S. Nag, performed analyses with additional datasets, models, and hardware implementations for both ULEEN and LogicNets. For more discussion, refer to the published paper [109].

5.7 Summary

This chapter discussed the ULEEN weightless model, which made significant improvements on BTHOWeN in both accuracy and efficiency. ULEEN introduces multi-pass learning using continuous Bloom filters, efficient WNN ensembles, and a RAM node pruning strategy. Compared to iso-accuracy BNNs built on the Xilinx FINN platform, ULEEN achieves a 3.8–9.1× reduction in steady-state energy per inference and a 1.7–7.8× reduction in area-delay product across the MNIST, KWS, and ToyADMOS/car datasets. Overall, ULEEN compares favorably against BNNs and LogicNets, another LUT-based approach to inference on FPGAs, demonstrating its suitability for high-throughput, low-latency edge inference.

Chapter 6: Multilayer Weightless Neural Networks[1]

While ULEEN has excellent latency, throughput, and energy efficiency compared to several leading approaches to edge inference, it is frequently larger in circuit area. This is an obstacle to deploying it on ultra-low-end FPGAs or in low-cost, tiny ASICs. Multilayer WNNs have been previously explored as a strategy to reduce model sizes (§2.1.4.1). By breaking a single large RAM node into a tree or "pyramid" of many small LUTs, memory requirements can be greatly reduced. The advantage of this approach over ULEEN is that it does not require any hashing. However, prior work was unable to find effective strategies for training these models.

While the multi-pass learning rule used for ULEEN calculates gradients with respect to RAM node entries, extending it for a multilayer model requires the ability to define gradients with respect to RAM node *inputs*. Determining a way to do this that was both effective and computationally efficient proved to be a major challenge. This project culminated in the development of the **D**ifferentiable **W**eightless Neural **N**etwork (DWN), which far exceeds the efficiency of ULEEN models in parameter size, inference energy, and circuit area.

This chapter begins by discussing the motivations that lead me to pursue the development of multilayer architectures, and some of my initial approaches to this problem. It then provides an overview of the DWN model, including some optimizations I introduced to make it more accurate and more efficient. Next, it explains the approaches I used to deploy DWNs on FPGAs and on the Elegoo Nano (a low-end microcontroller), and the evaluation results for these devices. My co-first-author [22],

[1] (Co-first author) Alan Tendler Leibel Bacellar, Zachary Susskind, Mauricio Breternitz Jr, Eugene John, Lizy Kurian John, Priscila Machado Vieira Lima, and Felipe M.G. França. Differentiable weightless neural networks. In *Forty-first International Conference on Machine Learning*, 2024. URL `https://openreview.net/forum?id=GBxflz0qdX`

A. Bacellar, made substantial contributions which assisted in the rapid training of accurate DWNs, including the Hamming EFD method (§6.2.4) and learnable mapping (§6.3.3), and performed additional comparisons of DWNs with Tiny Classifier circuits [73] and leading models for tabular data. I provide brief background on his contributions in this work but defer detailed discussion to the ICML paper.

The code for this project is released at `https://github.com/alanbacellar/DWN`.

6.1 Motivation

There are several contributing factors which made multilayer WNN models seem worth revisiting. Replacing Bloom filters with a hierarchy of RAM nodes eliminates the need to evaluate hash functions, and makes false positives impossible. Additionally, rather than having a separate discriminator for each class, RAM nodes in the intermediate layers of a model can be shared, reducing its parameter size.

6.1.1 False Positive Rates for Bloom Filters

Bloom Filters enable the compression of WNN models such as ULEEN while maintaining the ability of RAM nodes to learn Boolean functions with arbitrary minterms. However, they are prone to producing false positives, with the frequency of this increasing sharply as more items are stored. Figure 6.1 shows the maximum number of items that can be stored in a Bloom filter before the false positive rate (FPR) rises above 10% [63]. For a WNN, this is equivalent to the maximum number of minterms in the Boolean function represented by the filter. This cap increases linearly with the size of the filter, but remains relatively low. For instance, a Bloom filter with two hash functions and 2048 entries can hold just 389 items. Using four hash functions provides a small ($\sim$8.5%) increase in effective capacity but introduces additional implementation complexity.

This limited capacity is a potential impediment to generalization. Prior work

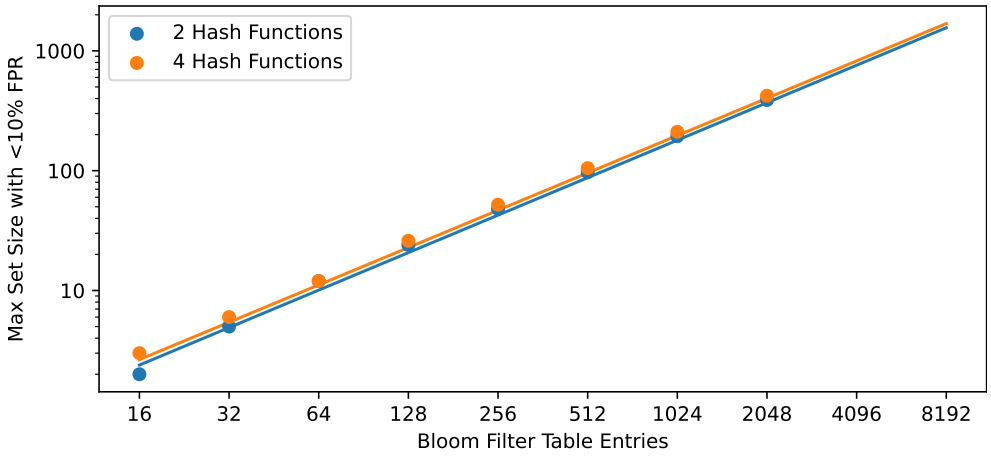


Figure 6.1: The number of items that can be stored in a Bloom filter with given table and hash function set sizes without exceeding a 10% false positive rate. Capacity increases with table size and to some extent with the number of hash functions, but is limited even for large Bloom filters

has demonstrated that to maximize accuracy, similar input patterns should often produce identical outputs [11]. In other words, if a specific input to a RAM node results in an output of '1', other inputs which are nearby (in terms of Hamming distance) should likely also produce '1'. Consider a Bloom filter with 36 inputs, 2048 entries, and two hash functions. If patterns stored in the filter form hyperspheres with radius 1, each consisting of a single centroid pattern and its 36 Hamming-adjacent neighbors, then the effective capacity of the filter is reduced by a factor of 37, meaning just 10 such hyperspheres can be represented within the given false positive bound.

6.1.2 Elimination of Hash Computation

Hash computation introduces a very large amount of area overhead for ULEEN models. For instance, hierarchical Vivado utilization reports indicate that 31,278 of the 49,445 LUTs in the ULN-M model (63%) are consumed by the hash functional units. By contrast, an *unoptimized* implementation of the tables within the Bloom fil-

ters would only require 12,880 LUTs.[2] While it is possible that the reported hardware area for hashing is somewhat inflated by cross-hierarchy optimizations, this example demonstrates that hashing can be a bottleneck for creating more area-efficient WNNs.

6.1.3 LUT Sharing

Like WiSARD, ULEEN uses separate discriminators, composed of independent sets of RAM nodes, for each output class. This means that if a specific input pattern is relevant for multiple classes, it must be learned in multiple places. As shown in Figure 6.2a, previous multilayer weightless models were based on RAM node pyramids. In these models, not only does each discriminator have a separate set of pyramids, but each pyramid also has separate sets of RAM nodes. Therefore, these pyramids operate completely independently until the popcount at the end of the discriminator. This organization can result in a great deal of redundancy in the data that is stored in separate pyramids, but was required for the learning rules that were used for these models.

DWN models are trained using a gradient-based learning rule that does not require this restrictive organization. Therefore, instead of pyramids, DWNs use multiple layers of sparsely but arbitrarily interconnected LUTs, shown in Figure 6.2b. With this approach, most layers of the model consist of randomly-connected sets of lookup tables arranged in a feedforward topology. The last layer of the model is usually still broken into separate discriminators so that popcounts can be used to compute per-class response scores.

[2]This estimate comes from dividing the parameter size (in bits) of the model by 64, the number of entries in a FPGA LUT-6. Some additional logic is needed to combine partial results for filters with more than 64 entries, but this can be accomplished using 2:1 MUXes.

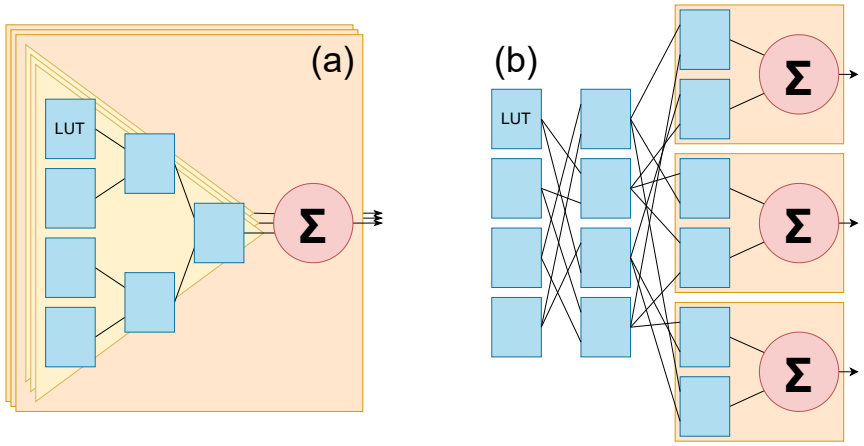


Figure 6.2: (a) Prior multilayer WNNs built discriminators out of pyramids of lookup tables. This can potentially result in redundancy, where the same behaviors are learned in multiple places in the model. (b) By contrast, DWNs share LUTs between classes in all but the final layer. This approach reduces model parameter sizes, but requires a more advanced learning rule.

6.2 Learning Rules for Multilayer WNNs

Identifying a gradient-based learning rule for multilayer WNNs proved to be nontrivial, and involved the exploration of many different approaches. The performance of some of the approaches that were explored is shown in Table 6.1. My initial approaches used a multilinear polynomial "Finite Difference" method that was efficient to compute but struggled to match the accuracy of ULEEN. Subsequently, I tried approaches that used relaxed approximations of LUTs during training, including sparsely-connected dense subnetworks and an adaptation of the alpha-blending BNN learning rule. These improved accuracy, but were slow and memory-intensive for larger models. Ultimately, the Extended Finite Difference (EFD) learning rule, which is both fast and efficient, and the Polynomial EFD variant, which has more stable gradient behavior for deeper models, were used to create DWNs.

Learning Rule	Test Acc.%	Parameter Size (KiB)	Train Time/ Epoch (s)
ULEEN (Baseline)	97.79%	100.6	89.7
Finite Difference	96.59%	53.1	2.9
FD + Gradient Spreading	96.99%	53.1	3.0
Subnetwork Equivalents	97.25%	220.5	32.9
Alpha-Blending	98.35%	23.4	26.3
EFD	98.04%	23.4	5.0
Polynomial EFD	98.31%	23.4	5.0

Table 6.1: Performance of different training strategies for multilayer WNNs on the MNIST dataset. The "EFD" and "Polynomial EFD" rules strike the best balance between speed, memory efficiency, and accuracy, and were therefore chosen for DWNs. All models were trained using an NVIDIA A100 GPU.

6.2.1 Finite Difference Learning Rule

Any Boolean function can be expressed as a multilinear polynomial [116], which is a useful relaxation for defining gradients. To begin, for a binary vector $x \in \{-1, 1\}^n$, we can write the Kronecker delta function δ_{ax} (i.e., the function that returns 1 if $a{=}x$ and 0 otherwise) as:

$$\delta_{ax} = \prod_{i=1}^{n} \frac{1 + a_i x_i}{2} \tag{6.1}$$

Each LUT in a DWN model represents a function $f : \{-1, 1\}^n \longrightarrow \{-1, 1\}$. As with ULEEN, during training, table entries are treated as floating-point values between -1 and 1 and binarized using the straight-through estimator. This means that for a LUT T, $f(x) = \text{STE}(T[x])$. We can therefore expand f as:

$$f(x) = \sum_{a \in \{-1,1\}^n} f(a)\delta_{ax} = \text{STE}\left(\sum_{a \in \{-1,1\}^n} T[a]\delta_{ax} \right) \tag{6.2}$$

The formulation in Equation 6.2 allows gradients to be defined with respect to both T and x. The gradient with respect to an entry $T[a]$ is simply equal to the

delta function:

$$\frac{\partial f}{\partial T[a]} = \text{STE}'(T[x])\,\delta_{ax} = \delta_{ax} \tag{6.3}$$

Finding the gradient with respect to x_j, the jth element of x, is trickier. By moving the gradient inside the summation and breaking out the $\frac{1+a_j x_j}{2}$ term from the product, it can be simplified to a function of $x_{[j\to+1]}$ and $x_{[j\to-1]}$, the value of x when j is forced to 1 or -1, respectively:

$$\begin{aligned}
\frac{\partial f}{\partial x_j} &= \sum_{a\in\{-1,1\}^n} T[a]\,\frac{\partial}{\partial x_j}\prod_{i=1}^{n}\frac{1+a_i x_i}{2} \\
&= \sum_{a\in\{-1,1\}^n} T[a]\,\frac{a_j}{2}\prod_{i=1}^{j-1}\frac{1+a_i x_i}{2}\prod_{i=j+1}^{n}\frac{1+a_i x_i}{2} \\
&= \frac{1}{2}\left(T[x_{[j\to+1]}] - T[x_{[j\to-1]}]\right)
\end{aligned} \tag{6.4}$$

I refer to this as the Finite Difference learning rule because of the strong similarity of the gradient with respect to x_j to the finite difference methods used to approximately solve differential equations [119]. Note, however, that Equation 6.4 gives an exact gradient for this construction of f.

6.2.1.1 Gradient Spreading

To improve the generalization of the models trained with the finite difference learning rule, I explored using a gradient spreading technique which distributes a fraction of the gradients of each table entry to other entries at close Hamming distances. Given gradient "smearing matrix" S with $S_{ij} = e^{-d_H(i,j)}$ (where d_H is the Hamming distance metric), the adjusted table gradient $\frac{\partial f}{\partial T_S}$ is given by:

$$\frac{\partial f}{\partial T_S} = S\frac{\partial f}{\partial T} = \begin{bmatrix} 1 & e^{-1} & e^{-1} & e^{-2} & \cdots & e^{-n} \\ e^{-1} & 1 & e^{-2} & e^{-1} & \cdots & e^{1-n} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ e^{-n} & e^{1-n} & e^{1-n} & e^{2-n} & \cdots & 1 \end{bmatrix} \begin{bmatrix} \delta_{\{-1,-1,\ldots,-1\}x} \\ \delta_{\{+1,-1,\ldots,-1\}x} \\ \vdots \\ \delta_{\{+1,+1,\ldots,+1\}x} \end{bmatrix} \tag{6.5}$$

6.2.2 LUTs as Subnetwork Equivalents

LogicNets [146] proposed the idea of using lookup tables to replace neurons in a sparsely-connected DNN: by training a model with low-precision activations but full-precision weights, then explicitly enumerating the output of each neuron for each combination of its input bits, the trained model could be effectively tabularized. However, by only replacing individual neurons, LogicNets foregoes one of the key strengths of lookup tables: their ability to represent complex nonlinear behaviors. Therefore, I explored using LUTs to instead replace small trained subnetworks.

As shown in Figure 6.3, at training time, this strategy uses a model composed of many fully-connected DNNs, each with n binary inputs, one binary output, and a configurable number of internal layers which use full-precision weights and activations. These DNNs are then connected to form layers. After training, DNNs can be converted directly into LUTs by enumerating their responses to all possible inputs. In general, this approach is somewhat more accurate than the finite difference learning rule, but it is far slower and has a very large GPU memory footprint during training due to the need to track gradient information for each of the subnetworks. A related method for training sparsely-connected DNNs was independently discovered by NeuraLUT [18].

6.2.3 Alpha-Blending

Alpha-blending [100] (§2.2.2.3) is a methodology for training BNNs which does not use the straight-through estimator, instead performing an affine combination of quantized and unquantized weights using a non-learned parameter α. I extended this methodology to learn the LUT values in WNNs.

When training with the alpha-blending strategy and $\alpha<1$, the LUTs in the first layer of the model can output any value in the range $[-1, 1]$. Therefore, the inputs to subsequent layers are not binary, and Equation 6.1 no longer behaves as a Kronecker delta function. However, as Equation 6.6 shows, the outputs of the

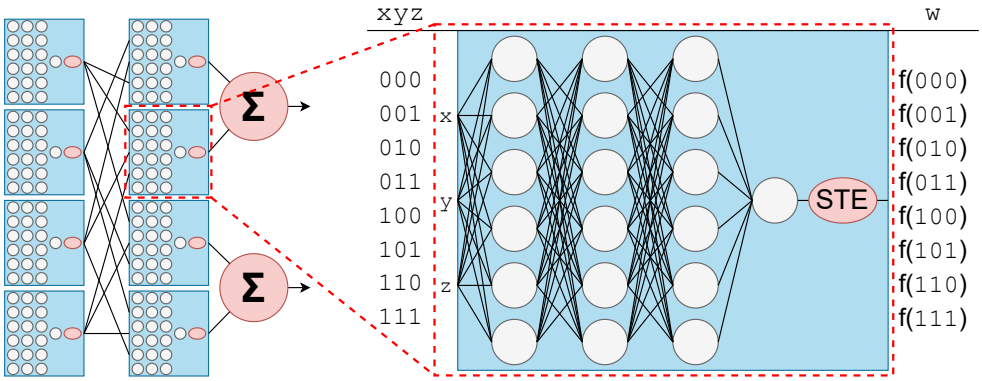


Figure 6.3: During training, LUTs can be represented as DNNs with dense, full-precision internal connectivity but sparse, binary external connectivity. This allows for straightforward training using backpropagation, followed by post-training tabularization.

functional units in these layers are still bounded between -1 and 1. Hence, these units can be viewed as continuous relaxations of LUTs which represent equations of the form $f : [-1, 1]^n \longrightarrow [-1, 1]$.

$$
\begin{aligned}
|f(x)| &= \left| \sum_{a \in \{-1,1\}^n} \left(\alpha \operatorname{sign}\left(T[a]\right) + (1-\alpha)\ T[a]\right) \prod_{i=1}^{n} \frac{1 + a_i x_i}{2} \right| \\
&\leq \sum_{a \in \{-1,1\}^n} \left| \left(\alpha \operatorname{sign}\left(T[a]\right) + (1-\alpha)\ T[a]\right) \prod_{i=1}^{n} \frac{1 + a_i x_i}{2} \right| \\
&\leq \sum_{a \in \{-1,1\}^n} \prod_{i=1}^{n} \frac{1 + a_i x_i}{2} \quad \text{(since } a_i x_i \geq -1 \text{ for } x_i \in [-1, 1]\text{)} \qquad (6.6) \\
&= \sum_{a \in \{-1,1\}^{n-1}} \frac{1 - x_n}{2} \prod_{i=1}^{n-1} \frac{1 + a_i x_i}{2} + \sum_{a \in \{-1,1\}^{n-1}} \frac{1 + x_n}{2} \prod_{i=1}^{n-1} \frac{1 + a_i x_i}{2} \\
&= \sum_{a \in \{-1,1\}^{n-1}} \prod_{i=1}^{n-1} \frac{1 + a_i x_i}{2} = \sum_{a \in \{-1,1\}^{n-2}} \prod_{i=1}^{n-2} \frac{1 + a_i x_i}{2} = \ldots = \mathbf{1}
\end{aligned}
$$

The alpha-blending learning rule provides excellent accuracy, but like the previous method, it proved difficult to scale to larger models due to its slow execution speed and large memory footprint.

6.2.4 Extended Finite Difference

The Extended Finite Difference (EFD) learning rule for multilayer WNNs was originally proposed by A. Bacellar [22]. EFD uses a gradient spreading approach based on Hamming distance, but applies it to the gradients of the inputs as in Equation 6.7, rather than applying it to the table entries as my original Finite Difference method did. EFD allows a network that is trained using EFD to consider table entries that are non-Hamming-adjacent when determining input gradients. This is critical since multiple inputs could potentially change during the same training pass, causing an abrupt shift to a more distant table position.

$$\frac{\partial f}{\partial x_j} = \sum_{a \in \{-1,\,1\}^n} \frac{a_j T[a]}{d_H(a,\,x) + \frac{a_j x_j - 1}{2} + 1} \tag{6.7}$$

6.2.4.1 Polynomial EFD

While the EFD algorithm as specified in Equation 6.7 is effective, it is prone to causing exploding gradients with deeper models. For instance, as shown in Figure 6.4, when training a 10-layer DWN with this approach, the average parameter gradient in the first layer of the model exceeds the average in the last layer by a factor of more than 10^{12}. On the other hand, the alpha-blending technique, which can also consider table entries that are further away due to its use of continuous-valued entries in x, does not exhibit this behavior.

To help stabilize gradients, I created a variant of EFD which leverages interpolating polynomials similarly to alpha-blending, shown in Equation 6.8. With this approach, the values of x are multiplied by a "hardness" coefficient $h \in [0,\,1]$. $h = 1$ gives behavior identical to the conventional Finite Difference learning rule, while $h = 0$ gives equal weight to all LUT entries. I have empirically found that $h = 0.5$ works well.

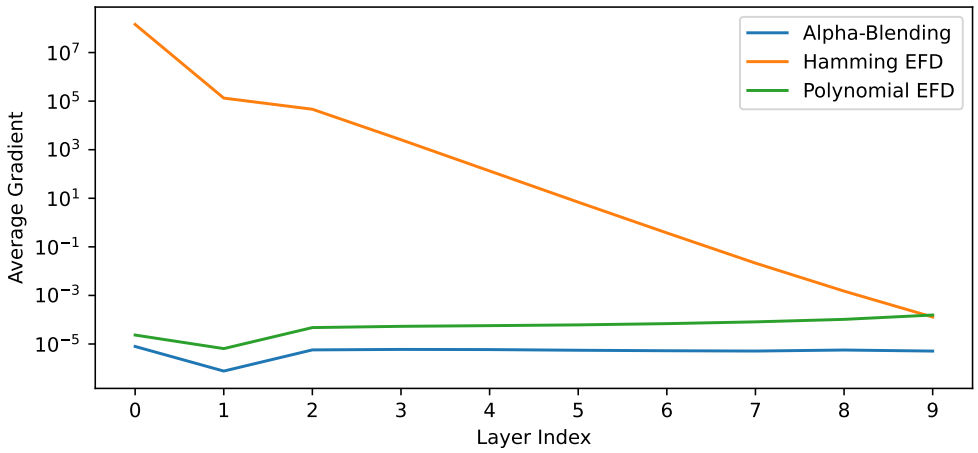


Figure 6.4: Average gradients (taken across all batches on the tenth epoch of training) for ten-layer DWNs using the alpha-blending, Hamming EFD, and polynomial EFD strategies. While the Hamming EFD strategy is quick to train, it encounters instability with deeper model architectures. The polynomial EFD technique resolves this issue while maintaining speed and accuracy.

$$\frac{\partial f}{\partial x_j} = \sum_{a \in \{-1,1\}^n} T[a] \, \frac{a_j}{2} \prod_{i=1}^{j-1} \frac{1 + h a_i x_i}{2} \prod_{i=j+1}^{n} \frac{1 + h a_i x_i}{2} \tag{6.8}$$

This polynomial-based approach gives much more stable gradients for deeper models than Hamming distance-based EFD. However, like in alpha-blending, this function is expensive to evaluate explicitly. *Unlike* alpha-blending, x is guaranteed to be binary when using EFD, which enables optimizations during the forward and backward passes. During the forward pass of the model, a table lookup can be used instead of evaluating the interpolating polynomial. During the backwards pass, observe that T and x are the only variables in Equation 6.8, and x can only assume 2^n possible unique values. Therefore, I precompute a three-dimensional tensor of partial results for all possible values of j and x, a row from which can be selected and multiplied with the contents of the LUT T, greatly reducing the time to calculate this gradient. Table 6.1 shows that this optimization allows this polynomial EFD strategy to run as quickly as the original EFD; additionally, its accuracy is generally equal or slightly better.

6.3 Optimizing DWNs

Beyond the choice of learning rule, I explored several ways to further improve the accuracy and efficiency of DWNs. These include regularization strategies and a way of performing classification that eliminates the need for separate RAM nodes for each class in the last layer.

6.3.1 Regularization Strategies

Like ULEEN, DWNs can be prone to overfitting. In fact, even very small DWNs can sometimes perfectly memorize their training data. Unfortunately, conventional DNN regularization techniques can not be applied directly to DWNs. For instance, since only the sign of a table entry is relevant for address calculation during inference, using L1 or L2 regularization to push entries towards 0 during training results in instability as values repeatedly flip from positive to negative. Therefore, I developed two regularization strategies to aid in generalization.

6.3.1.1 Spectral Normalization

Spectral regularization is a normalization technique which penalizes the representation of complex behaviors in LUTs.For an n-input pseudo-Boolean function $f : \{-1, 1\}^n \to \mathbb{R}$, I define the L2 spectral norm of f as:

$$\frac{1}{2^n}\Big\|\Big\{\sum_{x\in\{-1,1\}^n} f(x)\Big(\prod_{i\in S} x_i\Big) \;\Big|\; S \in [n]\Big\}\Big\|_2$$

Note that this is simply the L2 norm of the Fourier coefficients of f [116]. Additionally, since all terms except $f(x)$ are constant, it is possible to precompute a coefficient matrix $\mathcal{C} \in \mathbb{R}^{2^n \times 2^n}$ to simplify evaluation of the spectral norm at runtime. In particular, for a layer of u LUTs with n inputs each and data matrix $L \in [-1, 1]^{u\times 2^n}$, the spectral norm can be written as:

$$\text{specnorm}(L) = \|L\mathcal{C}\|_2,\ \mathcal{C}_{ij} := \frac{1}{2^n}\prod_{a\in\{b \mid i_b=1\}} j_a$$

The effect of spectral regularization is to increase the resiliency of the model to perturbations of single inputs. For instance, if an entry in a RAM node is never accessed during training, but all locations at a Hamming distance of 1 away hold the same value, then the unaccessed location should most likely hold this value as well.

I find that spectral normalization usually works best for small models—for DWNs with more RAM nodes, it has minimal effect, and sometimes can even slightly harm accuracy. One possible explanation for this behavior is that small models may need for LUTs to act as "generalists" which are sensitive to a large number of behaviors, while larger models can afford for LUTs to have more specialized responses.

6.3.1.2 Random Bitflip Injection

A simple but effective form of regularization is to corrupt a small ($\sim$3%) portion of the inputs to a layer of LUTs by introducing random bitflips. Doing so helps to limit the dependence of the model on any single bit holding a specific value. This form of regularization is conceptually similar to the dropout [135] technique used for training DNNs. In fact, dropout can also be applied to the outputs of the last layer of LUTs, before summation is performed. However, I have found that while random bitflips provide a small but consistent increase in accuracy, especially for larger models, the impact of traditional dropout is less clear.

6.3.2 Ternary Summation

Rather than dedicating a separate set of LUTs to each class in the last layer of a DWN, the outputs of a shared set of LUTs can be passed into a ternary-weighted linear classifier. This means that each LUT can have a learned positive (w=+1), neutral (w=0), or negative (w=−1) impact on the response score for each class. Reusing LUTs in this way makes it possible to reduce the size of the last layer of the model. Strictly speaking, the addition of the linear classifier makes this no longer a weightless model, but the number of weights is negligible when compared to both the

number of non-weight parameters as well as the number of weights in a comparable BNN, as shown in Table 6.2.

Model	Weights	Non-Weight Parameters	Accuracy
FINN (BNN)	930,816 (114 KiB)	0	97.7%
DWN	0	96,000 (11.7 KiB)	97.8%
Ternary DWN	3,000 (0.58 KiB)	83,200 (10.2 KiB)	97.9%

Table 6.2: Impact of ternary summation on DWNs.

In a hardware accelerator for inference, the ternary-weighted layer can be implemented by performing two popcounts for each class, corresponding to LUTs with weights $+1$ and -1, respectively, and then taking the difference. However, the area overhead for the additional popcount is usually larger than the savings from sharing LUTs. Therefore, the ternary summation strategy is better suited for cases where minimizing the *paramter size* of the model is more important, such as a memory-limited microcontroller implementation.

6.3.3 Learnable Mapping

WNNs conventionally use a random mapping of inputs to LUTs. A. Bacellar [22] developed an alternative strategy which instead allows for the connectivity of LUTs to be learned during the training process. This strategy involves maintaining a matrix of weights which indicate the affinity of each LUT input for each model input. During the forward training pass, the indices of the highest-valued weights are used to select the inputs to LUTs; during the backward pass, an approach based on the softmax of connection weights is used to compute weight gradients. After training, the inputs corresponding to the highest-valued weights are chosen to form a static mapping, so this technique has no inference overhead.

Using learnable mappings for multiple layers of LUTs in a model can degrade accuracy, likely due to the cascading disruptions that can occur when several levels

of interconnect are altered simultaneously. Therefore, learnable mapping is normally only used before the first layer of LUTs in DWNs.

6.3.4 Other Optimizations Explored

In the pursuit of optimizing DWNs, I explored several other approaches that were ultimately unsuccessful. Some of these approaches merit brief discussions, particularly since they may help to inform future research in this domain.

- **Stochastic Binarization:** Instead of using the signs of table entries to determine LUT outputs, I explored treating table entries as Bernoulli random variables with $p = (T[x] + 1)/2$ during training, which is similar to the approach used for MPLNs. This had no discernible impact on accuracy, which mirrors results that were observed for BNNs [43], but it increased training times due to the need for random sampling.

- **Skeletal Mapping:** The Lottery Ticket Hypothesis [57] states that DNNs contain sparse subnetworks which, when trained from initialization, can reach accuracy similar to the original model. In other words, the connections in a DNN that will ultimately be important are determined at initialization rather than during training. Several works have explored methods to identify these critical connections *a priori* and train pre-pruned, "skeletonized" models. I adapted one of these techniques, FORCE [46], to initialize WNNs. My approach uses subnetwork equivalents similar to §6.2.2 but connected to all outputs in the previous layer, which are then pruned down to the desired number of inputs (i.e., n) using the modified FORCE strategy, and lastly converted into LUTs which are trained using polynomial EFD. The advantage of this approach to connecting LUTs is that it only needs to be done once at initialization, allowing it to train more quickly than learnable mapping, which requires continual updates to a large weight matrix. However, while skeletal mapping performs better than random initialization, learnable mapping remains significantly more accurate.

This concurs with recent work [88] which argues that initialization-time pruning must inherently result in greater loss of accuracy.

- **Ensembling and Mixtures of Experts:** I explored both additive submodel ensembles of DWNs, similar to what I used for ULEEN, and a modified, weightless version of a mixture-of-experts (MoE) algorithm [131] which uses a gating network to select a subset of "expert" subnetworks to run. Additive ensembles did not have any clear advantage over monolithic models of the same total size, which contradicts my observations with ULEEN. This suggests that the ensembling strategies which are effective for multilayer WNNs are different than those for single-layer models. The MoE degraded accuracy unless the individual experts were made very large (which defeats its purpose); it is possible that a different approach to MoE would give better results, or that the minimum model size needed for MoE to be beneficial is larger than what I am targeting with DWNs.

- **Learnable Thermometer Thresholds:** Since all layers of LUTs, including the first, can backpropagate gradients to their inputs, it seemed reasonable to try using gradient information to update thermometer encoding thresholds. I developed a differentiable input binarization scheme that divided an input feature by its standard deviation, subtracted a learned threshold parameter, and took the sign of the hyperbolic tangent of the result (so that input values far from the threshold would have less impact on its gradient). This improved accuracy when using random mapping, but caused degradation when using learnable mapping. Notably, even if I trained a model with learnable mapping and fixed thresholds, replaced the final learnable mapping with an equivalent fixed mapping, and then fine-tuned the thresholds, accuracy still stayed the same or decreased.

6.4 DWN Software Model

Training of DWNs is performed using learnable mapping and either the Hamming or polynomial EFD learning rules, as described previously. Instead of the Gaussian thermometer encoding that I used for BTHOWeN and ULEEN, DWNs use the recently developed Distributive thermometer encoding [21]. I train using the Adam optimizer for 100 epochs, with a starting learning rate of 1e-3 and a factor of 10 reduction after every 30 epochs. I also use a softmax temperature derived from the method used in DiffLogicNet [120], with $\tau = 10 \cdot \sqrt{10}^{\log_{10}\left(\frac{m}{300}\right)}$ for a model with m RAM nodes per class in the last layer.

After training, learnable mapping is converted into a fixed interconnect by finding the indices of the largest weights for each LUT input and forming connections to the corresponding model inputs. Lookup tables are binarized using the sign function, as in ULEEN. The result of this process is a multilayer model which is entirely free of arithmetic except for the popcounts after the last LUT layer. An extremely simple DWN for the Iris [56] dataset is shown in Figure 6.5 in its post-training form.

6.5 DWN Inference Accelerator

I deploy inference accelerators for DWN models on the Xilinx Zynq Z-7045, the same device that was used for FINN [145] and ULEEN [139]. I also reuse the input compression scheme that I developed for ULEEN, and target the same clock frequency of 200 MHz. The elimination of time-multiplexed hashing, which required the propagation of stall signals backwards through the accelerator, removes the need for double-buffering in the bus deserialization and decompression units. This significantly reduces the number of flip-flops in the design and provides some savings in LUT counts due to simplified buffering logic. Figure 6.6 gives a high-level overview of my DWN accelerator design. Xilinx FPGAs are largely composed of configurable logic blocks (CLBs), which are in turn composed of six-input lookup tables (LUT-6s),

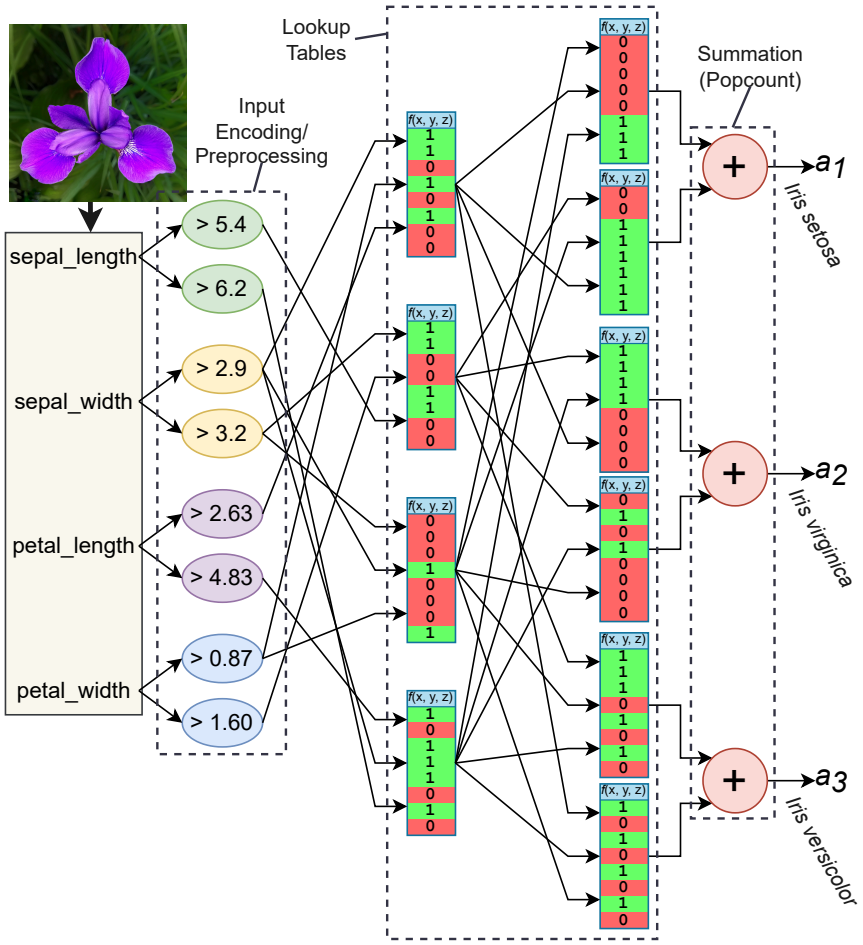


Figure 6.5: A small, trained DWN for the Iris dataset, demonstrating how the model is composed out of multiple layers directly chained lookup tables, with the outputs of one layer forming the address bits to the next.

flip-flops, and miscellaneous interconnect and muxing logic [159].[3] The Z-7045 provides a total of 218,600 LUT-6s, each of which can be used to represent a six-input RAM node. Therefore, using n=6 is a natural choice when developing DWNs with the objective of deploying them on Xilinx FPGAs, as it allows them to make efficient use of hardware resources.

[3]As shown in Figure 6.6, a LUT-6 can also function as two LUT-5s, but this is only possible if both LUTs have identical inputs.

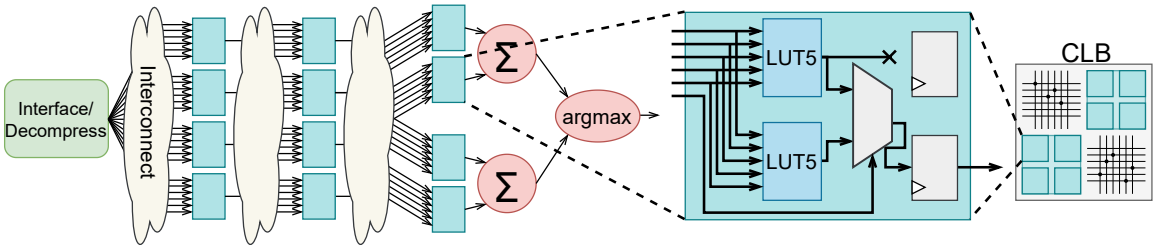


Figure 6.6: Implementation of a DWN on an FPGA. Each hardware LUT-6 (internally subdivided into two LUT-5s and a 2:1 MUX) can be used to implement a single six-input RAM node. Registers are inserted between layers of LUTs to buffer outputs and improve device timing.

6.6 DWNs on Microcontrollers

I also explore deployment of DWNs on the Elegoo Nano, a tiny microcontroller (MCU) with only 2 KB of RAM and 30 KB of flash storage (see §3.2.2). Running BTHOWeN and ULEEN models on this device is impractical: they are simply too large, particularly since the input mapping information for RAM nodes (i.e., the indicies of each input to each RAM node into the array of outputs from the previous layer) must also be stored in device memory. BNNs are also too large; for instance, even the SFC FINN model has a parameter size of 40.8 KiB. Therefore, the options for machine learning that are viable on this class of device are very limited. However, with careful optimization, it is possible for me to fit DWNs within these constraints. I use two strategies: an approach which uses aggressive bit-packing of LUT data and mapping information in order to fit larger models at the cost of speed, and an implementation which foregoes most bit-packing in order to attain the fastest possible execution speed, but is consequently restricted to smaller, less accurate models.

6.6.1 Bit-packed Implementation

Figure 6.7 shows the memory layout for a packed DWN inference model on the Elegoo Nano MCU. To reduce storage space, LUT data is packed bitwise, and input mapping information is compressed by using 10 or 12-bit indices when possible.

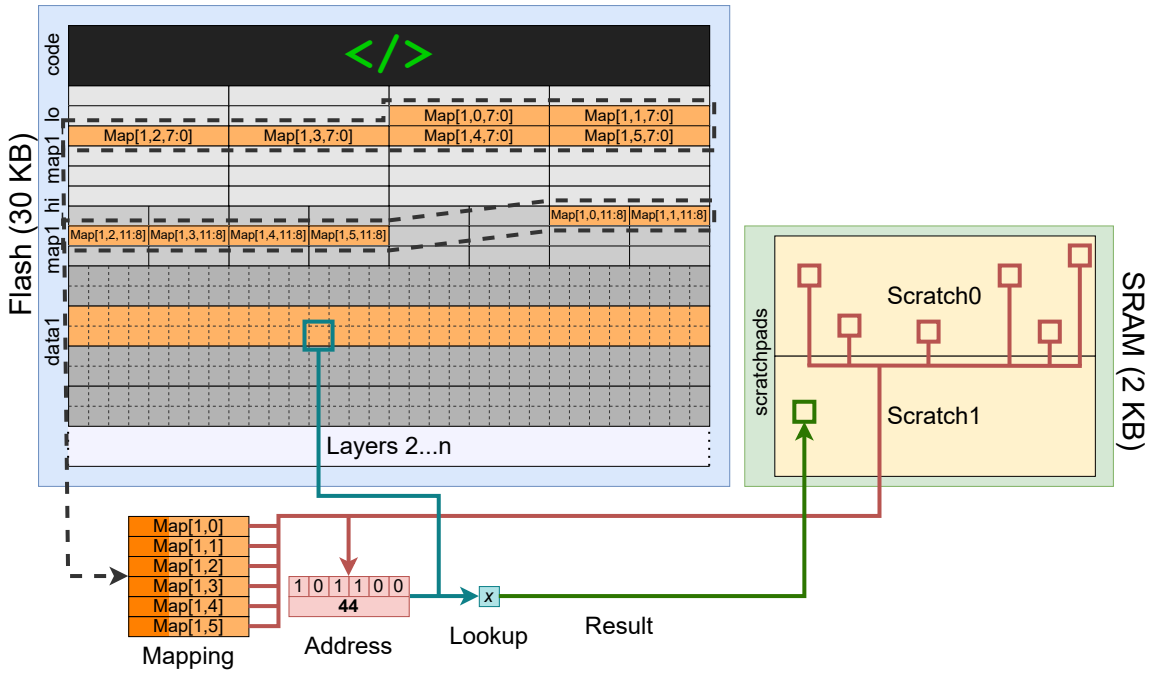


Figure 6.7: An overview of the data layout of a bit-packed DWN model implemented on the Elegoo Nano MCU.

For instance, a LUT-6 which comes after a layer of 1000 LUTs can be represented in 15.5 bytes of flash storage: 8 bytes ($\frac{2^6}{8}$) for its data array, and 7.5 bytes ($\frac{6 \cdot 10}{8}$) to store the indices of the LUTs on the previous layer that produce each input.

My model conversion script automatically determines the minimum possible bit width for mapping indices in each layer (possible choices are 2, 4, 8, 10, 12, or 16 bits). For a non-power-of-two bit width (10 or 12 bits), the map is further subdivided into `map_lo`, which stores the low 8 bits, and `map_hi`, which stores the remaining high bits. While Figure 6.7 only shows mappings and data for one layer, these data structures are repeated for each layer in the model. SRAM is divided into two large scratchpads, which are used alternately between layers. Layer i of a model reads activations for the previous layer from scratchpad ($(i-1) \bmod 2$) and writes to scratchpad ($i \bmod 2$).

Unpacking mapping and data arrays during inference is a relatively slow process, particularly since the ATmega328P used in the Nano has poor ISA support for bitwise operations. For instance, its logical shift instructions can only shift by one bit position, meaning it takes many instructions to perform larger shifts. By optimizing the data layout of the model, I am able to somewhat reduce the need for multi-bit shifts—for instance, if every bit in a byte should be read from or written to sequentially, organizing this process so that it goes from the least significant to the most significant bit is much faster than going from MSB to LSB. Additionally, using bit packing reduces the data footprint of the model in flash by $\sim$4.5$\times$ and in SRAM by 8$\times$, which significantly increases the complexity of the models that can be implemented.

6.6.2 Unpacked Implementation

The unpacked model does not perform bit-packing for LUT data or mapping information, ternary sum weights, and activations, which makes it much faster but less memory-efficient. Layers are restricted to having at most 256 LUTs, which means that mapping indices can be represented using 8-bit integers for all layers except the first (which requires 16-bit integers for datasets with more than 256 inputs).

6.7 Evaluation Methodology

I compare my accelerators for DWNs against ULEEN, fully-connected FINN models, and DiffLogicNet [120]. DiffLogicNet constructs multilayer feed-forward networks out of learned two-input logic gates. It operates by assigning weights for each of the 16 (2^{2^2}) Boolean functions of two inputs to each gate, which are updated during training. The softmax of these weights is used to determine the probability of a gate implementing any particular Boolean function. After training, the Boolean function with the largest weight is chosen for each gate. While the authors of DiffLogicNet do not describe their model as a weightless neural network, it es-

sentially is a multilayer WNN with $n = 2$. However, this approach scales poorly to larger values of n; for instance, a DiffLogicNet model with n=6 would require $2^{2^6} =$ 18,446,744,073,709,551,616 trained parameters per LUT, while DWNs require $2^6 =$ 64.

After training and binarization, DiffLogicNet is similar enough to DWNs with n=2 that it can be implemented using my accelerator framework with minimal modification. Therefore, I report hardware results for DiffLogicNet with n=2 and DWNs with n=2 and n=6. I use the MNIST, KWS, and ToyADMOS datasets from ULEEN, plus FashionMNIST (which has the same dimensions as MNIST but is considerably more difficult) and CIFAR-10.

As discussed previously, most approaches to machine learning require too much memory to run on the Nano MCU, even for inference. XGBoost [39] is a widely-used tree boosting system notable for its ability to achieve high accuracies with tiny parameter sizes. Therefore, it is a good choice for this domain. I use the MicroML [129] library to convert trained XGBoost models into a more efficient form which is intended for inference on MCUs. In order to fit entire samples into the Nano's SRAM, I quantize input features to 8-bit integers by linearly scaling the minimum and maximum values from 0 to 255. I did not observe significant any significant impact on accuracy from performing this transformation. All XGBoost models have a max tree depth of 3, with forest size maximized for the board's memory. I found that this gave better results than the default max depth of 6, which required extremely small forests to fit in the device. The parameter sizes of these XGBoost models are generally quite small, but their complex control flow means that the source code is large, even after compiler optimization. For the Arduino comparison, I also include results for three tabular datasets, *phoneme*, *skin-seg*, and *higgs*, since XGBoost is known to excel on tabular data. A. Bacellar assisted in training some DWN and DiffLogicNet models for the FPGA comparison. The FPGA and MCU implementations of DWNs and the data collection on these devices are wholly my own work.

6.8 Results

6.8.1 Selected Models

Table 6.3 summarizes the configuration parameters of the DWN models I used in my evaluations, including those used for FPGA and MCU deployments. For the FPGA implementation of MNIST, I used two different model sizes in order to provide iso-accuracy comparisons with both FINN and DiffLogicNet. For ToyADMOS, I also used two model sizes since I found that a tiny DWN was able to match prior work, and a somewhat larger model was significantly more accurate.

Most tabular datasets have only a small number of unique inputs, and seem to benefit much more from high thermometer resolution than other workloads. I use 128 thermometer bits for these on the bit-packed MCU implementation, and 255 on the unpacked implementation. The unpacked implementation does not attempt to fit multiple inputs into a byte, so there is no real disadvantage to using 255 encoding bits per input during inference. I only use 32 encoding bits for the other four datasets to reduce training times, since they see very little accuracy benefit from higher resolutions.

6.8.2 FPGA Implementation Results

Table 6.4 shows the results of my comparisons of DWNs against FINN, ULEEN, and DiffLogicNet on the Z-7045 FPGA. For all datasets except CIFAR-10, the DWN models are smaller, faster, and more energy-efficient than all prior work, with comparable or better accuracy. In particular, latency, throughput, energy per sample, and hardware area (in terms of FPGA LUTs) are improved by geometric averages of (20.7, 12.3, 121.6, 11.7)×, respectively, versus FINN, and (3.3, 2.3, 19.0, 22.7)×, respectively, versus ULEEN. This translates to a 2522× improvement in average energy-delay product versus FINN and a 63× improvement versus ULEEN. Unlike the other models listed, FINN supports convolution. This gives it vastly superior accuracy on the CIFAR-10 dataset, albeit at a hefty cost to speed and energy effi-

Target Platform	Dataset	LUT Inputs	LUT Layer Sizes	Encoding Bits	Ternary Sum	Parameter Size (KiB)
FPGA	MNIST	n=2	[6000, 6000]	3	✗	5.9
		n=6	[1000, 500]	1	✗	11.7
		n=6	[2000, 1000]	3	✗	23.4
	FashionMNIST	n=2	[8000, 8000]	7	✗	7.8
		n=6	[2000, 2000]	7	✗	31.3
	KWS	n=2	[3000, 3000]	8	✗	2.9
		n=6	[1600]	8	✗	12.5
	ToyADMOS	n=2	[900, 900]	2	✗	0.9
		n=6	[400]	2	✗	3.1
		n=6	[1800, 1800]	3	✗	28.1
	CIFAR-10	n=2	[24000, 24000]	10	✗	23.4
		n=6	[8000]	10	✗	62.5
Packed MCU	MNIST	n=6	[1000, 500]	3	✓	12.7
	FashionMNIST	n=6	[1000, 500]	3	✓	12.7
	KWS	n=6	[1000, 500]	3	✓	12.8
	ToyADMOS	n=6	[1000, 500]	3	✓	11.9
	phoneme	n=6	[1000, 500]	128	✓	11.9
	skin-seg	n=6	[1000, 500]	128	✓	11.9
	higgs	n=6	[1000, 500]	128	✓	11.9
Unpacked MCU	MNIST	n=6	[220, 110]	32	✓	2.8
	FashionMNIST	n=6	[220, 110]	32	✓	2.8
	KWS	n=6	[220, 110]	32	✓	2.8
	ToyADMOS	n=6	[220, 110]	32	✓	2.6
	phoneme	n=6	[80, 40]	255	✓	1.0
	skin-seg	n=6	[80, 40]	255	✓	1.0
	higgs	n=6	[90, 90]	255	✓	1.4

Table 6.3: Selected DWN model configurations for FPGA and microcontroller deployment.

ciency. As with ULEEN (§5.4.1.1), the lack of convolution limits DWNs' effectiveness on datasets with high degrees of positional independence.

Several models in Table 6.4 could not be implemented on the target FPGA (indicated by a '$*$' next to their name). The primary cause of this was routing congestion. Since it would be infeasibly expensive for FPGAs to implement a full crossbar interconnect, they instead have a finite number of wires which they assign

Dataset	Model	Test Accuracy%	Parameter Size (KiB)	Latency (ns)	Throughput (Samples/s)	Energy (nJ/Samp.)	LUTs (1000s)
MNIST	FINN	98.40	355.3	2440	1.56M	5445	83.0
	ULEEN†	98.46	262.0	940	4.05M	823	123.1
	DiffLogicNet *(xs)*	96.87	11.7	90	33.3M	17.2	9.6
	DiffLogicNet *(sm)**	97.62	23.4	*95*	*33.3M*	—	*19.1*
	DWN *(n=2; lg)*	98.27	**5.9**	135	25.0M	42.3	10.3
	DWN *(n=6; sm)*	97.80	11.7	**60**	**50.0M**	**2.5**	**2.1**
	DWN *(n=6; lg)*	98.31	23.4	125	25.0M	19.0	4.6
	DWN *(n=6; lg; +aug)*†	**98.77**	23.4	135	22.2M	21.6	4.6
Fashion-MNIST	FINN	84.36	355.3	2440	1.56M	5445	83.0
	ULEEN	87.86	262.0	940	4.05M	823	123.1
	DiffLogicNet	87.44	11.7	270	9.52M	119.8	11.4
	DWN *(n=2)*	**89.12**	**7.8**	255	**10.0M**	145.4	13.6
	DWN *(n=6)*	89.01	31.3	**250**	**10.0M**	**90.9**	**7.6**
KWS	FINN	70.60	324	7780	0.67M	5716	42.8
	ULEEN	70.34	101.0	390	10.0M	642	141.1
	DiffLogicNet*	64.18	23.4	*265*	*10.0M*	—	*20.3*
	DWN *(n=2)*	70.92	**2.9**	**235**	**10.5M**	79.2	6.8
	DWN *(n=6)*	**71.52**	12.5	**235**	**10.5M**	**42.3**	**4.8**
ToyADMOS	FINN	86.10	36.1	3520	1.57M	547	14.1
	ULEEN	86.33	16.6	340	11.1M	143	29.4
	DiffLogicNet	86.66	3.1	165	16.7M	23.2	3.9
	DWN *(n=2; sm)*	86.68	**0.9**	**115**	**25.0M**	7.6	2.2
	DWN *(n=6; sm)*	86.93	3.1	120	22.2M	**5.8**	**1.3**
	DWN *(n=6; lg)*	**89.03**	28.1	165	16.7M	45.4	6.2
CIFAR-10	FINN	**80.10**	183.1	283000	21.9K	150685	46.3
	ULEEN	54.21	1379	—	—	—	—
	DiffLogicNet*	57.29	250.0	*11510*	*87.5K*	—	*283.3*
	DWN *(n=2)**	57.51	**23.4**	*2200*	*468K*	—	*45.7*
	DWN *(n=6)*	57.42	62.5	**2190**	**468K**	**3972**	**16.7**

Table 6.4: Implementation results for DWNs and prior efficient inference models, including my prior work ULEEN, on a Z-7045 FPGA. *Model could not be synthesized; hardware values are approximate. †ULEEN used augmented data for MNIST, so I present DWN results for MNIST with and without this augmentation strategy.

signals to during synthesis. In general, routing resources are more readily available for short-distance signals versus long-distance signals. Both DiffLogicNet and DWNs have unstructured interconnect between layers. This results in many long wires, making these models more difficult to route. The DiffLogicNet models and DWNs with n=2 were more affected by this issue since they need more LUTs (and therefore more complex interconnect) to match the accuracy of the DWNs with n=6. All DWNs

with n=6 were successfully routed and implemented, though congestion would become an issue for them as well if they were sufficiently scaled up.

An interesting takeaway from these results is that the parameter sizes of DWN models are not necessarily good predictors of their hardware efficiency. For instance, the large MNIST model with n=2 has $\sim$1/4 the parameter size of the n=6 model, yet more than twice the area and energy consumption. Since modern Xilinx FPGAs use LUT-6s natively, models with n=6 are inherently more efficient to implement. Vivado can perform logic optimizations which map multiple DWN LUT-2s to a single FPGA LUT-6, but this is not enough to offset the $\sim$4$\times$ larger number of RAM nodes needed to achieve the same accuracy with n=2.

6.8.3 Microcontroller Implementation Results

Table 6.5 compares both DWN implementations against XGBoost on the Nano. The bit-packed DWN implementation is consistently more accurate than XGBoost, by an average of 5.4%, and particularly excels on non-tabular multi-class datasets such as MNIST and KWS. However, it is considerably slower, by an average factor of 8.3$\times$. The unpacked implementation is 15% faster than XGBoost and is still 1.2% more accurate on average, but is slightly less accurate ($-$0.6%) on one of the evaluated datasets (higgs). Overall, DWNs are good choices for low-end microcontrollers when accuracy is the most important consideration, but may not always be the best option when high throughput is a priority.

6.8.4 Sensitivity Analysis

Table 6.6 shows the results of adding the EFD learning rule and learnable input mapping to a minimal multilayer DWN. Using EFD without learnable mapping does not impact accuracy for KWS or CIFAR-10 since there is only one learnable layer for those two models. EFD increases accuracy by an average of 0.6% on models with multiple layers, while learnable mapping increases accuracy by 6.0%. When both

Dataset	DWN				XGBoost	
	Bit-Packed		Unpacked			
	Acc.	Thrpt	Acc.	Thrpt	Acc.	Thrpt
MNIST	97.9%	16.5/s	94.5%	108/s	90.2%	81/s
FashionMNIST	88.2%	16.4/s	84.1%	95/s	83.2%	81/s
KWS	69.6%	16.4/s	53.6%	109/s	51.0%	103/s
ToyADMOS	88.7%	17.7/s	86.1%	112/s	85.9%	94/s
phoneme	89.5%	17.8/s	87.5%	298/s	86.5%	265/s
skin-seg	99.8%	17.5/s	99.7%	298/s	99.4%	268/s
higgs	72.4%	17.1/s	71.2%	254/s	71.8%	245/s

Table 6.5: Model accuracies and throughputs (in inferences per second) for DWNs and XGBoost on the Elegoo Nano, a low-cost commodity microcontroller. All models are as large as possible within the constraints of the device's memory. I created DWN implementations with and without bit-packing, which reduces the memory footprint of a model at the cost of inference speed.

improvements are used, accuracy increases by 7.0%, suggesting that they behave synergistically. Table 6.7 shows the impact of adding spectral normalization to the models that I use for packed MCU deployment. This gives a significant benefit for four out of the seven datasets, with an average improvement of 1.2%.

Dataset	FD	+EFD	+LM	+EFD +LM (Full DWN)
MNIST	96.15%	96.59%	98.30%	98.31%
FashionMNIST	85.74%	86.88%	87.94%	89.01%
KWS	52.33%	52.33%	70.24%	71.52%
ToyADMOS	87.73%	88.02%	88.52%	89.03%
CIFAR-10	48.37%	48.37%	55.36%	57.42%

Table 6.6: Impacts from adding the EFD learning rule and learnable mapping to minimal multilayer weightless models based on the FD learning rule.

Figure 6.8 summarizes the improvements contributed by the three works discussed in this dissertation, BTHOWeN, ULEEN, and DWNs, over Bloom WiSARD [130], which was previously the state-of-the-art for WNNs. My initial implementations of

Dataset	DWN	+SN	Change
MNIST	97.82%	97.88%	0.06%
FashionMNIST	87.10%	88.16%	1.06%
KWS	67.17%	69.60%	2.43%
ToyADMOS	88.04%	88.68%	0.64%
phoneme	89.55%	89.50%	-0.05%
skin-seg	99.65%	99.83%	0.18%
higgs	68.51%	72.42%	3.91%

Table 6.7: Impacts of spectral normalization on the DWN models used for the "packed" inference implementation on the Elegoo Nano MCU.

multilayer WNNs could achieve quite small parameter sizes, but suffered in accuracy. By contrast, the final DWN models have even smaller memory footprints—up to 44× smaller than ULEEN—while still achieving equal or better accuracy. This establishes them as compelling options for edge applications in a wide range of contexts.

6.8.5 Additional Comparisons

In the last few years, there has been a rapid growth in interest in edge-optimized solutions to deep learning, and a proliferation of approaches. I discuss here a few extra comparisons with some interesting recent works as lagniappe.

6.8.5.1 Versus LogicNets Successors

PolyLUT [17] and NeuraLUT [18] are two very recent works which extend LogicNets [146] by using a more complex internal representation for NEQ units. Since ULEEN compared favorably against LogicNets (§5.6), I include a comparison of DWN against these successor models in Table 6.8. Among the small JSC models, NeuraLUT has the lowest latency, but it is unclear whether this includes the argmax computation; if it does not, then the comparable latency of the DWN model is only 1.5ns. Among the large JSC models, LogicNets uses the fewest flip-flops, but also uses more than 17× the LUTs of the DWN and is 3.8% less accurate. Besides these two exceptions,

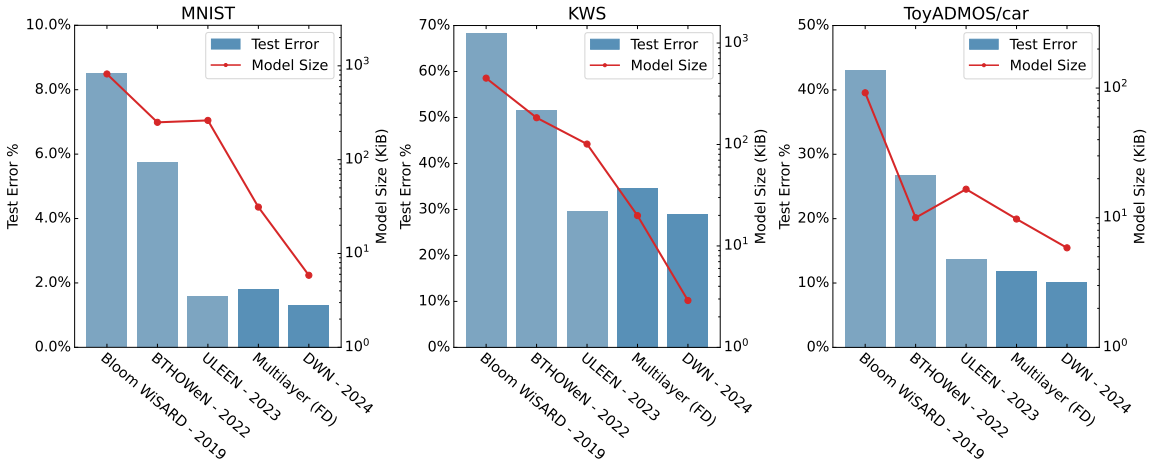


Figure 6.8: Accuracies and model sizes for Bloom WiSARD, BTHOWeN, ULEEN, and DWNs on the MNIST, KWS, and ToyADMOS datasets. I also show results for a simpler multilayer WNN without EFD or learnable mapping. Multilayer models were optimized for parameter size by setting n=2—note the log scale.

the DWN models are superior on all metrics.

6.8.5.2 Versus FPGA-Based XGBoost

There is surprisingly little literature regarding XGBoost on FPGAs. FAXID [59] proposes an FPGA-based inference accelerator architecture and deploys it using Alveo U50 and U200 accelerator cards, which are much more capable than the Z-7045 used for DWNs. Table 6.9 compares DWNs against their XGBoost models on two out of their four datasets; of the remaining two, one is proprietary, and the other appears to no longer be available. Despite using a less capable FPGA, the DWN models are massively superior in area, latency, and energy, while still achieving higher accuracy. This may suggest that XGBoost is inherently a poor choice for FPGAs due to the control flow complexity of its tree traversal, or it may merely mean that this is a domain in which more research is needed.

Dataset	Model	Accuracy	LUT	FF	Fmax (MHz)	Latency (ns)	ADP (LUT×ns)
MNIST	PolyLUT	96%	70673	4681	378	16	1.1e+06
	NeuraLUT	96%	54798	3757	431	12	6.6e+05
	DWN *(sm)*	**97.8%**	**2092**	**1757**	**873**	**9.2**	**1.9e+04**
JSC *(sm)*	LogicNets	67.8%	214	244	1585	—	—
	PolyLUT	72%	12436	773	646	5	6.2e+04
	NeuraLUT	72%	4684	341	727	**3**	1.4e+04
	DWN	**73.7%**	**134**	**106**	**1361**	3.7	**4.9e+02**
JSC *(lg)*	LogicNets	71.8%	37931	**810**	427	13	4.9e+05
	PolyLUT	75%	236541	2775	235	21	5.0e+06
	NeuraLUT	75%	92357	4885	368	14	1.3e+06
	DWN	**75.6%**	**2144**	1457	**903**	**8.9**	**1.9e+04**

Table 6.8: Performance of DWNs versus other recent LUT-based models.

Dataset	Model	Device	Clock (MHz)	LUTs (1000s)	FFs (1000s)	BRAMs	Latency (ms/10k)	Energy (mJ/10k)	Test Acc.%
Madeline	XGB	U50	344	244	362	984	3.61	51.6	81.83
	XGB	U200	341	494	755	1740.5	3.99	39.9	81.83
	DWN	Z-7045	200	2.6	5.4	0	0.50	0.23	83.30
Fraud Detection	XGB	U50	344	244	362	984	2.27	32.5	93.51
	XGB	U200	341	494	755	1740.5	2.34	23.4	93.51
	DWN	Z-7045	200	5.0	7.9	0	0.25	0.16	94.02

Table 6.9: Performance of DWNs versus the FAXID XGBoost accelerator.

6.8.5.3 Versus Boosted Race Trees

Boosted Race Trees [143] are another hardware implementation of tree-based ensemble methods which use asynchronous logic and temporal coding to reduce circuit energy. Directly comparing this work with DWNs is difficult since it is implemented as a 14 nm ASIC while DWNs were evaluated on a 28 nm FPGA. A small Boosted Race Tree model achieves 95.7% accuracy on the MNIST dataset with 16.1M inferences per second, while consuming 7.8 nJ per inference. By comparison, the small DWN model from Table 6.4 achieves 97.8% accuracy on MNIST with 50.0M inferences per

second, while consuming 6.6 nJ per inference (2.5 nJ dynamic). Therefore, DWNs are still superior to this model, even before accounting for the huge improvements in efficiency we would expect to see from halving the feature size and moving from an FPGA to an ASIC.

6.8.5.4 Versus Other Tabular Models and Tiny Classifiers

A. Bacellar demonstrated that DWNs also perform well when compared with Tiny Classifier circuits and a range of leading approaches for tabular data, achieving state-of-the-art accuracies for many workloads. For more details on these comparisons, refer to the paper [22].

6.9 Summary

The Differentiable Weightless Neural Network (DWN) model represents a radical departure from prior WNNs. Most weightless models, including my prior works, BTHOWeN and ULEEN, are composed of a single layer of RAM nodes and have separate discriminators for each output class. Previous multilayer WNNs split RAM nodes into pyramidal structures, but maintained partitionings between and within classes. By contrast, DWNs are composed of multiple layers of LUTs with flexible, learnable interconnect. This allows for the reuse of LUTs and enables model parameter sizes far smaller than in any previous WNN, while maintaining state-of-the-art accuracy.

For edge FPGA inference, DWNs compare favorably against a variety of other architectures, including ULEEN, FINN, LogicNets, PolyLUT, and NeuraLUT, in accuracy, throughput, latency, and circuit area. DWN models are also small enough to be implemented on tiny, low-cost microcontrollers, which enables inference with reasonable accuracy and throughput on a device with only 2 KB of RAM. These factors establish DWNs as a leading solution for machine learning on extreme edge devices.

Chapter 7: Conclusion

7.1 Summary

Deploying machine learning on edge devices requires models to be optimized to reduce their parameter footprints, computational costs, and energy demands. To some extent, this is possible by applying well-established optimizations such as pruning and quantization to deep neural networks (DNNs); however, for particularly small "extreme edge" or "mist" devices such as deeply embedded sensors, this may be insufficient. To target this domain, prior works have instead used techniques such as binary neural networks (BNNs) [145], learned gate networks [73, 120], and lookup table (LUT) conversions of sparse low-precision DNNs [17, 18, 146].

In this dissertation, I proposed new approaches to machine learning on the extreme edge based on weightless neural networks (WNNs), a class of model which perform computation using nonlinear LUTs. WNNs can capture complex behaviors with shallow, sparsely-connected configurations of LUTs, which in theory makes them appealing for latency-critical tasks. However, prior WNNs lagged behind DNNs and BNNs in parameter sizes and accuracies, which prevented them from achieving this potential. Therefore, my research focused on developing small, accurate WNN models and high-throughput, low-latency, area-and-energy-efficient hardware accelerators via algorithm-hardware co-design.

I first proposed BTHOWeN [140], a weightless model which includes several optimizations over the prior weightless state-of-the-art. BTHOWeN introduces counting Bloom filters to bridge an incompatibility between the techniques proposed by two leading weightless models, optimizes RAM node input hashing to reduce hardware area and energy, and generalizes a Gaussian nonlinear thermometer encoding technique to improve the fidelity of input encoding. Compared against Bloom WiSARD [130], BTHOWeN reduces average model test errors and parameter sizes by

geometric averages of 62% and 56%, respectively, across nine multi-class classification datasets. Versus 8-bit quantized DNNs of comparable accuracy implemented with `hls4ml` [50], BTHOWeN reduces latency by 91% and dynamic energy by 82% on an FPGA.

I next introduced ULEEN [137, 138, 139], which provides additional improvements in model accuracy and efficiency. ULEEN proposes a novel multi-pass WNN learning rule which leverages backpropagation by using continuous-valued Bloom filters with the straight-through estimator. It also composes ensembles out of small weightless submodels using an additive aggregation technique, and introduces a weightless pruning strategy which can be used to identify and eliminate the least-useful RAM nodes on either a global or a per-discriminator basis. Using an enhanced FPGA-based accelerator architecture, ULEEN achieves a 7.1× reduction in steady-state energy per inference and a 4.5× reduction in area-delay product (ADP) versus fully-connected BNNs implemented using Xilinx FINN [145]. It also compares favorably against LogicNets [146], an FPGA platform for extreme-throughput inference.

Lastly, I discussed methods of constructing multilayer WNNs, a challenging problem given the difficulty of defining derivatives with respect to LUT inputs when using a backpropagation-based approach to training. I presented several strategies to training and additional optimizations that were explored, culminating in the development of the DWN [22] model. DWNs eliminate the need for hashing and are extraordinarily efficient in terms of parameter size, which enables them to be implemented in tiny microcontrollers. In particular, compared to an optimized implementation of XGBoost on an Elegoo Nano, throughput-optimized implementations of DWNs achieve a 15% speedup with a 1.2% improvement in accuracy. Accuracy-optimized implementations achieve a 5.4% improvement, albeit with a large (8.3×) slowdown due to the overhead of manipulating bit-packed data structures. FPGA implementations of DWNs can be made very efficient by aligning the sizes of the model LUTs with the underlying FPGA LUTs (i.e., LUT-6s for Xilinx FPGAs). This enables large

improvements in energy, latency, and circuit area, yielding a 2522× improvement in energy-delay product versus FINN and a 63× improvement versus ULEEN.

Overall, although they are not yet suitable for all applications, the optimized weightless neural networks I presented in this dissertation yield reasonably accurate models with tiny parameter sizes which can be implemented with exceptionally fast and efficient hardware. This makes them an important part of the solution for scaling machine learning to the extreme edge.

7.2 Future Work

There are abundant opportunities for future work in this domain. This section summarizes a few research directions which show promise to further improve the efficiency and generality of WNNs.

Convolutional WNNs: Convolutional filters can be represented using one or multiple LUTs. The polynomial EFD learning rule used for DWNs can be readily adapted to allow gradient-based updates to these filters, enabling the construction and training of models conceptually similar to convolutional DNNs (i.e., CNNs). While these models can achieve accuracies similar to DWNs without convolutional filters, with smaller parameter sizes, they struggle to achieve significantly higher accuracy. The challenge here seems to be related to the difficulty of training sparsely-connected convolutional filters in general. Solving this problem may necessitate developing a new form of learnable mapping that can be applied to multiple convolutional layers.

Hybrid Weightless Architectures: Because DWNs allow gradients to be defined with respect to their inputs, they can be integrated alongside other layers in a DNN or BNN. One example of where this may be useful is in transformer models, which contain very large MLPs between self-attention layers. Replacing the MLPs in transformers with DWNs requires some method of producing real-valued output. This

could be accomplished by allowing LUTs on the last layer to produce real-valued output, or by adding a linear layer with real-valued weights.

Improved Hardware for Training and Inference: The memory access patterns of WNNs have an unusual form of structure, since each LUT is guaranteed to only access data within a small range, but the exact address it accesses is effectively random. GPUs are not particularly well-optimized to take advantage of this behavior, and training is therefore often bottlenecked by memory bandwidth. A hardware accelerator for training WNNs could be composed of many local memories laid out with this behavior in mind to provide better performance and assist in the exploration of more complex models. For inference, strategies should be explored which simplify the routing between layers, such as time-multiplexing of buses or pseudo-random mappings with some degree of underlying regular structure.

Online Learning with WNNs: The ability to dynamically update parameters in WNNs, particularly without needing gradient-based operations, could make them suitable for edge training or microarchitectural predictors. One work [150] achieved very promising accuracy using a weightless conditional branch predictor, but its parameter size was much too large to be practical in hardware. New approaches based on hashed or multi-layer WNNs with transfer learning could be helpful here.

"Mental Images" with DWNs: The DRASiW model is a modification of WiSARD which enables the generation of new samples after training [65], which can help in understanding the behaviors that were learned. Similar goals have also been explored for CNNs, albeit with very different methods [105]. An extension of the latter technique to DWNs should be reasonably straightforward and may give some additional insights into the representational ability of these models.

Conversion of DNNs to WNNs: Post-training DNN tabularization strategies such as those based on the MADDNESS [27] algorithm greatly reduce but do not entirely eliminate arithmetic during inference. By contrast, an approach which replaced layers or sequential groups of layers with DWNs could feasibly be arithmetic-free. A key challenge in doing this is that DNNs have dense connectivity, which enables them to learn certain behaviors which are linear but difficult to replicate with a sparsely-connected weightless model (e.g., the majority function).

Weightless Mixtures of Experts: While I did not have much success using DWNs as experts or gating models in a mixture-of-experts (MoE) context, I only explored one approach to this due to time constraints, and weightless adaptations of more recent work may prove to be more fruitful. It is also possible that the sizes of the models I was evaluating were simply beneath the minimum size for an MoE to be beneficial, in which case this will be worth revisiting as DWNs are scaled.

WNNs and Tsetlin Machines: The automata in Tsetlin machines [64] are in some ways the *inverse* of the RAM nodes in WNNs. RAM nodes based on LUTs can learn Boolean functions containing any number of conjunctive clauses, but their parameter sizes scale exponentially with their number of inputs. Conversely, the parameter size of a Tsetlin automaton scales linearly with its number of inputs, but it can only learn a single conjunctive clause. Integrating Tsetlin machines with DWNs is complicated, since they use a learning mechanism based on an iterated game that causes discrete state updates. However, if a continuous relaxation of a Tsetlin machine could be trained using gradient-based methods (similar to how DWNs are to some extent continuous relaxations of prior multi-layer WNNs), this could aid in the development of hybrid model architectures or differentiable Tsetlin machines.

Works Cited

[1] 4Paradigm. Madeline dataset. URL https://www.openml.org/search?type=data&sort=runs&id=41144&status=active.

[2] Youssef Abadade, Anas Temouden, Hatim Bamoumen, Nabil Benamar, Yousra Chtouki, and Abdelhakim Senhaji Hafid. A comprehensive survey on TinyML. *IEEE Access*, 11:96892–96922, 2023. doi: 10.1109/ACCESS.2023.3294111.

[3] Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez. PolyBench/Python: benchmarking Python environments with polyhedral optimizations. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 59–70, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383257. doi: 10.1145/3446804.3446842. URL https://doi.org/10.1145/3446804.3446842.

[4] Jyotibdha Acharya, Arindam Basu, Robert Legenstein, Thomas Limbacher, Panayiota Poirazi, and Xundong Wu. Dendritic computing: Branching deeper into machine learning. *Neuroscience*, 489:275–289, 2022. ISSN 0306-4522. doi: https://doi.org/10.1016/j.neuroscience.2021.10.001. Dendritic contributions to biological and artificial computations.

[5] Stefan Aeberhard and M. Forina. Wine dataset. UCI Machine Learning Repository, 1991. URL https://doi.org/10.24432/C5PC7J.

[6] R. Al-Alawi and T.J. Stonham. A training strategy and functionality analysis of digital multi-layer neural networks. *Journal of Intelligent Systems*, 2(1-4): 53–94, 1992. doi: doi:10.1515/JISYS.1992.2.1-4.53. URL https://doi.org/10.1515/JISYS.1992.2.1-4.53.

[7] Raida Al Alawi. FPGA implementation of a pyramidal weightless neural networks learning system. *International journal of neural systems*, 13:225–37, 09 2003. doi: 10.1142/S012906570300156X.

[8] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. Bit-pragmatic deep neural network computing. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 382–394, 2017.

[9] I. Aleksander. Canonical neural nets based on logic nodes. In *1989 First IEE International Conference on Artificial Neural Networks, (Conf. Publ. No. 313)*, pages 110–114, 1989.

[10] I. Aleksander, W.V. Thomas, and P.A. Bowden. WISARD: a radical step forward in image recognition. *Sensor Review*, 4(3):120–124, 1984. ISSN 0260-2288. doi: 10.1108/eb007637. URL `https://www.emerald.com/insight/content/doi/10.1108/eb007637/full/html`.

[11] Igor Aleksander, Thomas Clarke, and Antônio Braga. Binary neural systems: combining weighted and weightless properties. *Intelligent Systems Engineering*, 3:211 – 221, 02 1994. doi: 10.1049/ise.1994.0022.

[12] Igor Aleksander, Massimo De Gregorio, Felipe França, Priscila Lima, and Helen Morton. A brief introduction to weightless neural systems. In *17th European Symposium on Artificial Neural Networks (ESANN)*, pages 299–305, 04 2009.

[13] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. Ternary neural networks for resource-efficient AI applications. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2547–2554, 2017. doi: 10.1109/IJCNN.2017.7966166.

[14] P. Alinat. Periodic progress report 4. Technical report, ROARS Project ESPRIT II- Number 5516, 1993.

[15] Edgar Anderson. The species problem in iris, 1936.

[16] Renzo Andri, Lukas Cavigelli, Davide Rossi, and Luca Benini. YodaNN: An architecture for ultralow power binary-weight CNN acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1): 48–60, 2018. doi: 10.1109/TCAD.2017.2682138.

[17] Marta Andronic and George A. Constantinides. PolyLUT: learning piecewise polynomials for ultra-low latency FPGA LUT-based inference. In *2023 International Conference on Field Programmable Technology (ICFPT)*, pages 60–68. IEEE, 2023.

[18] Marta Andronic and George A. Constantinides. NeuraLUT: Hiding neural network density in boolean synthesizable functions. *arXiv preprint arXiv:2403.00849*, 2024.

[19] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM, April 2024. doi: 10.1145/3620665.3640366. URL `https://pytorch.org/assets/pytorch2-2.pdf`.

[20] Austin Appleby. Murmurhash3. https://github.com/aappleby/smhasher, 2016.

[21] Alan Bacellar, Zachary Susskind, Luis Villon, Igor Miranda, Leandro Santiago, Diego Dutra, Mauricio Jr, Lizy John, Priscila Lima, and Felipe França. Distributive thermometer: A new unary encoding for weightless neural networks. pages 31–36, 01 2022. doi: 10.14428/esann/2022.ES2022-94.

[22] Alan Tendler Leibel Bacellar, Zachary Susskind, Mauricio Breternitz Jr, Eugene John, Lizy Kurian John, Priscila Machado Vieira Lima, and Felipe M.G. França. Differentiable weightless neural networks. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=GBxflz0qdX.

[23] Colby Banbury, Vijay Janapa Reddi, Peter Torelli, Jeremy Holleman, Nat Jeffries, Csaba Kiraly, Pietro Montino, David Kanter, Sebastian Ahmed, Danilo Pau, Urmish Thakker, Antonio Torrini, Peter Warden, Jay Cordaro, Giuseppe Di Guglielmo, Javier Duarte, Stephen Gibellini, Videet Parekh, Honson Tran, Nhan Tran, Niu Wenxu, and Xu Xuesong. MLPerf tiny benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 2021.

[24] Michael Bayer. Mako templates for Python, 2021. URL https://www.makotemplates.org/.

[25] Younes Ben Mazziane, Sara Alouf, and Giovanni Neglia. A formal analysis of the count-min sketch with conservative updates. 05 2022.

[26] Rajen Bhatt and Abhinav Dhall. Skin Segmentation dataset. UCI Machine Learning Repository, 2012. URL https://doi.org/10.24432/C5T30C.

[27] Davis Blalock and John Guttag. Multiplying matrices without multiplying, 2021.

[28] W. W. Bledsoe and C. L. Bisson. Improved memory matrices for the n-tuple pattern recognition method. *IRE Transactions on Electronic Computers*, EC-11(3):414–415, 1962. doi: 10.1109/IRETELC.1962.5407930.

[29] W. W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), page 225–232, New York, NY, USA, 1959. Association for Computing Machinery. ISBN 9781450378680. doi: 10.1145/1460299.1460326. URL https://doi.org/10.1145/1460299.1460326.

[30] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692. URL https://doi.org/10.1145/362686.362692.

[31] Ernesto Burattini, Massimo De Gregorio, Victor M. G. Ferreira, and Felipe M. G. França. NSP: a neuro–symbolic processor. In José Mira and José R. Álvarez, editors, *Artificial Neural Nets Problem Solving Methods*, pages 9–16, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-44869-3.

[32] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-All: Train one network and specialize it for efficient deployment, 2019. URL https://arxiv.org/abs/1908.09791.

[33] Douglas O. Cardoso, Danilo Carvalho, Daniel S. F. Alves, Diego F. P. de Souza, Hugo C. C. Carneiro, Carlos E. Pedreira, Priscila M. V. Lima, and Felipe M. G. França. Financial credit analysis via a clustering weightless neural classifier. *Neurocomputing*, 183:70–78, 2016.

[34] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(79)90044-8. URL https://www.sciencedirect.com/science/article/pii/0022000079900448.

[35] Danilo Carvalho, Hugo Carneiro, Felipe França, and Priscila Lima. B-bleaching: Agile overtraining avoidance in the WiSARD weightless neural classifier. In *ESANN*, 04 2013.

[36] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. QuIP: 2-bit quantization of large language models with guarantees, 2024.

[37] Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. AdderNet: Do we really need multiplications in deep learning? In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1465–1474, 2020. doi: 10.1109/CVPR42600.2020.00154.

[38] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019. doi: 10.1109/JPROC.2019.2921977.

[39] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL `https://doi.org/10.1145/2939672.2939785`.

[40] Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations, 2023.

[41] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*, pages 3009–3018, 2019. doi: 10.1109/ICCVW.2019.00363.

[42] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training deep neural networks with binary weights during propagations.

In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 3123–3131, Cambridge, MA, USA, 2015. MIT Press.

[43] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1, 2016.

[44] Erika Covi, Elisa Donati, Xiangpeng Liang, David Kappel, Hadi Heidari, Melika Payvand, and Wei Wang. Adaptive extreme edge computing for wearable devices. *Frontiers in Neuroscience*, 15, 2021. ISSN 1662-453X. doi: 10.3389/fnins.2021.611300. URL `https://www.frontiersin.org/article/10.3389/fnins.2021.611300`.

[45] Nhu-Ngoc Dao, Yunseong Lee, Sungrae Cho, Eungha Kim, Ki-Sook Chung, and Changsup Keum. Multi-tier multi-access edge computing: The role for the fourth industrial revolution. In *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1280–1282, 2017. doi: 10.1109/ICTC.2017.8190921.

[46] Pau de Jorge, Amartya Sanyal, Harkirat Behl, Philip Torr, Grégory Rogez, and Puneet K. Dokania. Progressive skeletonization: Trimming more fat from a network at initialization. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=9GsFOUyUPi`.

[47] David Deterding, Mahesan Niranjan, and Tony Robinson. Connectionist Bench (Vowel Recognition - Deterding Data) dataset. UCI Machine Learning Repository. URL `https://doi.org/10.24432/C58P4S`.

[48] Thomas G. Dietterich. Ensemble methods in machine learning. In *Multiple Classifier Systems*, pages 1–15, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-45014-6.

[49] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997. ISSN 0196-6774. doi: https://doi.org/10.1006/jagm.1997.0873. URL https://www.sciencedirect.com/science/article/pii/S0196677497908737.

[50] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran, and Z. Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13 (07):P07027–P07027, jul 2018. doi: 10.1088/1748-0221/13/07/p07027. URL https://doi.org/10.1088/1748-0221/13/07/p07027.

[51] Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, and Joey Yiwei Li. DeepShift: Towards multiplication-less neural networks. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 2359–2368, 2021. doi: 10.1109/CVPRW53098.2021.00268.

[52] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SylO2yStDr.

[53] Victor C. Ferreira, Alexandre S. Nery, Leandro A. J. Marzulo, Leandro Santiago, Diego Souza, Brunno F. Goldstein, Felipe M. G. França, and Vladimir Alves. A feasible FPGA weightless neural accelerator. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2019. doi: 10.1109/ISCAS.2019.8702797.

[54] Victor M. G. Ferreira and Felipe M. G. França. Weightless implementations of weighted neural networks. In *Anais do IV Simpósio Brasileiro de Redes Neurais*, 1997.

[55] E.C.D.B.C. Filho, M.C. Fairhurst, and D.L. Bisset. Adaptive pattern recognition using goal seeking neurons. *Pattern Recognition Letters*, 12(3):131–138, 1991. ISSN 0167-8655. doi: https://doi.org/10.1016/0167-8655(91)90040-S. URL https://www.sciencedirect.com/science/article/pii/0167865591900405.

[56] R. A. Fisher. Iris dataset. UCI Machine Learning Repository, 1988. URL https://doi.org/10.24432/C56C76.

[57] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *ICLR*. OpenReview.net, 2019. URL http://dblp.uni-trier.de/db/conf/iclr/iclr2019.html#FrankleC19.

[58] Joshua Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel. Riptide: Fast end-to-end binarized neural networks. In I. Dhillon, D. Papailiopoulos, and V. Sze, editors, *Proceedings of Machine Learning and Systems*, volume 2, pages 379–389, 2020. URL https://proceedings.mlsys.org/paper/2020/file/2a79ea27c279e471f4d180b08d62b00a-Paper.pdf.

[59] Archit Gajjar, Priyank Kashyap, Aydin Aysu, Paul Franzon, Sumon Dey, and Chris Cheng. FAXID: FPGA-accelerated XGBoost inference for data centers using HLS. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–9, 2022. doi: 10.1109/FCCM53951.2022.9786085.

[60] S.I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, 1990. doi: 10.1109/72.80230.

[61] Lulu Ge and Keshab K. Parhi. Classification using hyperdimensional computing: A review. *IEEE Circuits and Systems Magazine*, 20(2):30–47, 2020. doi: 10.1109/MCAS.2020.2988388.

[62] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference, 2021.

[63] Kiran Gopinathan and Ilya Sergey. Certifying certainty and uncertainty in approximate membership query structures. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification*, pages 279–303, Cham, 2020. Springer International Publishing. ISBN 978-3-030-53291-8.

[64] Ole-Christoffer Granmo. The Tsetlin machine – a game theoretic bandit driven approach to optimal pattern recognition with propositional logic, 2021.

[65] Bruno P. A. Grieco, Priscila M. V. Lima, Massimo De Gregorio, and Felipe M. G. França. Producing pattern examples from "mental" images. *Neurocomput.*, 73(7-9):1057–1064, March 2010. ISSN 0925-2312. doi: 10.1016/j.neucom.2009.11.015. URL `http://dx.doi.org/10.1016/j.neucom.2009.11.015`.

[66] Qingyu Guo, Xiaoxin Cui, Jian Zhang, Aifei Zhang, Xinjie Guo, and Yuan Wang. A 4-bit integer-only neural network quantization method based on shift batch normalization. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 707–711, 2022. doi: 10.1109/ISCAS48785.2022.9938013.

[67] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585 (7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

[68] Haibo He, Yang Bai, Edwardo A. Garcia, and Shutao Li. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, pages 1322–1328, 2008. doi: 10.1109/IJCNN.2008.4633969.

[69] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for model compression and acceleration on mobile devices. In *European Conference on Computer Vision*, 2018. URL `https://api.semanticscholar.org/CorpusID:52048008`.

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

[71] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL `https://proceedings.neurips.cc/paper/2016/file/d8330f857a17c53d217014ee776bfd50-Paper.pdf`.

[72] Justin C. Hulbert and Michael C. Anderson. The role of inhibition in learning. In Aaron S. Benjamin, J. Steven De Belle, Bruce Etnyre, and Thad A. Polk, editors, *Human Learning*, volume 139 of *Advances in Psychology*, pages 7–20. North-Holland, 2008. doi: https://doi.org/10.1016/S0166-4115(08)10002-4. URL `https://www.sciencedirect.com/science/article/pii/S0166411508100024`.

[73] Konstantinos Iordanou, Timothy Atkinson, Emre Ozer, Jedrzej Kufel, Grace Aligada, John Biggs, Gavin Brown, and Mikel Luján. Low-cost and efficient

prediction hardware for tabular data using tiny classifier circuits. *Nature Electronics*, 04 2024.

[74] Vikram Iyer, Hans Gaensbauer, Thomas Daniel, and Shyamnath Gollakota. Wind dispersal of battery-free wireless devices. *Nature*, 603:1–7, 03 2022. doi: 10.1038/s41586-021-04363-9.

[75] 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 *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[76] Younghyun Jo and Seon Joo Kim. Practical single-image super-resolution using look-up table. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 691–700, 2021. doi: 10.1109/CVPR46437.2021.00075.

[77] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham,

Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45 (2):1–12, jun 2017. ISSN 0163-5964. doi: 10.1145/3140659.3080246. URL `https://doi.org/10.1145/3140659.3080246`.

[78] Herve Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011. doi: 10.1109/TPAMI.2010.57.

[79] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. SMASH: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 600–614, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi: 10.1145/3352460.3358286. URL `https://doi.org/10.1145/3352460.3358286`.

[80] Andressa Kappaun, Karine Camargo, Fabio Rangel, Fabrício Firmino, Priscila Machado Vieira Lima, and Jonice Oliveira. Evaluating binary encoding techniques for WiSARD. In *2016 5th Brazilian Conference on Intelligent Systems (BRACIS)*, pages 103–108, 2016. doi: 10.1109/BRACIS.2016.029.

[81] Mikail Khona, Sarthak Chandra, Joy J. Ma, and Ila R. Fiete. Winning the lottery with neural connectivity constraints: Faster learning across cognitive tasks with spatially constrained sparse rnns. *Neural Computation*, 35(11): 1850–1869, 2023. doi: 10.1162/neco_a_01613.

[82] Kibeom Kim, Yongjo Jeong, Youngjoo Lee, and Sunggu Lee. Analysis of counting Bloom filters used for count thresholding. *Electronics*, 8(7), 2019.

ISSN 2079-9292. doi: 10.3390/electronics8070779. URL https://www.mdpi.com/2079-9292/8/7/779.

[83] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[84] Yuma Koizumi, Shoichiro Saito, Hisashi Uematsu, Noboru Harada, and Keisuke Imoto. ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection. In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 313–317, 2019. doi: 10.1109/WASPAA.2019.8937164.

[85] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. DDoS in the IoT: Mirai and other botnets. *Computer*, 50:80–84, 01 2017. doi: 10.1109/MC.2017.201.

[86] Nickolaos Koroniotis, Nour Moustafa, Elena Sitnikova, and Benjamin Turnbull. Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset, 2018. URL https://arxiv.org/abs/1811.00701.

[87] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical Report 0, University of Toronto, Toronto, Ontario, 2009.

[88] Tanishq Kumar, Kevin Luo, and Mark Sellke. No free prune: Information-theoretic barriers to pruning at initialization. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=Uzb45nolTb.

[89] Aditya Kusupati, Vivek Ramanujan, Raghav Somani, Mitchell Wortsman, Prateek Jain, Sham Kakade, and Ali Farhadi. Soft threshold weight reparameterization for learnable sparsity. In *Proceedings of the 37th International Conference on Machine Learning*, ICML'20. JMLR.org, 2020.

[90] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340052. doi: 10.1145/2833157.2833162. URL https://doi.org/10.1145/2833157.2833162.

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

[92] Yann LeCun and Corinna Cortes. MNIST handwritten digit database, 2010. URL http://yann.lecun.com/exdb/mnist/.

[93] Walter D. Leon-Salas, Thomas Fischer, Xiaozhe Fan, Golsa Moayeri, and Shaocheng Luo. A 64×64 image energy harvesting configurable image sensor. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1914–1917, 2016. doi: 10.1109/ISCAS.2016.7538947.

[94] Fengfu Li and Bin Liu. Ternary weight networks. *CoRR*, abs/1605.04711, 2016. URL http://arxiv.org/abs/1605.04711.

[95] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. FP-BNN: Binarized neural network on FPGA. *Neurocomputing*, 275:1072–1086, 2018. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2017.09.046. URL https://www.sciencedirect.com/science/article/pii/S0925231217315655.

[96] James Lighthill. Artificial intelligence: A general survey. In *Artificial Intelligence: a paper symposium*, 1973.

[97] Tao Lin, Sebastian U. Stich, Luis Barba, Daniil Dmitriev, and Martin Jaggi. Dynamic model pruning with feedback. In *International Conference on*

Learning Representations, 2020. URL https://openreview.net/forum?id=SJem8lSFwB.

[98] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing. ISBN 978-3-319-10602-1.

[99] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *CoRR*, abs/1510.03009, 2016.

[100] Zhi-Gang Liu and Matthew Mattina. Learning low-precision neural networks without straight-through estimator (STE). In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, IJCAI'19, page 3066–3072. AAAI Press, 2019. ISBN 9780999241141.

[101] Teresa Ludermir, Andre de Carvalho, Antônio Braga, and M.C.P. Souto. Weightless neural models: A review of current and past works. *Neural Computing Surveys*, 2:41–61, 01 1999.

[102] Leopoldo A.D. Lusquino Filho, Luiz F.R. Oliveira, Aluizio Lima Filho, Gabriel P. Guarisa, Lucca M. Felix, Priscila M.V. Lima, and Felipe M.G. França. Extending the weightless WiSARD classifier for regression. *Neurocomputing*, 416:280–291, 2020. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2019.12.134. URL https://www.sciencedirect.com/science/article/pii/S092523122030504X.

[103] Mostafa Mahmoud, Kevin Siu, and Andreas Moshovos. Diffy: a déjà vu-free differential deep neural network accelerator. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 134–147, 2018. doi: 10.1109/MICRO.2018.00020.

[104] Philipp Mayer, Michele Magno, and Luca Benini. Combining microbial fuel cell and ultra-low power event-driven audio detector for zero-power sensing in underwater monitoring. In *2018 IEEE Sensors Applications Symposium (SAS)*, pages 1–6, 2018. doi: 10.1109/SAS.2018.8336772.

[105] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going deeper into neural networks, 2015. URL `https://research.google/blog/inceptionism-going-deeper-into-neural-networks/`.

[106] Nour Moustafa and Jill Slay. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In *2015 Military Communications and Information Systems Conference (MilCIS)*, pages 1–6, 2015. doi: 10.1109/MilCIS.2015.7348942.

[107] Pete Mowforth and Barry Shepherd. Statlog (Vehicle Silhouettes) dataset. UCI Machine Learning Repository. URL `https://doi.org/10.24432/C5HG6N`.

[108] C.E. Myers. Output functions for probabilistic logic nodes. In *1989 First IEE International Conference on Artificial Neural Networks, (Conf. Publ. No. 313)*, pages 310–314, 1989.

[109] Shashank Nag, Zachary Susskind, Aman Arora, Alan T. L. Bacellar, Diego L. C. Dutra, Igor D. S. Miranda, Krishnan Kailas, Eugene B. John, Mauricio Breternitz Jr., Priscila M. V. Lima, Felipe M. G. França, and Lizy K. John. LogicNets vs. ULEEN: Comparing two novel high throughput edge ML inference techniques on FPGA. In *Proceedings of the Midwest Symposium on Circuits and Systems*, MWSCAS '24, page to appear. Institute of Electrical and Electronics Engineers, 2024.

[110] Kenta Nakai. Ecoli dataset. UCI Machine Learning Repository, 1996. URL `https://doi.org/10.24432/C5388M`.

[111] National Aeronautics and Space Administration. Statlog (Shuttle) dataset. UCI Machine Learning Repository. URL https://doi.org/10.24432/C5WS31.

[112] Nadia Nedjah, Felipe P. da Silva, Alan O. de Sá, Luiza M. Mourelle, and Diana A. Bonilla. A massively parallel pipelined reconfigurable design for M-PLN based neural networks for efficient image classification. *Neurocomputing*, 183:39–55, 2016. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2015.05.138. URL https://www.sciencedirect.com/science/article/pii/S092523121501989X. Weightless Neural Systems.

[113] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. *PatDNN: Achieving Real-Time DNN Execution on Mobile Devices with Pattern-Based Weight Pruning*, page 907–922. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450371025. URL https://doi.org/10.1145/3373376.3378534.

[114] NVIDIA. NVIDIA H100 tensor core GPU architecture. Technical report, NVIDIA Corporation, 2023. URL https://resources.nvidia.com/en-us-tensor-core.

[115] NVIDIA. Using FP8 with transformer engine. Technical report, NVIDIA Corporation, 2024. URL https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html.

[116] Ryan O'Donnell. Analysis of boolean functions, 2021.

[117] R H Olsson, C Gordon, and R Bogoslovov. Zero and near zero power intelligent microsystems. *Journal of Physics: Conference Series*, 1407(1):012042, nov 2019. doi: 10.1088/1742-6596/1407/1/012042. URL https://dx.doi.org/10.1088/1742-6596/1407/1/012042.

[118] Alessandro Pappalardo. Xilinx/brevitas, 2021. URL https://doi.org/10.5281/zenodo.3333552.

[119] Nicholas Perrone and Robert Kao. A general finite difference method for arbitrary meshes. *Computers & Structures*, 5(1):45–57, 1975. ISSN 0045-7949. doi: https://doi.org/10.1016/0045-7949(75)90018-8. URL https://www.sciencedirect.com/science/article/pii/0045794975900188.

[120] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Deep differentiable logic gate networks. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 2006–2018. Curran Associates, Inc., 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/0d3496dd0cec77a999c98d35003203ca-Paper-Conference.pdf.

[121] Chi-Sang Poon and Kuan Zhou. Neuromorphic silicon neurons and large-scale neural networks: Challenges and opportunities. *Frontiers in Neuroscience*, 5, 2011. ISSN 1662-453X. doi: 10.3389/fnins.2011.00108. URL https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2011.00108.

[122] Jorge Portilla, Gabriel Mujica, Jin-Shyan Lee, and Teresa Riesgo. The extreme edge at the bottom of the internet of things: A review. *IEEE Sensors Journal*, 19(9):3179–3190, 2019. doi: 10.1109/JSEN.2019.2891911.

[123] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016. URL http://arxiv.org/abs/1603.05279.

[124] Andres Rodriguez, Eden Segal, Etay Meiri, Evarist Fomenko, Young Jim Kim, Haihao Shen, and Barukh Ziv. Lower numerical precision deep learning inference and training. Technical report, Intel Corporation, 2018. URL https:

//www.intel.com/content/dam/develop/external/us/en/documents/lower-numerical-precision-deep-learning-jan2018-754765.pdf.

[125] Frank Rosenblatt. *Principles of Neurodynamics*. Spartan Books, 1962.

[126] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 10 1986. doi: 10.1038/323533a0.

[127] Arish S., Sharad Sinha, and Smitha K.G. Optimization of convolutional neural networks on resource constrained devices. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 19–24, 2019. doi: 10.1109/ISVLSI.2019.00013.

[128] Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. On the effect of dropping layers of pre-trained transformer models. *Comput. Speech Lang.*, 77(C), jan 2023. ISSN 0885-2308. doi: 10.1016/j.csl.2022.101429. URL https://doi.org/10.1016/j.csl.2022.101429.

[129] Simone Salerno. micromlgen 1.1.28, 2022. URL https://pypi.org/project/micromlgen/.

[130] Leandro Santiago, Leticia Verona, Fabio Rangel, Fabricio Firmino, Daniel S Menasché, Wouter Caarls, Mauricio Breternitz Jr, Sandip Kundu, Priscila MV Lima, and Felipe MG França. Weightless neural networks as memory segmented Bloom filters. *Neurocomputing*, 416:292–304, 2020.

[131] Noam Shazeer, *Azalia Mirhoseini, *Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=B1ckMDqlg.

[132] David Slate. Letter Recognition dataset. UCI Machine Learning Repository, 1991. URL https://doi.org/10.24432/C5ZP40.

[133] Joram Soch. Proof: Quantile function of the normal distribution, 2020. URL https://statproofbook.github.io/P/norm-qf.html.

[134] Ashwin Srinivasan. Statlog (Landsat Satellite) dataset. UCI Machine Learning Repository, 1993. URL https://doi.org/10.24432/C55887.

[135] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014. ISSN 1532-4435.

[136] Statista. Market size and revenue comparison for artificial intelligence worldwide from 2020 to 2030, 2024. URL https://www.statista.com/statistics/941835/artificial-intelligence-market-size-revenue-comparisons/.

[137] Zachary Susskind, Alan T.L. Bacellar, Aman Arora, Luis A.Q. Villon, Renan Mendanha, Leandro S. De Araújo, Diego L.C. Dutra, Priscila M.V. Lima, Felipe M.G. França, Igor D.S. Miranda, Mauricio Breternitz, and Lizy K. John. Pruning weightless neural networks. In *ESANN 2022 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pages 37–42, 2022. doi: http://dx.doi.org/10.14428/esann/2022.ES2022-55.

[138] Zachary Susskind, Aman Arora, Alan T. L. Bacellar, Diego L. C. Dutra, Igor D. S. Miranda, Mauricio Breternitz, Priscila M. V. Lima, Felipe M. G. França, and Lizy K. John. An FPGA-based weightless neural network for edge network intrusion detection. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '23,

page 232, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450394178. doi: 10.1145/3543622.3573140. URL https://doi.org/10.1145/3543622.3573140.

[139] Zachary Susskind, Aman Arora, Igor D. S. Miranda, Alan T. L. Bacellar, Luis A. Q. Villon, Rafael F. Katopodis, Leandro S. de Araújo, Diego L. C. Dutra, Priscila M. V. Lima, Felipe M. G. França, Mauricio Breternitz Jr., and Lizy K. John. ULEEN: A novel architecture for ultra-low-energy edge neural networks. *ACM Trans. Archit. Code Optim.*, 20(4), dec 2023. ISSN 1544-3566. doi: 10.1145/3629522. URL https://doi.org/10.1145/3629522.

[140] Zachary Susskind, Aman Arora, Igor D. S. Miranda, Luis A. Q. Villon, Rafael F. Katopodis, Leandro S. de Araújo, Diego L. C. Dutra, Priscila M. V. Lima, Felipe M. G. França, Mauricio Breternitz, and Lizy K. John. Weightless neural networks for efficient edge inference. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, page 279–290, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450398688. doi: 10.1145/3559009.3569680. URL https://doi.org/10.1145/3559009.3569680.

[141] Telefonaktiebolaget LM Ericsson. IoT connections outlook. *Ericsson Mobility Report*, 2023. URL https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/iot-connections-outlook.

[142] Vitor A.M.F. Torres, Brayan R.A. Jaimes, Eduardo S. Ribeiro, Mateus T. Braga, Elcio H. Shiguemori, Haroldo F.C. Velho, Luiz C.B. Torres, and Antonio P. Braga. Combined weightless neural network FPGA architecture for deforestation surveillance and visual navigation of UAVs. *Engineering Applications of Artificial Intelligence*, 87:103227, 2020. ISSN 0952-1976. doi: https://doi.org/10.1016/j.engappai.2019.08.021. URL https://www.sciencedirect.com/science/article/pii/S095219761930212X.

[143] Georgios Tzimpragos, Advait Madhavan, Dilip Vasudevan, Dmitri Strukov, and Timothy Sherwood. Boosted race trees for low energy classification. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 215–228, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362405. doi: 10.1145/3297858.3304036. URL https://doi.org/10.1145/3297858.3304036.

[144] J.R. Ullmann. Experiments with the n-tuple method of pattern recognition. *IEEE Transactions on Computers*, C-18(12):1135–1137, 1969. doi: 10.1109/T-C.1969.222599.

[145] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 65–74, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450343541. doi: 10.1145/3020078.3021744. URL https://doi.org/10.1145/3020078.3021744.

[146] Yaman Umuroglu, Yash Akhauri, Nicholas J. Fraser, and Michaela Blott. LogicNets: Co-designed neural networks and circuits for extreme-throughput applications, 2020.

[147] Unknown. Bank fraud detection dataset. URL https://www.kaggle.com/volodymyrgavrysh/fraud-detection-bank-dataset-20k-records-binary.

[148] Mart van Baalen, Andrey Kuzmin, Suparna S Nair, Yuwei Ren, Eric Mahurin, Chirag Patel, Sundar Subramanian, Sanghyuk Lee, Markus Nagel, Joseph Soriaga, and Tijmen Blankevoort. FP8 versus INT8 for efficient deep learning inference, 2023.

[149] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[150] Luis A.Q. Villon, Zachary Susskind, Alan T.L. Bacellar, Igor D.S. Miranda, Leandro S. de Araújo, Priscila M.V. Lima, Mauricio Breternitz, Lizy K. John, Felipe M.G. França, and Diego L.C. Dutra. A conditional branch predictor based on weightless neural networks. *Neurocomputing*, 555:126637, 2023. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2023.126637. URL https://www.sciencedirect.com/science/article/pii/S0925231223007609.

[151] Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SkgsACVKPH.

[152] Erwei Wang, James J. Davis, Peter Y. K. Cheung, and George A. Constantinides. LUTNet: Rethinking inference in FPGA soft logic, 2019.

[153] Pete Warden. Speech commands: A dataset for limited-vocabulary speech recognition, 2018. URL https://arxiv.org/abs/1804.03209.

[154] Paul J. Werbos. Applications of advances in nonlinear sensitivity analysis. In R. F. Drenick and F. Kozin, editors, *System Modeling and Optimization*, pages 762–770, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg. ISBN 978-3-540-39459-4.

[155] Daniel Whiteson. HIGGS dataset. UCI Machine Learning Repository, 2014. URL https://doi.org/10.24432/C5V312.

[156] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at Facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344, 2019. doi: 10.1109/HPCA.2019.00048.

[157] Pedro Xavier, Massimo De Gregorio, Felipe M. G. França, and Priscila M. V. Lima. Detection of elementary particles with the WiSARD n-tuple classifier. In *28th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2020, Bruges, Belgium, October 2-4, 2020*, pages 643–648, 2020. URL `https://www.esann.org/sites/default/files/proceedings/2020/ES2020-170.pdf`.

[158] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms, 2017.

[159] Xilinx. 7 series FPGAs configurable logic block user guide, 2016. URL `https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB`.

[160] Xilinx. Xilinx Power Estimator (XPE). 2021. URL `https://www.xilinx.com/products/technology/power/xpe.html`.

[161] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley J. Osher, Yingyong Qi, and Jack Xin. Understanding straight-through estimator in training activation quantized neural nets. *CoRR*, abs/1903.05662, 2019. URL `http://arxiv.org/abs/1903.05662`.

[162] Pengmiao Zhang, Neelesh Gupta, Rajgopal Kannan, and Viktor K. Prasanna. Attention, distillation, and tabularization: Towards practical neural network-based prefetching, 2024.

[163] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016. doi: 10.1109/MICRO.2016.7783723.

[164] Zhekai Zhang, Hanrui Wang, Song Han, and William J. Dally. SpArch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274, 2020. doi: 10.1109/HPCA47549.2020.00030.

[165] Xingyu Zhou, Robert Canady, Shunxing Bao, and Aniruddha Gokhale. Cost-effective hardware accelerator recommendation for edge computing. In *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020. URL `https://www.usenix.org/conference/hotedge20/presentation/zhou-xingyu`.

[166] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL `https://openreview.net/forum?id=S1_pAu9xl`.

Vita

Zachary Susskind was born in Houston, Texas. He received his Bachelor of Science in Electrical Engineering from The University of Texas at Austin in 2019 and continued immediately onward to graduate school. Throughout his undergraduate and graduate studies, he has completed a total of six internships at NVIDIA. He is a student member of the IEEE.

Address: ZSusskind@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.