Weightless Neural Networks for Efficient Edge Inference

OMITTED FOR BLIND REVIEW

ABSTRACT

Weightless neural networks (WNNs) are a class of machine learning model which use table lookups to perform inference, rather than the multiply-accumulate operations typical of deep neural networks (DNNs). Individual weightless neurons are capable of learning non-linear functions of their inputs, a theoretical advantage over the linear neurons in DNNs, yet state-of-the-art WNN architectures still lag behind DNNs in accuracy on common classification tasks. Additionally, many existing WNN architectures suffer from high memory requirements, hindering implementation. In this paper, we propose a novel WNN architecture, BTHOWeN, with key algorithmic and architectural improvements over prior work, namely counting Bloom filters, hardware-friendly hashing, and Gaussian-based nonlinear thermometer encodings. These enhancements improve model accuracy while reducing size and energy per inference. BTHOWeN targets the large and growing edge computing sector by providing superior latency and energy efficiency to both prior WNNs and comparable quantized DNNs. Compared to state-of-the-art WNNs across nine classification datasets, BTHOWeN on average reduces error by more than 40% and model size by more than 50%. We demonstrate the viability of a hardware implementation of BTHOWeN by presenting an FPGA-based inference accelerator, and compare its latency and resource usage against similarly accurate quantized DNN inference accelerators, including multi-layer perceptron (MLP) and convolutional models. The proposed BTHOWeN models consume almost 80% less energy than the MLP models, with nearly 85% reduction in latency. In our quest for efficient ML on the edge, WNNs are clearly deserving of additional attention.

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Hardware** → **Hardware accelerators**; • **Computer systems organization** → *Special purpose systems.*

KEYWORDS

Weightless Neural Networks, WNN, WiSARD, Neural Networks, Hardware Acceleration, Inference, Edge Computing

ACM Reference Format:
OMITTED FOR BLIND REVIEW. 2022. Weightless Neural Networks for Efficient Edge Inference. In *PACT '22: International Conference on Parallel Architectures and Compilation Techniques (PACT), October 10–12, 2022, Chicago, IL.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3559009.3569680

1 INTRODUCTION

In the last decade, deep neural networks (DNNs) have driven revolutionary advancements in fields such as object detection, image classification, speech recognition, and natural language processing. In fact, it is widely acknowledged that modern DNNs can achieve superhuman accuracy on image recognition and classification tasks [33]. However, the implementation of these models is expensive in both memory and computation. Table 1 shows the number of weights and multiply-accumulate operations (MACs) needed for some widely-known networks. These networks have excellent accuracy, but performing inference with them requires significant memory capacity and numerous MAC computation, which in turn consume a substantial amount of energy. This may be acceptable on large servers, but in the emerging domain of edge computing, models must be run on small, power-constrained devices. The amount of weight memory and the number of computations required by these DNNs make them impractical to implement in edge solutions. Consequently, DNNs for edge inference must typically trade off accuracy for reduced complexity through techniques such as pruning and low-precision quantization [22].

Weightless neural networks (WNNs) are an entirely distinct class of neural model, inspired by the decode processing of input signals in the dendritic trees of biological neurons [2]. WNNs are composed of artificial neurons known as *RAM nodes*, which have binary inputs and outputs. Unlike neurons in DNNs, RAM nodes do not use weighted combinations of their inputs to determine their response. Instead, RAM nodes use lookup tables (LUTs) to represent a Boolean functions of their inputs as a truth table. RAM nodes concatenate their inputs to form an address into this table, and produce the corresponding table entry as their response. A RAM node with n inputs can represent any of the 2^{2^n} possible logical functions of its inputs using 2^n bits of storage.

Table 1: Weights and MACs for popular DNNs [20, 39]

Metric	LeNet-5	AlexNet	VGG-16	Resnet-50	OpenPose
#Weights	60k	61M	138M	25.5M	46M
#MACs	341k	724M	15.5G	3.9G	180G
Year	1998	2012	2014	2015	2018

Foundational research in WNNs occurred from the 1950s through the 1970s. However, WiSARD (Wilkie, Stonham, and Aleksander's Recognition Device) [3], which was introduced in 1981 and sold commercially from 1984, was the first WNN to be broadly viable. WiSARD was a pattern recognition machine, specialized for image recognition tasks. Two factors led to its success. First, then-recent advancements in integrated circuit manufacturing allowed for the fabrication of complex devices with large RAMs. Additionally, WiSARD incorporated algorithmic improvements which greatly increased its memory efficiency over simpler WNNs, allowing for

PACT '22, October 10–12, 2022, Chicago,IL

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/3559009.3569680

the implementation of more sophisticated models. As recent results have formally shown, the VC dimension[1] of WiSARD is very large [10], meaning it has a large theoretical capacity to learn patterns. Many subsequent WNNs [28], including the model proposed in this paper, draw inspiration from WiSARD's basic architecture.

Training a WNN entails learning Boolean functions in its component RAM nodes. Both supervised [32] and unsupervised [41] learning techniques have been explored for this purpose. Many training techniques for WNNs directly set values in the RAM nodes. The mutual independence between nodes when LUT entries are changed means that each input in the training set only needs to be presented to the network once. By contrast, most DNN training techniques involve iteratively adjusting weights, and many epochs of training may be needed before a model converges. By leveraging one-shot training techniques, WNNs can be trained up to four orders of magnitude faster than DNNs and other well-known computational intelligence models such as SVM [9].

Algorithmic and hardware improvements, combined with widespread research efforts, drove rapid and substantial increases in DNN accuracies during the 2010s. The ability to rapidly train large networks on powerful GPUs and the availability of big data fueled an AI revolution which is still taking place. While DNNs drove this revolution, we believe that WNNs are now a concept worth revisiting due to increasing interest in low-power edge inference. WNNs also have potential as tools to accompany DNNs. For instance, it has been demonstrated that WNNs can be used to dramatically speed up the convergence of DNNs during training [5]. There are also many applications where a small network is run first for approximate detection; then, if needed, a larger network is used for more precision [25]. The approximate networks by design do not need high accuracy; high speed and low energy usage are more important considerations. WNNs are perfect for these applications. However, in order to realize the benefits of this class of neural network, work needs to be done to design optimized WNNs with high accuracy and low area and energy costs.

Microcontroller-based approaches to edge inference, such as tinyML, have attracted a great deal of interest recently due to their ability to use inexpensive off-the-shelf hardware [36]. However, these approaches to machine learning are thousands of times slower than dedicated accelerators.

In this paper, we explore techniques to improve the accuracy and reduce the hardware requirements of WNNs. These techniques include hardware-efficient counting Bloom filters, hardware implementation of recent algorithmic improvements such as bleaching [12, 24], and a novel nonlinear thermometer encoding. We combine these techniques to create a software model and hardware architecture for WNNs which we call BTHOWeN (**B**leached **T**hermometer-encoded **H**ashed-input **O**ptimized **We**ightless Neural **N**etwork; pronounced as *Beethoven*). We present FPGA implementations of inference accelerators for this architecture, discuss their associated tradeoffs, and compare them against prior work in WNNs and against DNNs with similar accuracy.

Our specific contributions in this paper are as follows:

(1) BTHOWeN, a weightless neural network architecture designed for edge inference, which incorporates novel, hardware-efficient counting Bloom filters, nonlinear thermometer encoding, and bleaching.
(2) Comparison of BTHOWeN with state-of-the-art WiSARD-based WNNs across nine datasets, with a mean 41% reduction in error and 51% reduction in model size.
(3) An FPGA implementation of the BTHOWeN architecture, which we compare against MLP and CNN models of similar accuracy on the same nine datasets, finding a mean 79% reduction in energy and 84% reduction in latency versus MLP models. Compared to CNNs of similar accuracy, the energy reduction is over 98% and latency reduction is over 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 our accelerator architecture. These are available at: https://github.com/ZSusskind/BTHOWeN.

The remainder of our paper is organized as follows: In Section 2, we provide additional background on WNNs, WiSARD, and prior algorithmic improvements. In Section 3, we present the BTHOWeN architecture in detail. In Section 4, we discuss software and hardware implementation details. In Section 5, we compare our model architecture against prior memory-efficient WNNs, and compare our accelerator architecture against a prior WNN accelerator and against MLPs and CNNs of comparable accuracy. Lastly, in Section 6, we discuss future work and conclude.

2 BACKGROUND AND PRIOR WORK

2.1 Weightless Neural Networks

Weightless neural networks (WNNs) are a type of neural model which use table lookups for computation. WNNs are sometimes considered a type of Binary Neural Network (BNNs), but their method of operation differs significantly from other BNNs. Most BNNs are based around popcounts, i.e. counting the number of 1s in some bit vector. For instance, the McCulloch-Pitts neuron [29], one of the oldest and simplest neural models, performs a popcount on its inputs and compares the result against a fixed threshold in order to determine its output. More modern approaches first take the XNOR of the input with a learned weight vector, allowing an input to be negated before the popcount occurs [14].

By contrast, the fundamental unit of computation in WNNs is the RAM node, an n-input, 2^n-output lookup table with learned 1-bit entries. Conventionally, all entries in the RAM nodes are initialized to 0. During training, inputs are binarized or discretized using some encoding scheme and then presented to the RAM nodes. The input bits to a node are concatenated to form an address, and the corresponding entry in the node's LUT is set to 1. Note that presenting the same input to the node again has no effect, since the corresponding bit position has already been set. Therefore, an advantage of this approach is that each training sample only needs to be presented once.

Lookup tables are able to implement any Boolean function of their inputs. Therefore, in theory, a WNN can be constructed with a single RAM node which takes all (encoded) input features as inputs.

[1] The Vapnik–Chervonenkis (VC) dimension measures the complexity of the knowledge represented by a set of functions that can be encoded by a binary classification algorithm [40]. While usually approximated by statistical methods, it is possible to establish the exact VC dimension for some learning methods, including WiSARD.

However, this approach has two major issues. First, the size of a RAM node grows exponentially with its number of inputs. Suppose we take a dataset such as MNIST [27] and apply a simple encoding strategy such that each of the original inputs is represented using 1 bit. Since images in the MNIST dataset are 28x28, our input vector has 784 bits, and therefore the RAM node requires 2^{784} bits of storage, about $1.7 * 10^{156}$ times the number of atoms in the visible universe. The second issue is that such a RAM node has no ability to generalize: if a single bit is flipped in an input pattern, the node can not recognize it as being similar to a pattern it has seen before.

A great deal of WNN literature revolves around finding solutions to these two issues. A discussion of many of these approaches can be found in [2, 28]. Many of these techniques require random behavior (e.g. replacing the entries in RAM nodes with Bernoulli random variables), which is challenging to implement in hardware. The WiSARD model is deterministic, addressing both issues with the single-RAM-node model while avoiding the pitfalls of other solutions, and is therefore a good starting point for designing more complex WNNs.

There are some structural similarities between WNNs and architectural predictors in microprocessors. For instance, using a concatenated input vector to index into a RAM node is conceptually similar to using a branch history register in a table-based branch predictor.

2.2 WiSARD

WiSARD [3], depicted in Figure 1, is perhaps the most broadly successful weightless neural model. WiSARD is intended primarily for classification tasks, and constructs a submodel known as a *discriminator* for each output class. Each discriminator is in turn composed of n-input RAM nodes; for an I-input model, there are $N \triangleq I/n$ such nodes per discriminator. Inputs are assigned to these RAM nodes using a pseudo-random mapping; typically, as in Figure 1, the same mapping is shared between all discriminators.

During training, inputs are presented only to the discriminator corresponding to the correct output class, and its component RAM nodes are updated. During inference, inputs are presented to *all* discriminators. Each discriminator then forms a bit vector from the outputs of its component RAM nodes and performs a popcount on this vector to produce a response value. The index of the discriminator with the highest response is taken to be the predicted class. For example, if the input image contains the digit "1", the response from discriminator 1 should be the highest.

If an input seen during inference is identical to one seen during training, then all RAM nodes of the corresponding discriminator will yield a 1, resulting in the maximum possible response. On the other hand, if the input is similar but not identical, then some subset of the RAM nodes may produce a 0, but many will still yield a 1. As long as the response of the correct discriminator is still stronger than the responses of all other discriminators, the network will output a correct prediction. In practice, WiSARD has a far greater ability to generalize than simpler WNN models.

WiSARD's performance is directly related to the choice of n. Small values of n give the model a great deal of ability to generalize, but may be insufficient to capture complex input patterns. Larger values of n increase the complexity of the Boolean functions that the model can represent [3], but may result in overfitting.

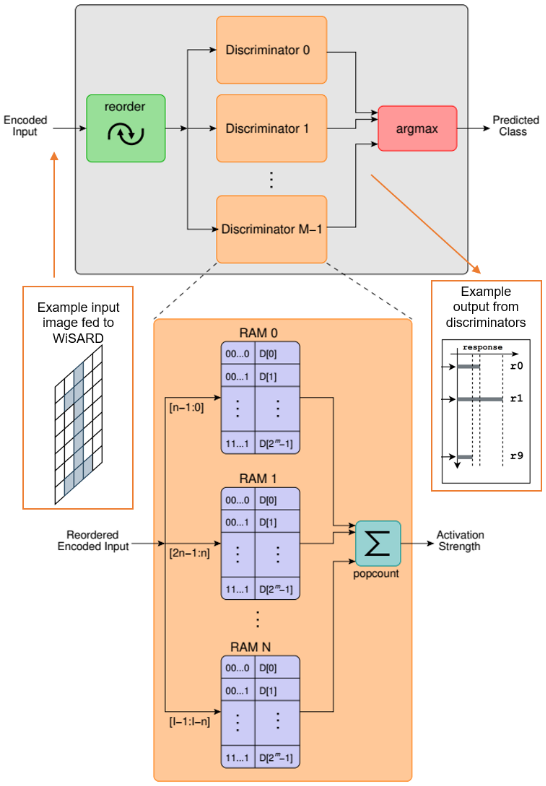

Figure 1: A depiction of the WiSARD WNN model with I inputs, M classes, and n inputs per RAM node. I/n RAM nodes are needed per discriminator, for a total of $M(I/n)$ nodes and $M(I/n)2^n$ bits of state.

2.3 Bloom Filters

Although the WiSARD model avoids the state explosion problem inherent in large, simple WNNs, practical considerations still limit the sizes of the individual RAM nodes. Increasing the number of inputs to each RAM node will, up to a point, improve the accuracy of the model; however, the model size will also increase exponentially. Fortunately, the contents of these large RAM nodes are highly sparse, as few distinct patterns are seen during training relative to the large number of table entries. Prior work has shown that using hashing to map large input sets to smaller RAMs can greatly decrease model size at a minimal impact to accuracy [15].

A Bloom Filter [7] is a hash-based data structure for approximate set membership. When presented with an input, a Bloom filter can return one of two responses: 0, indicating that the input is definitely not a member of the set, or 1, indicating that the element is *possibly* a member of the set. False negatives do not occur, but false positives can occur with a probability that increases with the number of elements in the set and decreases with the size of the underlying data structure [23]. Bloom filters have found widespread application for membership queries in areas such as networking, databases, web caching, and architectural predictions [8]. A recent model, Bloom WiSARD [15], demonstrated that replacing the RAM nodes in WiSARD with Bloom filters improves memory efficiency and model robustness [35].

Internally, a Bloom Filter is composed of k distinct hash functions, each of which takes an n-bit input and produces an m-bit output, and a 2^m-bit RAM. When a new value is added to the set represented by the filter, it is passed through all k hash functions, and the corresponding bit positions in the RAM are set. When the filter is checked to see if a value is in the set, the value is hashed, and the filter reports the value as present only if all k corresponding bit positions are set.

2.4 Bleaching

Traditional RAM nodes activate when presented with any pattern they saw during training, even if that pattern was only seen once. This can result in overfitting, particularly for large datasets, a phenomenon known as *saturation.* Bleaching [12] is a technique which prevents saturation by choosing a threshold b such that nodes only respond to patterns they saw at least b times during training. During training, this requires replacing the single-bit values in the RAM nodes with counters which track how many times a pattern was encountered. After training is complete, b is selected to maximize the accuracy of the network[2]. Once b has been selected, counter values greater than or equal to b can be statically replaced with 1, and counter values less than b with 0. Therefore, while additional memory is required during training, inference with a bleached WNN does not introduce any overhead.

In practice, bleaching can substantially improve the accuracy of WNNs. There have been several strategies proposed for finding the optimal bleaching threshold b; we use a binary search strategy based on the method proposed in [12]. Our approach performs a search between 1 and the largest counter value present in any RAM node. Thus, both the space and time overheads of bleaching are worst-case logarithmic in the size of the training dataset.

2.5 Thermometer Encoding

Traditionally, WNNs represent their inputs as 1-bit values, where an input is 1 if it rises above some pre-determined threshold[3] and 0 otherwise. However, it is frequently advantageous to use more sophisticated encodings, where each parameter is represented using multiple bits [26]. Binary integer encodings are not a good choice for WiSARD, since individual bits carry dramatically different amounts of information. For instance, in an 8-bit integer encoding, the most significant bit carries a great deal of information about the value of a parameter, while the least significant bit is essentially noise when taken in isolation. Since the assignment of bits to RAM nodes is randomized, this would result in some inputs to some RAM nodes being effectively useless.

In a thermometer encoding, a value is compared against a series of increasing thresholds, with the i'th bit of the encoded value representing the result of the comparison against the i'th threshold. Clearly if a value is greater than the i'th threshold, it is also greater than thresholds $\{0 \ldots (i-1)\}$. This *unary* encoding resembles mercury passing the markings on an analog thermometer, with bits becoming set from least to most significant as the value increases.

3 PROPOSED DESIGN: BTHOWEN

In this section, we present BTHOWeN, a WNN architecture which improves on the prior work by incorporating (i) counting Bloom filters to reduce model size while enabling bleaching, (ii) an inexpensive hash function which does not require arithmetic operations, and (iii) a Gaussian-based non-linear thermometer encoding to improve model accuracy. We also present an FPGA-based accelerator for this architecture, targeting low-power edge devices, shown in Figure 2. We incorporate both hardware and software improvements over the prior work.

3.1 Model

Our objective is to create a hardware-aware, high-accuracy, high-throughput WNN architecture. To accomplish this goal, we enhance the techniques described in Section 2 with novel algorithmic and architectural improvements.

3.1.1 Counting Bloom Filters. While Bloom filters were used in prior work [15], we augment them to be *counting* Bloom filters. Bloom filters can only track whether a pattern has been seen; in order to implement bleaching, we need to know *how many times* each pattern has been encountered. To accomplish this, we replace single-bit filter entries with multi-bit counters. When an item is added to the filter, rather than setting the corresponding entries, we instead increment the corresponding counter with the smallest value (or multiple counters in the event of a tie).

When performing a lookup, a counting Bloom filter returns 1 if the *smallest* counter value accessed is at least some threshold b; thus, the possible responses become “possibly seen at least b times" and “definitely not seen b times". Note that false negatives are still impossible; if a pattern has been seen i times, then the smallest of its corresponding counter values must be at least i.

Our implementation of counting Bloom filters is conceptually similar to count-min sketches under the conservative update rule [6]. However, count-min sketches use a separate data array for each hash function, while counting Bloom filters use a unified data array; this creates a tradeoff of false positive rate versus memory footprint.

3.1.2 Hash Function Selection. Bloom filters require multiple distinct hash functions, but do not prescribe what those hash functions should be. Prior work, including Bloom WiSARD [15, 35], used a double-hashing technique based on the MurmurHash [4] algorithm. However, this approach requires many arithmetic operations (e.g. 5 multiplications to hash a 32-bit value), and is therefore impractical in hardware. We identified an alternative approach based on sampling universal families of hash functions which is much less expensive to implement. Thus, while prior work used software-implemented Bloom filters, our design incorporates realistic filters which abide by hardware constraints.

A *universal family* of hash functions is a set of functions such that the odds of a hash collision are low in expectation for all functions in the family [11]. Some universal families consist of

[2] Alternatively, b may be chosen dynamically to serve as a tiebreaker when two or more discriminators produce an equal response. We do not explore this method of bleaching in this work.
[3] Frequently the mean value of the input in the training data

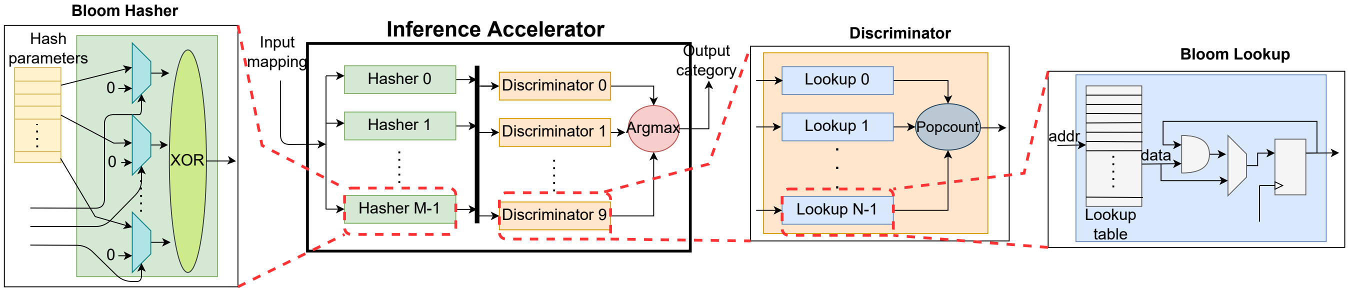

Figure 2: A diagram of the BTHOWeN inference accelerator architecture. We divide Bloom filters into dedicated Hasher and Lookup blocks. The Hasher block computes the H3 hash function on the input data, using a shared set of random hash parameters. The Discriminator block takes hashed data as input, passes it through Lookup units, 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.

highly similar functions, which differ only by the choices of constant "seed" parameters. We considered two such families when designing BTHOWeN.

The Multiply-Shift hash family [17] is a universal family of non-modulo hash functions which, for an n-bit input size and an m-bit output size, 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. The Multiply-Shift hash function consists of only a few machine instructions, so a software implementation is inexpensive. However, multiplication is a relatively costly operation in FPGAs, especially when many computations must be performed in parallel.

By contrast, the H3 family of hash functions [11] requires no arithmetic operations. 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 i'th 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 $2n - m$.

In practice, using Bloom filters in a WiSARD model requires many independent filters, each replacing a single RAM node. Each filter in turn requires multiple hash functions. We draw all hash functions from the same universal family, and use $\mathcal{P} = \{P_0 \ldots P_{k-1}\}$ to represent the random parameters for a filter's k hash functions.

For an implementation which uses Multiply-Shift hash functions, many multiplications need to be computed in parallel. This requires a large number of DSP slices on an FPGA. On the other hand, when using H3 hash functions, a large register file is needed for each set of hash parameters $\mathcal{P}$. However, we observed that sharing $\mathcal{P}$ between Bloom filters did not cause any degradation in accuracy. This effectively eliminates the only comparative disadvantage of the H3 hash function; hence, BTHOWeN uses the H3 hash function with the same $\mathcal{P}$ shared between all filters.

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.

3.1.3 Implementing Thermometer Encoding. Another enhancement we introduce in BTHOWeN is Gaussian non-linear thermometer encoding. Most prior work using thermometer encodings uses equal intervals between the thresholds. The disadvantage of this approach is that a large number of bits may be dedicated to encoding outlying values, leaving fewer bits to represent small differences in the range of common values.

For thermometer encoding in BTHOWeN, we assume that each input follows a normal distribution, and compute its mean and standard deviation from training data.For a t-bit encoding, we divide the Gaussian into $t + 1$ regions of equal probability. The values of the divisions between these regions become the thresholds we use for encoding. This provides increased resolution for values near the center of their range.

3.2 Training BTHOWeN

The process of training a network with the BTHOWeN architecture is shown in Figure 3. Hyperparameters, including the number of inputs in each sample, the number of output classes, details of the thermometer encoding, and configuration information for the Bloom filters, are used to initialize the model.

During training, samples are presented sequentially to the model. The label of the sample is used to determine which discriminator to train. The input is encoded, passed through the pseudo-random mapping, and presented to the filters in the correct discriminator. Filters hash their inputs and update their corresponding entries.

After training, the model is evaluated using the validation set at different bleaching thresholds. A binary search strategy is used to select the bleaching threshold b which gives the highest accuracy. The model is then binarized by replacing filter entries less than b with 0, and all other entries with 1. Binarization does not impact model accuracy, and allows counting Bloom filters to be replaced with conventional Bloom filters, which require less memory and are simpler to implement in hardware.

3.3 Inference with BTHOWeN

Figure 2 shows the design of an accelerator for inference with BTHOWeN WNNs. Since reusing the same random hash parameters for all Bloom filters does not degrade accuracy, we use a central register file to hold the hash parameters. Since all discriminators receive the same inputs, Bloom filters which are at the same index

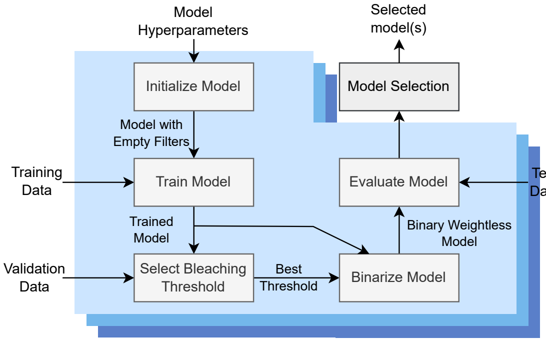

Figure 3: The training process for BTHOWeN models. Hyperparameters, consisting of the numbers of inputs and categories for the dataset, as well as tunable parameters, are used to construct an "empty" model, where all counter values are 0. Encoded training samples are sequentially presented to the model to update counter values. A validation set is used to select the optimal bleaching threshold b. This threshold is then used to binarize the trained model, replacing counters with binary values. We compare multiple models with different hyperparameters to find targets for implementation.

but in different discriminators (e.g. filter 1 in d_1 and filter 1 in d_2) also receive identical inputs. This means that their hashed values are also identical. It is therefore redundant and inefficient to compute hashes in each discriminator separately. Instead, we divide the Bloom filters into separate hashing units and lookup units, where hashing units perform H3 hash operations, and lookup units hold the Bloom filter data and perform AND reductions to determine the filter response. We place the hashing units at the top level of the design, before the discriminators, and broadcast their outputs to all discriminators. Since Bloom filters at the same index across different discriminators have different contents in their RAMs, lookup units can not be shared across discriminators.

If the bus bringing data from off-chip has insufficient bandwidth, then the accelerator will finish before the next input is ready. In this case, we can reduce the number of hash units by having each one compute the hashed inputs for multiple lookup units. We store partial results in a large central buffer until all hashes have been computed for a single set of parameters, then pass the hash results to all filters simultaneously, ensuring they operate in lockstep. This strategy allows us to reduce the area of the design without decreasing effective throughput.

The popcount module counts the number of 1s in the outputs of the filters in a discriminator, and the argmax module determines the index of the discriminator with the strongest response. These are equivalent to the corresponding modules in a conventional WiSARD model.

Since training with bleaching requires multi-bit counters for each entry in each Bloom filter, it introduces a large amount of memory overhead. For instance, in our experimentation, we found that some models had optimal bleaching values of more than 400. If we used saturating counters large enough to represent this value in the accelerator, it would increase the memory usage of the design by a factor of 9. Since our accelerator is intended for use in low-power edge devices, the advantages of supporting on-chip training do not seem worth the cost.

4 EVALUATION METHODOLOGY

4.1 Hardware Implementation

Our hardware source is written using Mako-templated SystemVerilog. Mako is a template library for Python which allows the Python interpreter to be run as a preprocessing step for arbitrary languages. This allows for greater flexibility and ease of use than the SystemVerilog preprocessor alone. When Mako is invoked, it generates pure SystemVerilog according to user-specified design parameters.

We targeted two different Xilinx FPGAs for this project. For most designs, we used the xc7z020clg400-1, a small, inexpensive FPGA available in the Zybo Z7 development board, which was used for prior work [19]. For our largest design, we targeted the Kintex UltraScale xcku035-ffva1156-1-c. Timing, power, and area numbers were obtained from Xilinx Vivado. Prior work implementing WNNs on FPGAs chose to profile the entire system, which revealed a major bottleneck in the form of SD card read bandwidth [19]; we are interested in the performance of the accelerator itself. We implement all models with a 100 MHz clock rate, and collect power numbers assuming a 12.5% switching rate.

Hash units produce output at a maximum throughput of 1 hash per cycle. Lookup units can consume hashed inputs at a rate of 1/cycle, and produce output at a rate of $1/k$ cycles, where k is the number of hash functions associated with a Bloom filter. Therefore, there is no point in having more hashing units than lookup units, and the maximum throughput of the design is $1/k$ cycles. This throughput could be improved by allowing multiple addresses to be read simultaneously in the lookup units. However, this would greatly increase circuit area, and such a design would generally not be any faster in practice; k is typically small enough that reading data into the accelerator from off-chip is the bottleneck.

At the top level of the design, we use a double-buffered deserialization unit which accumulates input data from the bus until a full sample has been read, then passes the entire sample to the accelerator. This helps enable all hardware units to operate in lockstep, simplifying our state machine logic and verification effort.

4.2 Datasets and Training

We created models for all classifier datasets discussed in [15]: MNIST [27], Ecoli [31], Iris [21], Letter [37], Satimage [38], Shuttle [13], Vehicle [30], Vowel [16], and Wine [1]. Since our accelerator does not support on-chip training, we implemented the training of models in software. This was done in Python, using the Numba JIT compiler to reduce the runtime of performance-critical functions. We performed a 90-10 train/validation split on the input dataset, using the former to learn the values in the counting Bloom filters and the latter to set the bleaching threshold.

4.3 WNN Model Sweeping

There are several model hyperparameters which can be changed to impact the size and accuracy of the model. Increasing the size of the Bloom filters decreases the likelihood of false positives, and thus improves accuracy. However, this greatly increases the model size, and eventually provides diminishing returns for accuracy as false positives become too rare to matter. Increasing the number of input bits to each 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, since more unique patterns are seen by each filter. Increasing the number of hash functions per Bloom filter can improve accuracy up to a point, but past a certain point actually begins to *increase* the frequency of false positives [7]. Lastly, increasing the number of bits in the thermometer encoding can improve accuracy at the cost of model size, but again provides diminishing returns as the amount of information each bit conveys decreases.

In order to identify optimal model hyperparameters, we ran many different configurations in parallel using an automated sweeping methodology. For MNIST, we used 1008 distinct configurations, sweeping all combinations of the hyperparameter settings shown in Table 2. For smaller datasets, we explored using 1-16 encoding bits per input, 128-8192 entries per Bloom filter, 1-6 hash functions per filter, and 6-64 input bits per Bloom filter.

Table 2: Hyperparameters swept for the creation of BTHOWeN models for the MNIST dataset

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

4.4 DNN Model Identification and Implementation

For each dataset, we trained MLPs that had similar accuracy to our BTHOWeN models. We identified the smallest iso-accuracy MLPs using a hyperparameter sweep. The trained models were then quantized to 8-bit precision to generate a TensorFlow Lite model. Hardware was generated for each MLP using the `hls4ml` tool [18]. `hls4ml` takes 4 inputs: (1) the weights generated by TensorFlow (.h5 format), (2) the structure of the model generated by TensorFlow (.json format), (3) the precision to be used for the hardware, and (4) the FPGA part being targeted. It generates C++ code corresponding to the model, and then invokes Xilinx Vivado HLS to generate the hardware design. We modified the generated C++ code such that the I/O interface width matched that of our hardware design for WNNs in order to ensure a fair comparison. We also modified HLS pragmas as needed to ensure that the resultant RTL could fit on the Zybo FPGA. The hardware design generated by Vivado HLS (invoked by `hls4ml`) was then synthesized and implemented using Xilinx Vivado to obtain area, latency, and power consumption metrics.

For the MNIST dataset, in addition to MLPs, we compared the BTHOWeN implementation with comparably accurate CNNs based on the LeNet-1 [27] architecture. The default convolutional layer implementations generated by `hls4ml` were too large to fit on our FPGA, and tuning HLS pragmas to reduce area resulted in a very inefficient implementation. This issue only impacted convolutional layers, and did not affect the MLPs. To make a more fair comparison with CNNs, we used the latency and resource utilization values for optimized implementations reported by Arish et. al. [34]. We then used the Xilinx Power Estimator (XPE) [42] to get approximate power values for the CNN.

5 RESULTS

5.1 Selected BTHOWeN Models

After performing a hyperparameter sweep, we needed to select one or more trained models for FPGA implementation, balancing tradeoffs between model size and accuracy. For each dataset except for MNIST, there was one model which was very clearly the best, with all more accurate models being many times larger.

Since MNIST is a more complex dataset, there was no clear single “best" model - instead, we identified “Small", “Medium", and “Large" models, which balanced size and accuracy at different points. Our objectives for the three MNIST models were:

- The small model would be comparable in area to the prior FPGA model in [19]
- The medium would be larger, but could still fit on the same FPGA (i.e. the Zybo Z7 board)
- The large model would fit on a mid-size commercial FPGA

We also experimented with MNIST models using traditional linear thermometer encodings, and observed a 12.9% reduction in mean error using the Gaussian encoding.

The configurations for all the models we selected are shown in Table 3.

Table 3: Details of the selected BTHOWeN models

Model Name	Bits /Input	Bits /Filter	Entries /Filter	Hashes /Filter	Size (KiB)	Test Acc.
MNIST-Small	2	28	1024	2	70.0	0.934
MNIST-Medium	3	28	2048	2	210	0.943
MNIST-Large	6	49	8192	4	960	0.952
Ecoli	10	10	128	2	0.875	0.875
Iris	3	2	128	1	0.281	0.980
Letter	15	20	2048	4	78.0	0.900
Satimage	8	12	512	4	9.00	0.880
Shuttle	9	27	1024	2	2.63	0.999
Vehicle	16	16	256	3	2.25	0.762
Vowel	15	15	256	4	3.44	0.900
Wine	9	13	128	3	0.422	0.983

5.2 Comparison with Iso-Accuracy Deep Neural Networks

Table 4 shows FPGA implementation results for BTHOWeN models and iso-accuracy quantized DNNs identified using a hyperparameter sweep across the nine datasets. We italicize the superior results for throughput, power, energy, and area. For the MNIST dataset, the medium BTHOWeN model is only 0.3% less accurate than the MLP, consumes just 16% of the energy of the MLP model, and reduces latency by almost 96%. The MLP uses fewer LUTs and FFs than the medium BTHOWeN model, but also requires DSP blocks and BRAMs on the FPGA. The BTHOWeN model compares even

Table 4: Comparison of BTHOWeN FPGA Models with Quantized DNNs of similar accuracy implemented in FPGAs. CNNs for MNIST are LeNet-1 variations from [34]. The WNN and MLP for each dataset are grouped in nearby rows for easy comparison; winning results are italicized. (For MNIST, we compare BTHOWeN-Medium and the MLP.)

Dataset	Model	Bus Width	Cycles per Inf.	Hash Units	Dyn. Power (Tot. Power) (W)	Dyn. Energy (Tot. Energy) (nJ/Inf.)	LUTs	FFs	BRAMs (36Kb)	DSPs	Accuracy
MNIST	BTHOWeN-Small	64	25	5	0.195 (0.303)	48.75 (75.8)	15756	3522	0	0	0.934
	BTHOWeN-Medium	64	*37*	5	0.386 (0.497)	*142.8 (183.9)*	38912	6577	*0*	*0*	0.943
	BTHOWeN-Large	64	74	6	3.007 (3.509)	2225 (2597)	151704	18796	0	0	0.952
	BTHOWeN-Large*	256	19	24	3.158 (3.661)	600.0 (695.6)	158367	25905	0	0	0.952
	MLP 784-16-10	64	846	-	*0.029 (0.134)*	245 (1133)	*2163*	*3007*	8	28	0.946
	CNN 1 (LeNet1) [34]	64	33615	-	0.058 (0.163)	19497 (54792)	5753	3115	7	18	0.947
	CNN 2 (LeNet1) [34]	64	33555	-	0.043 (0.148)	14429 (49661)	3718	2208	5	10	0.920
	Hashed WNN [19]	32	28	-	0.423 (0.528)	118.4 (147.8)	9636	4568	128.5	5	0.907
Ecoli	BTHOWeN	64	*2*	7	*0.012 (0.117)*	*0.24 (2.34)*	*353*	*223*	0	0	0.875
	MLP 7-8-8	64	14	-	0.03 (0.135)	4.2 (18.9)	1596	1615	0	0	0.875
Iris	BTHOWeN	64	*1*	6	*0.005 (0.109)*	*0.05 (1.09)*	*57*	*90*	0	0	0.980
	MLP 4-4-3	64	10	-	0.008 (0.112)	0.8 (11.2)	427	488	0	0	0.980
Letter	BTHOWeN	64	*4*	12	0.623 (0.738)	*24.92 (29.52)*	21603	*2715*	0	0	0.900
	MLP 16-40-26	64	26	-	*0.109 (0.259)*	39.52 (67.34)	*17305*	15738	0	0	0.904
Satimage	BTHOWeN	64	*5*	24	0.084 (0.190)	*4.2 (9.5)*	*3771*	*1131*	0	0	0.880
	MLP 36-16-16-6	64	25	-	*0.039 (0.144)*	9.75 (36)	7007	7558	0	0	0.878
Shuttle	BTHOWeN	64	*2*	3	0.018 (0.123)	*0.36 (2.46)*	*593*	*121*	0	0	0.999
	MLP 9-4-7	64	14	-	*0.013 (0.118)*	1.82 (16.52)	693	711	0	0	0.999
Vehicle	BTHOWeN	64	*5*	18	0.038 (0.143)	*1.9 (7.15)*	*1781*	*597*	0	0	0.762
	MLP 18-16-4	64	15	-	*0.024 (0.128)*	3.6 (19.2)	2824	3035	0	0	0.766
Vowel	BTHOWeN	64	*2*	12	*0.040 (0.145)*	*0.8 (2.9)*	*1559*	*756*	0	0	0.900
	MLP 10-18-11	64	18	-	0.070 (0.175)	12.6 (31.5)	5743	4663	0	0	0.903
Wine	BTHOWeN	64	*3*	9	*0.012 (0.117)*	*0.36 (3.51)*	*585*	*239*	0	0	0.983
	MLP 13-10-3	64	14	-	0.026 (0.131)	3.64 (18.34)	1836	1832	0	0	0.983

more favorably against CNNs. For example, CNN-1 has an accuracy of 94.7%, which is only slightly better than the 94.3% accuracy of the medium BTHOWeN model. But even with a pipelined CNN implementation, BTHOWeN consumes less than 0.4% of the energy of the CNN, while reducing latency from 33.6k cycles to just 37.

As Table 4 illustrates, for all datasets except MNIST and Letter, the BTHOWeN model's hardware implementation consumes fewer resources (LUTs and FFs) than its MLP counterpart. The reduction in total energy consumption of the BTHOWeN models ranges from 56.2% on Letter to 90.8% on Vowel. Reduction in latency ranges from 66.7% on Letter to 90.0% on Iris.

Figure 4 summarizes these results, showing the relative latencies, dynamic energies, and total energies of BTHOWeN models compared to DNNs. Overall, BTHOWeN models are significantly faster and more energy efficient than DNNs of comparable accuracy.

The different-sized models for MNIST, shown in Table 4, provide multiple tradeoff points for energy and accuracy. The Small and Medium models provide good energy efficiency, though the Medium model uses over twice the energy for a 0.9% improvement in accuracy. The Large model does not fit on the Zybo FPGA, so is implemented on a larger FPGA with much higher static power consumption, and is much slower and less energy-efficient. However, if we take advantage of the large number of I/O pins on the large FPGA and implement a 256b bus, energy consumption is reduced by nearly a factor of 4 (the "Large*" row in Table 4).

5.3 Comparison with Prior Weightless Neural Networks

Bloom WiSARD, the prior state-of-the-art for WNNs, used Bloom filters to achieve far smaller model sizes than conventional WiSARD models with only slight penalties to accuracy [15]. Results were reported on nine multi-class classifier datasets, which we adopted for our analysis. No hardware implementation (FPGA or ASIC) was provided in this work, hence the comparison we present here is based only on accuracy and model parameter size.

We compared the BTHOWeN models in Table 3 against the results reported by Bloom WiSARD on all nine datasets, achieving superior accuracy with a smaller model parameter size in all cases. The results are summarized in Figure 5. On average, our models have 41% less error with a 51% smaller model size compared to Bloom WiSARD, which did not incorporate bleaching or thermometer encoding. Our improvements indicate the benefits of these techniques. Although the prior work did not propose a hardware implementation, we anticipate that our advantage in hardware would be even larger due to our much simpler and more efficient choice of hash function.

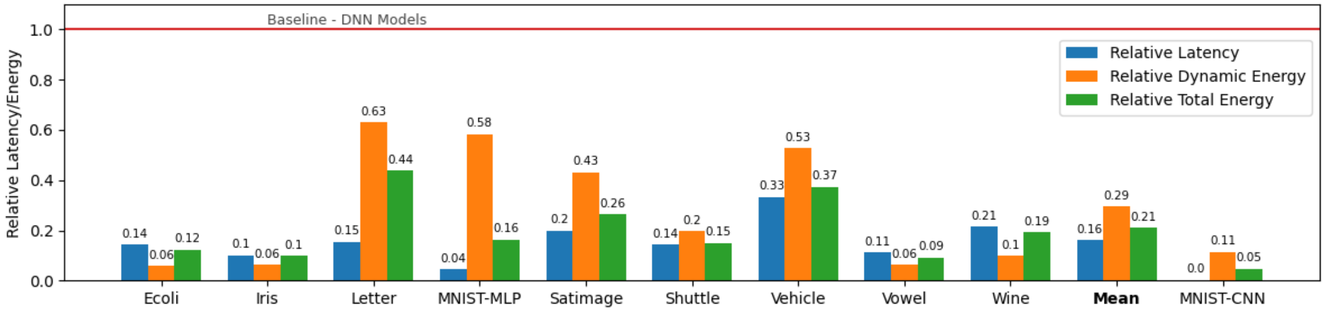

Figure 4: The relative latencies and energies of BTHOWeN models versus iso-accuracy DNNs. For MNIST, our Medium model is compared against the MNIST MLP model, and our Large (64b bus) model is compared against CNN 1. The results of the comparison with CNN 1 are not used in the computation of the average, since the Large model uses a different FPGA. We implemented baseline MLPs and BTHOWeN models at 100MHz and obtained metrics from Xilinx tools.

One unusual result is on the Shuttle dataset, for which our model has ~99% less error than prior work. Shuttle is an anomaly-detection dataset in which 80% of the training data belongs to the “normal" class [13]. We suspect that, since Bloom WiSARD does not incorporate bleaching, the discriminator corresponding to this class became saturated during training.

5.4 Comparison with Prior FPGA Implementation

In [19], a WNN accelerator for MNIST was implemented on the Zybo FPGA (xc7z020clg400-1) with Vivado HLS. We used this same FPGA at the same frequency (100 MHz). The row with Model=“Hashed WNN" in Table 4 shows the implementation results for the prior art accelerator. Its latency and energy consumption are between our small and medium models, but it is much less accurate than even our small model. This accelerator is greatly harmed by its slow memory access, which increases the impact of static power on its energy consumption.

Exact accelerator latency values were not published. The accelerator reads in one 28-bit filter input per cycle, and uses a 1-bit-per-input encoding, so it takes 28 cycles to read in a 784-bit MNIST sample. Therefore, we use 28 cycles as a lower bound on the time per inference for their design. The energy number in Table 4 for [19] is a lower bound based on this cycle count and published power values.

Our implementation has significant differences which contribute to BTHOWeN's superior accuracy and efficiency: (1) The prior accelerator used a simple hash-table-based encoding scheme which had explicit hardware for collision detection; we use an approach based on counting Bloom filters which does not need collision detection. (2) Models for the prior accelerator did not incorporate bleaching or thermometer encoding; instead, they used a simple 1-bit encoding based on comparison with a parameter's mean value. We use counting Bloom filters to enable bleaching.

Since the accelerator in [19] did not incorporate bleaching, training did not require multi-bit counters, making it inexpensive to support.

5.5 Model Tradeoff Analysis

We use MNIST as an illustrative example of the tradeoffs present in model selection. Figure 6 presents the results obtained from sweeping over the MNIST dataset with the configurations presented in Table 2. In the first four subplots of Figure 6, we vary one hyperparameter of the model: respectively, the number of bits used to encode each input, the number of inputs to each Bloom filter, the number of entries in each filter, and the number of distinct hash functions for each filter. We show four lines: three of them represent the Small, Medium, and Large models where only the specified hyperparameter was varied, while the fourth represents the best model with the given value for the hyperparameter.

We see diminishing returns as the number of encoding bits per input and the number of entries per Bloom filter increase. The Small and Medium models rapidly lose accuracy as the number of inputs per filter increases, but the Large model, with its large filter LUTs, is able to handle 49 inputs per filter without excessive false positives. These results align with the theoretical behaviors discussed earlier.

One surprising result was that, although there was a slight accuracy increase going from 1 hash function per filter to 2, continuing to increase this had minimal impact. In theory, we would expect that continuing to increase this value would eventually result in a loss of accuracy due to high false positive rates. One explanation for this is that the BTHOWeN model reports the index of the class with the strongest response; since a higher false positive rate would impact the response of *all* classes, the predicted class should remain unaffected as long as the increase is proportional. Another observation, shown in Figure 6.e with the second y-axis, is that the optimal bleaching value b (i.e. the smallest value which becomes 1 when the model is binarized) increases to compensate for the larger number of hash functions. This plot shows variants of the Small model with up to 128 hash functions per Bloom filter. When b is fixed at 16, accuracy collapses, but when the optimal value is chosen using the same binary search strategy we use normally, it is better able to compensate. This provides a good example of how bleaching improves the robustness of BTHOWeN.

The last subplot shows the most accurate MNIST model we were able to obtain with a given maximum model size. We notice diminishing returns as model size increases. It is evident that in order

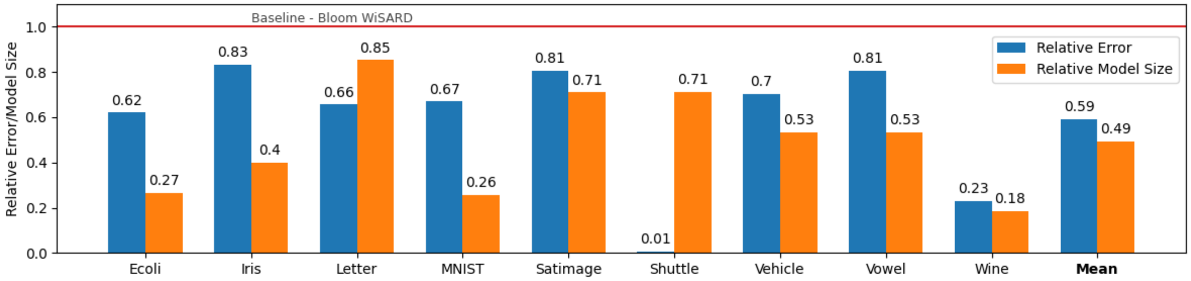

Figure 5: The relative errors and model sizes of the models shown in Table 3 versus Bloom WiSARD [15]. BTHOWeN outperforms the prior work on all nine datasets in both accuracy and model size. For the MNIST dataset, our MNIST-Medium model was used for comparison.

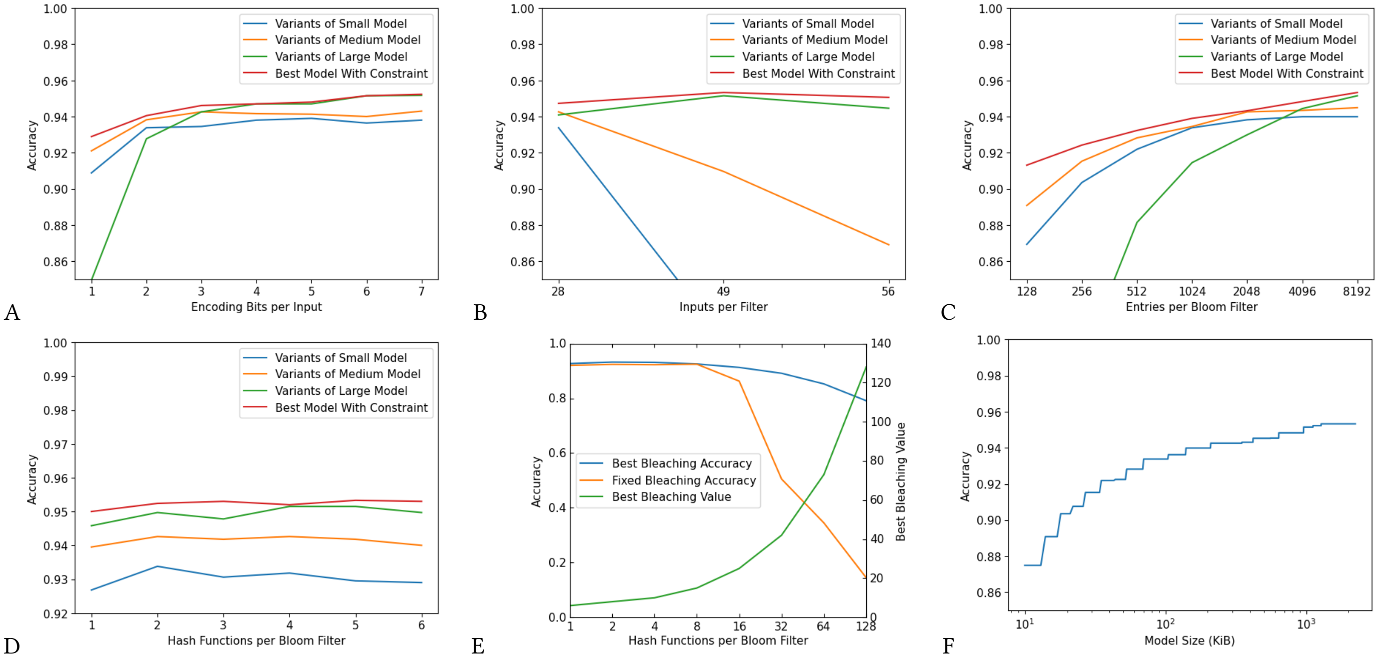

Figure 6: Sweeping results for MNIST with the configurations described in Table 2. (a) Accuracy versus the number of bits used to encode each input. (b) Accuracy versus the number of inputs to each Bloom filter. (c) Accuracy versus the number of entries in the LUTs of each Bloom filter. (d) Accuracy versus the number of distinct hash functions per Bloom filter. (e) Accuracy (shown on left y-axis) versus hash function with a fixed (b=16) and variable bleaching value. Right y-axis shows the best bleaching value. (f) The most accurate model which we could obtain under a given maximum model size.

to exceed 96% accuracy with reasonable model sizes, additional algorithmic improvements will be needed.

6 CONCLUSION

While most machine learning research centers around DNNs, we explore an alternate neural model, the Weightless Neural Network, for edge inference. We incorporate enhancements such as counting Bloom filters, inexpensive H3 hash functions and a Gaussian-based non-linear thermometer encoding into the WiSARD weightless neural model, improving state-of-the-art WNN MNIST accuracy [15] from 91.5% to 95.2%. The proposed BTHOWeN architecture is compared to state-of-the-art weightless models as well as MLPs and CNNs of similar accuracy. An FPGA accelerator for BTHOWeN is also presented. Compared to prior WNNs, BTHOWeN reduces error by 41% and model size by 51% across nine datasets. Compared to iso-accuracy MLP models, BTHOWeN consumes ~20% of the total energy while reducing latency by ~85%. Energy/latency improvements over CNNs are even larger, although CNNs have higher accuracy.

There are many opportunities for future work in this domain. There are algorithmic improvements we would like to explore, including weightless convolutional neural networks, better input remapping, and converting pretrained DNNs to WNNs. Preliminary experiments suggest that backpropagation-based training approaches can significantly improve WNN model accuracy, making them feasible for broader applications.

We believe that WNNs hold substantial promise for inference on the edge. While WNNs have historically trailed in accuracy to DNNs, algorithmic improvements such as bleaching demonstrate that accuracy and efficiency can be greatly improved by enhanced architectures and training techniques. The potential latency and energy efficiency benefits that can be obtained through WNNs warrant further research in this area.

ACKNOWLEDGMENTS

This research was supported in part by Semiconductor Research Corporation (SRC) Task 3015.001/3016.001, National Science Foundation grant number 1763848, Fundação para a Ciência e a Tecnologia, I.P. (FCT) [ISTAR Projects: UIDB/04466/2020 and UIDP/04466/2020], CAPES - Brazil - Finance Code 001, CNPq grant number 310676/2019-3, and FCT/COMPETE/FEDER, FCT/CMU IT Project FLOYD: POCI-01-0247-FEDER-045912. Any opinions, findings, conclusions or recommendations are those of the authors and not of the funding agencies.

REFERENCES

[1] Stefan Aeberhard. [n. d.]. Wine Data Set. https://archive.ics.uci.edu/ml/datasets/wine

[2] Igor Aleksander, Massimo De Gregorio, Felipe França, Priscila Lima, and Helen Morton. 2009. A brief introduction to Weightless Neural Systems. In 17th European Symposium on Artificial Neural Networks (ESANN). 299–305.

[3] I. Aleksander, W.V. Thomas, and P.A. Bowden. 1984. WISARD·a radical step forward in image recognition. Sensor Review 4, 3 (1984), 120–124. https://doi.org/10.1108/eb007637

[4] Austin Appleby. 2016. MurmurHash3. https://github.com/aappleby/smhasher.

[5] Alan Bacellar, Brunno Goldstein, Victor Ferreira, Leandro Santiago, Priscila Lima, and Felipe França. 2020. Fast Deep Neural Networks Convergence using a Weightless Neural Model. In ESANN.

[6] Younes Ben Mazziane, Sara Alouf, and Giovanni Neglia. 2022. A Formal Analysis of the Count-Min Sketch with Conservative Updates.

[7] Burton H. Bloom. 1970. Space/Time Trade-Offs in Hash Coding with Allowable Errors. Commun. ACM 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[8] M Breternitz, Gabriel H Loh, Bryan Black, Jeffrey Rupley, Peter G Sassone, Wesley Attrot, and Youfeng Wu. 2008. A segmented bloom filter algorithm for efficient predictors. In 2008 20th International Symposium on Computer Architecture and High Performance Computing. IEEE, 123–130.

[9] 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. 2016. Financial credit analysis via a clustering weightless neural classifier. Neurocomputing 183 (2016), 70–78. doi:10.1016/j.neucom.2015.06.105

[10] Hugo Carneiro, Carlos Pedreira, Felipe França, and Priscila Lima. 2018. The exact VC dimension of the WiSARD n-tuple classifier. Neural Computation (11 2018), 1–32. https://doi.org/10.1162/neco_a_01149

[11] J.Lawrence Carter and Mark N. Wegman. 1979. Universal classes of hash functions. J. Comput. System Sci. 18, 2 (1979), 143–154. https://doi.org/10.1016/0022-0000(79)90044-8

[12] Danilo Carvalho, Hugo Carneiro, Felipe França, and Priscila Lima. 2013. B-bleaching : Agile Overtraining Avoidance in the WiSARD Weightless Neural Classifier. In ESANN.

[13] Jason Catlett. [n. d.]. Statlog (Shuttle) Data Set. https://archive.ics.uci.edu/ml/datasets/Statlog+(Shuttle)

[14] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. arXiv:1602.02830 [cs.LG]

[15] Leandro Santiago de Araújo, Letícia Dias Verona, Fábio Medeiros Rangel, Fabrício Firmino de Faria, Daniel Sadoc Menasché, Wouter Caarls, Maurício Breternitz, Sandip Kundu, Priscila Machado Vieira Lima, and Felipe Maia Galvão França. 2019. Memory Efficient Weightless Neural Network using Bloom Filter. In ESANN.

[16] David Deterding, Mahesan Niranjan, and Tony Robinson. [n. d.]. Connectionist Bench (Vowel Recognition - Deterding Data) Data Set. https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+(Vowel+Recognition+-+Deterding+Data)

[17] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A Reliable Randomized Algorithm for the Closest-Pair Problem. Journal of Algorithms 25, 1 (1997), 19–51. https://doi.org/10.1006/jagm.1997.0873

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

[19] Victor C. Ferreira, Alexandre S. Nery, Leandro A. J. Marzulo, Leandro Santiago, Diego Souza, Brunno F. Goldstein, Felipe M. G. França, and Vladimir Alves. 2019. A Feasible FPGA Weightless Neural Accelerator. In 2019 IEEE International Symposium on Circuits and Systems (ISCAS). 1–5. https://doi.org/10.1109/ISCAS.2019.8702797

[20] Dave Fick and Mike Henry. 2018. Mythic AI's Presentation at Hot Chips 2018, Accessed November 22, 2021. (2018). https://www.mythic-ai.com/mythic-hot-chips-2018/

[21] R.A. Fisher. [n. d.]. Iris Data Set. https://archive.ics.uci.edu/ml/datasets/iris

[22] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. 2021. A Survey of Quantization Methods for Efficient Neural Network Inference. arXiv:2103.13630 [cs.CV]

[23] Kiran Gopinathan and Ilya Sergey. 2020. Certifying Certainty and Uncertainty in Approximate Membership Query Structures. In Computer Aided Verification, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 279–303.

[24] Bruno P. A. Grieco, Priscila M. V. Lima, Massimo De Gregorio, and Felipe M. G. França. 2010. Producing Pattern Examples from "Mental" Images. Neurocomput. 73, 7-9 (March 2010), 1057–1064. https://doi.org/10.1016/j.neucom.2009.11.015

[25] Daniel Kang. 2021. Accelerating Queries over Unstructured Data with ML. In CIDR.

[26] Andressa Kappaun, Karine Camargo, Fabio Rangel, Fabrício Firmino, Priscila Machado Vieira Lima, and Jonice Oliveira. 2016. Evaluating Binary Encoding Techniques for WiSARD. In 2016 5th Brazilian Conference on Intelligent Systems (BRACIS). 103–108. https://doi.org/10.1109/BRACIS.2016.029

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

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

[29] Warren S. McCulloch and Walter Pitts. 1943. A Logical Calculus of the Ideas Immanent in Nervous Activity. The Bulletin of Mathematical Biophysics 5, 4 (1943), 115–133. https://doi.org/10.1007/bf02478259

[30] Pete Mowforth and Barry Shepherd. [n. d.]. Statlog (Vehicle Silhouettes) Data Set. https://archive.ics.uci.edu/ml/datasets/Statlog+(Vehicle+Silhouettes)

[31] Kenta Nakai. [n. d.]. Ecoli Data Set. https://archive.ics.uci.edu/ml/datasets/ecoli

[32] Richard Rohwer and Michal Morciniec. 1998. The Theoretical and Experimental Status of the *n*-Tuple Classifier. Neural Netw. 11, 1 (Jan. 1998), 1–14.

[33] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. International Journal of Computer Vision 115 (4 2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[34] Arish S., Sharad Sinha, and Smitha K.G. 2019. Optimization of Convolutional Neural Networks on Resource Constrained Devices. In 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). 19–24. https://doi.org/10.1109/ISVLSI.2019.00013

[35] 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. 2020. Weightless neural networks as memory segmented bloom filters. Neurocomputing 416 (2020), 292–304.

[36] Simone. 2020. TinyML or Arduino and STM32: Convolutional Neural Network (CNN) Example, Accessed Nov 22, 2021. (2020). https://eloquentarduino.github.io/2020/11/tinyml-on-arduino-and-stm32-cnn-convolutional-neural-network-example/

[37] David J. Slate. [n. d.]. Letter Recognition Data Set. https://archive.ics.uci.edu/ml/datasets/letter+recognition

[38] Ashwin Srinivasan. [n. d.]. Statlog (Landsat Satellite) Data Set. https://archive.ics.uci.edu/ml/datasets/Statlog+(Landsat+Satellite)

[39] Vivienne Sze, Hsin Yu-Chen, Ju Tien-Yang, and Joel Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. https://arxiv.org/pdf/1703.09039.pdf

[40] V.N. Vapnik and A.Y. Chervonenkis. 2015. On the uniform convergence of relative frequencies of events to their probabilities. Springer, 11–30. https://doi.org/10.1007/978-3-319-21852-6_3

[41] Iuri Wickert and Felipe França. 2002. Validating an unsupervised weightless perceptron. In Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP '02. 537 – 541 vol.2. https://doi.org/10.1109/ICONIP.2002.1198114

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