Designing Efficient Machine Learning Architectures for Edge Devices

by

Tianen Chen

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Date of final oral examination: 12/01/2023

The dissertation is approved by the following members of the Final Oral Committee:
Younghyun Kim, Assistant Professor, Electrical and Computer Engineering
Azadeh Davoodi, Professor, Electrical and Computer Engineering
Umit Ogras, Associate Professor, Electrical and Computer Engineering
Suman Banerjee, Professor, Computer Sciences

CONTENTS

ABSTRACT

Machine learning has proliferated on many Internet-of-Things (IoT) applications designed for edge devices. Energy efficiency is one of the most crucial constraints in the design of machine learning applications on IoT devices due to battery and energy-harvesting power sources. Previous attempts use the cloud to transmit data back and forth onto the edge device to alleviate energy strain, but this comes at a great latency and privacy cost. Approximate computing has emerged as a promising solution to bypass the cloud by reducing the energy cost of secure computation on-device while maintaining high accuracy and low latency. Within machine learning, approximate computing can be used on overparameterized deep neural networks (DNNs) by removing the redundancy by sparsifying the network connections. This thesis attempts to leverage approximate computing techniques on the hardware and software-side of DNNs in order to port onto edge devices with limited power supplies. This thesis aims to implement reconfigurable approximate computing on low-power edge devices, allowing for optimization of the energy-quality tradeoff depending on application specifics. These objectives are achieved by three tasks as follows: i) hardware-side memory-aware logic synthesization, ii) designing energy-aware model compression techniques, and, iii) optimizing edge offloading techniques for efficient client and server communication. These contributions will help facilitate the efficient implementation of edge machine learning on resource-constrained embedded systems.

1 INTRODUCTION

IoT devices have proliferated in the past few years, with devices using machine learning as a way to interact with data collected on the edge. Mobile devices on the edge rely on unstable energy source from either battery or energy harvesting, which means that resource allocation must be carefully considered. Primarily, the millions of multiply-and-accumulate (MAC) operations must be carefully considered, as they are the most costly operation within neural network implementation. Cloud computing has been considered in order to offload computations from the edge device to the cloud. However, with cloud computing, latency and privacy is a concern due to constant data transmission.

This thesis aims to use the emerging paradigm of *approximate computing* to perform inference directly on the edge device and perform more efficient edge offloading than previous works. In this thesis, we further the advancement of approximate computing techniques for neural networks by combining both software- and hardware-side approximate computing techniques in order to close the gap between large server-scale machine learning and mobile edge device machine learning.

Approximate computing in this thesis will operate under a balancing act between energy and quality, with application-specific reconfigurability allowing for fine-grain adjustment of the energy-quality tradeoff. In particular, approximate computing techniques for neural networks in this thesis will focus on i) model storage size reduction, ii) latency reduction, and iii) memory access reduction. First, this thesis first aims to reduce the memory accesses via memory-aware logic synthesization [7]. We consider methods that eliminate memory accesses from heavy MAC operations which will lead to energy consumption reduction. Furthermore, we consider methods that simultaneously reduce the model size with energy-aware model compression techniques [6]. Lastly, we want to optimize our approximation methods by developing optimized edge offloading techniques for tightly coupled machine learning tasks.

The research goals of this thesis will be carried out through the following research thrusts.

- Designing memory-aware logic synthesization
- Designing energy-aware model compression
- Optimizing edge offloading

The rest of the thesis is organized as follows: Section 2 introduces three major thrusts of this thesis to realize energy-efficient approximate computing for embedded systems which includes the required background and prior works. Sections 3, 4, 5 describe the research works. Finally, we summarize the document in Section 6.

2 BACKGROUND

With the end of Moore's law approaching, engineers are seeking new paradigms to extract performance gains for the new generation of integrated circuits [60]. This holds especially true with IoT and embedded systems which demand performance and accuracy despite being hampered by power constraints. However, by examining the applications of these power-constrained devices, we notice that many applications do not necessarily need a fully-precise design. In fact, power intensive applications such as machine learning or image/video processing margins of error that do not degrade the overall quality for the end user [30, 55].

Approximate computing prioritizes low-power computing by relaxing accuracy requirements for applications that do not demand exact computation results [19]. Neural networks have proven to be a particularly error-resilient application that approximate computing can target. The self-healing nature of neural networks allows for significant approximation while maintaining an acceptable inference output. Outputs of neural networks are probabilistic in nature, with output probability percentage degradation still producing accurate inference in a majority of use cases. At the hardware level, different approximate computing techniques have been proposed, with approximate arithmetic units, load-value prediction, and voltage scaling included as potential solutions to heavy computational overhead [27]. At the software level, loop perforation and lossy compression are examples of algorithm approximation [56, 53]. Additionally, we have neural network specific designs that include memory access skipping and logarithmic approximate computing [71, 33].

In order to relax these accuracy constraints, approximate computing is used to find the optimal energy-accuracy trade-off [27, 64]. Approximate computing relies on the concept of sparsifying the system design and eliminating redundancy such that the appliation-defined or user-defined threshold of error tolerance is not breached [66, 71, 43, 61]. Within each of these works is i) an approximation method and ii) an optimization method that balances the energy-accuracy trade-off. The following research thrusts are novel methods which generate optimum approximate computing methods by exploiting both sparsification and redundancy within the system design.

3 MEMORY-AWARE LOGIC SYNTHESIZATION

The first approximate computing scheme we propose is memory-aware logic synthesization. In the following section, we introduce a new hardware-based technique that reduce memory accesses by implementing approximate computing techniques at a hardware-level.

Related Works

Researchers have proposed various methods to save energy in neural processing by exploiting its intrinsic error resilience, sparsity, and massive parallelism at different levels of the system stack, but the common goal has been reducing memory access. Examples include quantization [25], pruning [4, 35], and model compression [20]. Processing-in-memory (PIM) also reduces memory access by performing analog computing in or near specially designed memory without fetching the weights into the NPE [12, 10]. The feasibility of realization of NNs as combinational logic has been proven in recent work [11, 44, 52], but the problem of scaling the large logic size is yet to be addressed. At a hardware level, reserachers have proposed using shift and add operations to approximate multiplications such as in [58, 41], along with a similar approach using look-up tables such as in [51, 47].

Introduction

Reducing memory access is the core of realizing fast and efficient neural networks (NNs). In conventional neural processing element (NPE)-based NNs, frequent multiply-and-accumulate (MAC) operations incur heavy memory access overhead for fetching weight parameters and storing intermediate outputs, which is the primary source of latency and energy consumption [12]. An emerging hardware-oriented solution to this challenge is combinational logic NN (CLNN), where the inputs and outputs of the neurons are binarized and represented as *arbitrary Boolean functions* mapped to combinational logic circuits or look-up tables (LUTs) [52, 11, 44, 59, 62]. The evaluation of such hardware does not involve any memory access other than fetching the inputs and storing the final outputs, and hence is extremely faster than equivalent binarized MAC on NPEs. It makes CLNNs attractive and suitable for low-latency, fixed-function applications, such as in-sensor inference [52] and network intrusion detection [59].

The low latency of CLNNs, however, is powered by massively parallel hardware resources that cost significant area occupancy and hence energy consumption. The key to the realization of area- and energy-efficient CLNNs is to exploit its intrinsic redundancy and error resilience like in other NN optimization methods (e.g., pruning and quantization) in every design step including logic-level optimization. Unfortunately, conventional logic optimization methods that strictly preserve original input-to-output mapping are not able to capture and exploit the redundancy and error resilience, and thus are far from effective when optimizing CLNNs. As recognized in [48], this gap between machine learning and logic optimization is yet to be resolved, and should be addressed for more widespread adoption of CLNNs.

In this section, we propose a novel CLNN design method called SynthNet for bridging this gap and pushing the limit of CLNN adoption for high-throughput and low-power applications. Our techniques improve upon the scalability of CLNNs by proposing minimization techniques that allow for large scale networks to be compressed to synthesizable circuits. SynthNet exploits the intrinsic error resilience of NNs in order to selectively remove or replace Boolean mapping functions and thereby significantly reduce the logic circuit size. By judiciously *over-minimizing* the truth tables of neuron mapping functions based on the significance of each mapping with the awareness of neuron activation properties, the circuit implementation of the resultant truth tables is reduced by orders-of-magnitude than that of the original truth tables. This design method also boosts the accuracy by suppressing inference errors induced by random output mapped to input combinations unseen during training, which is a unique hazard in CLNNs.

The contributions of this part of the project are summarized as follows:

- We analyze activation of neurons realized as a logic circuit and identify opportunities to further minimize the logic size beyond what traditional logic optimization methods can achieve, in order to fully exploit the error resilience of NNs.
- Based on the analysis, we present two very effective CLNN optimization techniques: i) synthesis-aware pruning and ii) input-driven neural logic minimization. We explore the energy-accuracy trade-offs of the logic optimization and present an efficient and more scalable design framework to determine the optimal implementation that meets a given accuracy constraint.
- We evaluate SynthNet for a CLNN using the CIFAR-10 dataset [31]. Our method reduces the energy consumption per image by 90–99% compared to a systolic array-based architecture, while maintaining 82% accuracy, yet to be achieved by CLNN-only implementations on the CIFAR-10 dataset.

Combinational McCulloch-Pitts Neural Networks

The McCulloch-Pitts neuron model, which has binary inputs and a binary output, is an ideal model for CLNN implementation since combinational logic can implement any arbitrary binary mapping. A McCulloch-Pitts includes/proposed/figures/neuron_2.pdf

Figure 3.1: Combinational logic implementation of a McCulloch-Pitts neuron.

neuron's function can is defined as follows:

$$y = \begin{cases} 1 & \text{if } \sum_j x^j w^j \geqslant b \\ 0 & \text{otherwise} \end{cases}, \tag{3.1}$$

where y is the output of the neuron, x^j and w^j respectively are the j-th input and weight, and b is the bias of the neuron. Unlike binarized NNs that binarize everything including weights (e.g., XNOR-based NNs), w_j can be a high-precision floating-point value, which are desirable for high accuracy [44]. Boolean logic circuit is suitable for implementing this neuron model because the inputs and output are binary and their mapping requires the ability to realize arbitrary Boolean functions.

A trained McCulloch-Pitts neuron can be implemented as a logic circuit as illustrated in Fig. 3.1. If the number of inputs is small, outputs can be defined for all possible input combinations based on (3.1) to build the truth table of a completely specified function (CSF) with no don't-care (DC) output. In practical NNs, however, the number of inputs is much grater than three, making it impossible to enumerate outputs for all input combinations. Alternatively, defining the Boolean function as an incompletely specified function (ISF), where the output is specified only for a subset of input combinations and the rest are left as DC, can greatly reduce the enumerated outputs since only a small subset of input combinations are seen during training and inference [44]. Finally, logic minimization and synthesis is followed to implement the ISFs as logic circuits.

In this work, the straight-through estimator in the form of the hard hyperbolic tangent (tanh) linear activation function is used as in [25], in order for the binarized neurons to be able to successfully update gradients within the back-propagation algorithm when the derivative is zero everywhere. We substitute the classical dropout regularization technique with a random binarization probability. Activations are binarized with a includes/proposed/figures/hitcount_bar_chart_and_scatter.pdf

Figure 3.2: (a) Histogram of row hit counts greater than 0 in the training set. (b) Correlation between row hit counts between the training set and the test set. (Range limited for better visualization.)

certain specified probability. Stochastically binarizing activations ensures our gradients update during back-propagation, derived from the variant of dropout in [25].

Motivation: Unexploited Error Resilience

As mentioned above, conventional *precise* logic minimization and synthesis does not take advantage of the high error resilience of CLNNs. It leads to two limitations (and opportunities) in terms of energy efficiency and accuracy.

When input-to-output mapping does not *always* have to be precise, which is the case for CLNNs, *approximate* implementation of Boolean functions allows sharing not only exactly equivalent sub-circuits but also near-equivalent sub-circuits, resulting in a smaller logic size and thus high energy efficiency that precise implementation cannot achieve. For the approximate implementation of combinational neuron, we should exploit the property that the probability distributions of neuron inputs and outputs are not uniform.

Let us consider the VGG-like architecture in [57] of six convolutional layers followed by three fully-connected layers. As an example, to implement the second convolutional layer (3×3 convolution, 20 input channels, 20 output channels) as a logic circuit, we would need to build 20 truth tables of $3 \times 3 \times 20 = 180$ inputs (i.e., 2^{180} input combinations) of one output each. After building a truth table with 10,000 images from the training set, out of the 2^{180} rows, output is specified (as either 0 or 1) for only 1.1×10^6 rows on average across 20 output channels, and the output for the rest of the rows remains unspecified (DC). This corresponds to only $7.4 \times 10^{-47}\%$ of total maximum possible rows. More importantly, some rows are seen more frequently than other rows, implying that not all rows are equally important. As a metric of the importance of rows, we define *hit count*, HC, which refers to the number of occurrences that the row's input combination is seen during training or inference. Fig. 3.2(a) shows the histogram of the row hit counts greater than 0 (i.e., excluding DC rows with $\text{HC} = 0$) after training. We can see that 51% of the rows are hit (i.e., the corresponding input is seen) only once, and only 11% of the rows are hit 10 or more times. Furthermore, there exists a very high correlation between the hit counts of the training set and the test set (10,000 images) as shown in Fig. 3.2(b), which suggests that logic optimization based on training set will work as well for inference.

The input combinations defined in the ISFs from the training set cover the most of the input combinations of the test set, but not all. In the same example above, about 14% of the input combinations of the test set are not seen in the training set. Since the output for such input combinations is set to DC in the ISFs, an output that violates (3.1) may be mapped during synthesis, which becomes a source of accuracy loss. This is a unique hazard of CLNNs that does not exist in NPE-based NNs where outputs for unseen inputs are still correctly computed based on the loaded weights.

In order to mitigate the problem, we need to minimize the accuracy loss due to unspecified outputs. This could be achieved by increasing the possibility that the output is specified for given input combinations, i.e., increasing the *hit rate*, which is defined as the percentage of the input combinations that are seen during both the training and inference. However, specifying more output for unseen rows is not a viable option since it will increase the logic complexity. Rather, introducing DCs in the inputs will increase the hit rate because the total number of unseen input combinations is reduced, and as a result more *generalized* truth tables will be generated. This is similar to pruning of conventional NNs in that it requires a judicious choice of inputs to be ignored. In CLNNs, this is an opportunity for boosting accuracy by preventing *unknown outputs* during training (which does not happen in conventional NN training).

Design Optimization of CLNNs

Based on the intuitions discussed in the previous sections, we present a design optimization method of CLNNs called SynthNet, focusing on logic minimization. Specifically, we propose two complementary techniques, includes/proposed/figures/designflow.pdf

Figure 3.3: SynthNet's fully automated design optimization of CLNNs. The dashed box represents the territory of the proposed logic optimization.

synthesis-aware pruning and input-driven neural logic minimization, to address the above-mentioned limitations and exploit the error resilience of CLNNs for improving energy efficiency.

Design Flow

The overall design flow is presented in Fig. 3.3. We first train a McCulloch-Pitts NN and convert target layers into ISF truth tables. Our CLNN optimization is performed before the conventional logic minimization and synthesis of the truth tables, as highlighted by the dashed box in the figure. We split the multi-input multi-output truth tables into multi-input single-output truth tables to be optimized independently. The truth tables are then sent through our logic optimization procedure composed of iterative synthesis-aware pruning and input-driven neural logic minimization until it reaches a given target accuracy. Accuracy is evaluated on the test set using the reduced truth tables. Finally, the reduced truth tables are implemented as logic circuits through conventional logic minimization and synthesis. The synthesized circuit is evaluated for hardware metrics, and the test set is applied on the circuit to get the actual accuracy. The following two subsections respectively describe the synthesis-aware pruning and the input-driven neural logic minimization, followed by the integration of both in the design flow.

Synthesis-aware Pruning

Pruning, in general, removes low-magnitude weights that contribute little to the model output. In CLNNs, weight pruning is equivalent to removing an input from the truth table, or placing DC on the input to be removed. This serves two benefits. First, the truth table size (hence the circuit complexity) decreases exponentially as the number of inputs decreases. Second, the hit rate increases because some DC outputs, which would have mapped to a random output, are now specified based on remaining more significant weights, leading to accuracy improvement. There is a point of diminishing returns in accuracy improvement because beyond a certain point, accuracy loss due to over-generalization becomes greater than the accuracy gain due to the reduction of DC outputs.

Specifically, for a given ISF truth table, we gradually remove inputs (introduce DCs) beginning with the corresponding lowest-magnitude weights. We denote the percentage of removed inputs by *pruning degree*

Δ_p. For example, in the 3×3 convolutional layer where there are 20 input channels, we can remove up to 180 inputs. If $\Delta_p = 90\%$, 162 lowest-weight inputs will be removed, reducing the maximum of input combinations from 2^{180} to 2^{18} . The input removal results in multiple rows with different outputs mapping to the same row with the same output in the reduced truth table. In order to determine the new output, we take a weighted average of the inputs mapped together under the DCs to take the importance of each row into account for the new output y' as follows:

$$y' = \text{round}\left(\frac{\sum_{i \in I} y_i \times HC_i}{\sum_{i \in I} HC_i}\right), \tag{3.2}$$

where I is the set of inputs that are merged, y_i is the output and HC_i is the hit count of the i-th row that is merged. The hit count of the new rows, HC' is the sum of the hit counts of the merged rows. That is,

$$HC' = \sum_{i \in I} HC_i. \tag{3.3}$$

Consider an example shown in Fig. 3.4. The second input, x_2, has the lowest weight of 1, so we consider pruning it. Pruning a single input will cause pairs of inputs to respectively map to a single input. For example, consider the two input combinations $x_1x_2x_3 = 101$ and $x_1x_2x_3 = 111$. In the original truth table, their outputs differ as 0 and 1, respectively, but in the reduced truth table, they both map to $x_1x_3 = 11$, and the new output is round $\left(\frac{0\times10+1\times1000}{10+1000}\right) = 1$. The hit count of the new rows is now $10 + 1000 = 1010$.

Two input combinations $x_1x_2x_3 = 000$ and $x_1x_2x_3 = 010$ show how pruning improves accuracy. The first input combination, 000, is seen during training, and its output is specified as 0 in the original truth table. On the other hand, the second input combination, 010, is not seen during training, and its output is not specified. If this DC output is mapped to 1 during synthesis, it would become a source of inference error because it `includes/proposed/figures/minimization1.pdf`

Figure 3.4: Synthesis-aware pruning. In this example, $\Delta_p = 33.3\%$, and hence x_2 is removed.

includes/proposed/figures/minimization2.pdf

Figure 3.5: Input-driven logic minimization (row dropping). In this example, $\Delta_h = 40\%$, and output for four input combinations, 100, 101, 110, and 111, are unspecified.

violates (3.1). Pruning x_2 effectively specifies the output of 010 as 0, which is the correct output if it had been seen during training.

Input-driven Neural Logic Minimization

As discussed in the motivation sections, the hit counts of the rows varies significantly, and the majority of input combinations appear only a few times during training. These low-hit count rows contribute little to the

CLNN accuracy, as compared to high-hit count rows, and can be removed from the ISF truth table as if they have not appeared during training. This is done by unspecifiying the output, i.e., making 0 or 1 to DC, and the row is called *dropped* from the table. As we drop seldom-hit inputs and introducing more DC outputs, we can reduce the complexity of the synthesized circuit.

Specifically, our input-driven neural logic minimization, or simply *row dropping*, unspecifies the output for the rows with the lowest non-zero hit counts until the total hit counts of dropped rows reaches a *row dropping degree* Δ_h. We do not set the hit count of the dropped rows to zero because the hit counts should be preserved for making decisions in the following iterations of pruning. Fig. 3.5 shows an example of row dropping for $\Delta_h = 40\%$. Since the total hit count is 3165, we drop low-hit count rows until the sum of the hit counts of the dropped row reaches 1266. In this case, four rows $x_1x_2x_3 = 100$ $x_1x_2x_3 = 101$, $x_1x_2x_3 = 110$, and $x_1x_2x_3 = 111$ have the lowest non-zero hit counts, whose sum is $5+10+150+1000 = 1165$. Therefore, the four rows are dropped by unspecifying their output, but their hit counts, 1165 in total, are preserved.

Pruning and Row Dropping Thresholds

Determining the two thresholds, pruning degree Δ_p and row dropping degree Δ_h, can be time-consuming since the design space is large, and logic synthesis for accuracy evaluation takes a long time. We propose a variant of coordinate descent to determine the two thresholds to minimize the logic size while meeting an accuracy constraint c without time-consuming exhaustive search. Also, we estimate the accuracy of the resulting NN using the reduced ISF truth table without synthesizing it.

The threshold optimization procedure is described in Algorithm 1. Initially, Δ_p is set to 0% and Δ_h is set to 0%, i.e., no pruning and no row dropping is applied. For larger networks, we can start Δ_h and Δ_p at a higher number. In the Prune procedure, we first search along the space of Δ_p by gradually increasing it by a granularity of δ_p until the accuracy reaches the maximum. During this procedure, new accuracy $\textsc{acc}(\Delta_p^{new}, \Delta_h)$ is compared to the previous step's accuracy, $\textsc{acc}(\Delta_p^{old}, \Delta_h)$. At the maximum accuracy, we fix Δ_p and move on to the rowDrop procedure where we gradually increase Δ_h by a granularity of δ_h until the accuracy hits the user-defined accuracy constraint c. This is repeated until increasing Δ_p further does not improve accuracy and increasing Δ_h further causes the accuracy to drop below c. To reduce computational complexity, we start at a coarse granularity for the initial search and gradually increase the granularity. As a result, the coordinate descent algorithm's iterative procedure repeats the two-dimensional search and returns the optimal Δ_p and Δ_h that satisfy the given accuracy constraint.

In each iteration during coordinate descent for pruning, $\textsc{acc}()$ function estimates the accuracy of a candidate NN, which would require time-consuming synthesis and simulation of its logic implementation. In order to reduce execution time, we estimate the accuracy by keeping the ISFs as lookup tables. Instead of time-consuming synthesis and simulation, these lookup tables can be used to quickly estimate the output of the circuit. Because the truth tables are incomplete, some table lookups may fail when an input with unspecified output is encountered. Upon the completion of Algorithm 1 for each layer, all layers are integrated and synthesized for the evaluation of accuracy and power consumption.

Results and Discussion

In this section, we demonstrate the efficacy of SynthNet for energy-efficient implementation of CLNN.

We consider a small CNN model that is suitable for low-latency in-sensor image recognition as shown in Table 3.1. Note that, while small, this network is the deepest and most computationally complex network

Algorithm 1 CLNN logic optimization

```
1: procedure OPTIMIZE($c$)
2:     initialize $\delta_p$ and $\delta_h$
3:     let $\Delta_p^{old} = 0$, $\Delta_h^{old} = 0$, $\Delta_p^{new} = \delta_p$, $\Delta_h^{new} = \delta_h$
4:     while $\Delta_p^{new} > \Delta_p^{old}$ or $\Delta_h^{new} > \Delta_h^{old}$ do
5:         decrease $\delta_p$ and $\delta_h$
6:         $\Delta_p^{old} = \Delta_p^{new}$, $\Delta_h^{old} = \Delta_h^{new}$
7:         $\Delta_p^{new} = $ PRUNE($\Delta_p^{old}, \delta_p, \Delta_h^{old}$)
8:         $\Delta_h^{new} = $ ROWDROP($\Delta_h^{old}, \delta_h, \Delta_p^{new}, c$)
9:     end while
10:    return $\Delta_p^{old}$, $\Delta_h^{old}$
11: end procedure
12: procedure PRUNE($\Delta_p^{old}$, $\delta_p$, $\Delta_h$)
13:    while ACC($\Delta_p^{new}, \Delta_h$) > ACC($\Delta_p^{old}, \Delta_h$) do
14:        $\Delta_p^{old} = \Delta_p^{new}$, $\Delta_p^{new} = \Delta_p^{old} + \delta_p$
15:    end while
16:    return $\Delta_p^{old}$
17: end procedure
18: procedure ROWDROP($\Delta_h^{old}$, $\delta_h$, $\Delta_p$, $c$)
19:    while ACC($\Delta_p, \Delta_h^{new}$) $> c$ do
20:        $\Delta_h^{old} = \Delta_h^{new}$, $\Delta_h^{new} = \Delta_h^{old} + \delta_{hit}$
21:    end while
22:    return $\Delta_h^{old}$
23: end procedure
```

yet to be integrated with ultra-low latency CLNNs. The model is trained on the CIFAR-10 image dataset using Google Tensorflow.

Since making early layers energy-efficient is more effective [45], we apply the proposed method to the second and third convolutional layers. However, the applicability is not limited to any specific layer type and is applicable to any binary-input binary-output layers. The other layers are binarized as well and processed by NPEs.

We use batch normalization after every layer, a batch size of 128, and the last 5000 samples of the training set as a validation set with the test

Table 3.1: VGG-like NN architecture with 32×32 input map used in the experiments.

Layer	Type	Max pool 2×2	Dropout
1	Binarized 3×3 Conv (3,20)		
2	Binarized 3×3 Conv (20,20)	✓	✓
3	Binarized 3×3 Conv (20,40)		
4	Binarized 3×3 Conv (40,40)	✓	✓
5	Binarized 3×3 Conv (40,80)		
6	Binarized 3×3 Conv (80,80)	✓	✓
7	Fully connected (80,1024)		✓
8	Fully connected (1024,1024)		✓
9	Fully connected (1024,10)		

error rate reported on the best validation accuracy after 200 epochs, similar to [25]. We do not retrain on the validation set and use only the images in the training set, excluding the validation set, to build the truth tables. Therefore, the truth tables are built on a completely separate set of images than the images on which we test the classification accuracy. All accuracy numbers reported in this section are obtained by simulating synthesized circuits, not estimated by the ACC() function.

All results for CLNN are synthesized with Synopsys Design Compiler using the TSMC 45 nm library. Every circuit representing individual truth tables needs cycles according to the height $\times$ width so that the whole input is convolved. For example, the input to Layer 2 is 32×32, thus it takes 1024 cycles of the combinational logic to convolve the entire input. Note that one can exchange latency for area by instantiating more combinational circuits to reduce the number of cycles. As the baseline, we use SCALE-Sim [54] and DRAMPower [5] to estimate the energy consumption for memory accesses in systolic array-based architecture with the parameters in [9].

Figs. 3.6 and 3.7 show the change in hit rate and classification accuracy for varying Δ_p and Δ_h, respectively. In Fig. 3.6, we can see that hit rate increases as Δ_p increases. As a result, the accuracy increases up to 86.4% at $\Delta_p = 83.3\%$ in Layer 2 and 86.9% at $\Delta_p = 89.4\%$ in Layer 3, which is found by Algorithm 1. Beyond these Δ_p, combinational circuitry is coalescing too many inputs and making the output too generic. Figs. 3.7(a) and 3.7(b) shows that the hit rate decreases as Δ_h increases for Layers 2 and 3, while Δ_p is set to the respective optimal value. As a result, the classification accuracy also drops, but only by a few percent even at $\Delta_h = 90\%$, i.e., when only 10% of the input combinations from training are preserved. Fig. 3.7(c) shows the hit rate of Layers 2 and 3 and the accuracy after combining both layers with the same Δ_h. As a result of row dropping in subsequent layers, the accuracy is slightly decreased, but it is still as high as 80% even when 90% of the rows are dropped in each layer.

Finally, we evaluate the CLNN in comparison to a conventional systolic array-based NN estimated using SCALE-Sim and DRAMPower. We set the target accuracy of our design to 82%, which is achieved at $\Delta_h = 50\%$ (see Fig. 3.7(c)). Pruning degree Δ_p is set to the respective optimal, 83.3% for Layer 2 and 89.4% for Layer 3. Compared to the systolic array-based NN architecture, the proposed CLNN consumes only a fraction of energy because there is no energy consumption for fetching weights from off-chip memory. We can also observe the greatest power reduction for layers with smaller input size but larger output channel width. For example, Layer 3 is convolving over a 16×16 input map after a max pooling layer, which means less cycles are needed to perform the full convolution. However, the weight fetching DRAM accesses' expenditure is much higher, regardless of input size, and is only dependent upon the width of the input and output channels. Therefore, layers with smaller input map sizes and larger width experience the greatest reduction in power consumption.

CLNNs are a promising approach to reduce the energy overhead of memory access in ultra low-latency NNs, but the large logic size has limited their application at scale in area- and energy-constrained applications. We presented a design optimization method for the energy-efficient implementation of CLNNs allowing for scalable synthesize of deeper NNs.

includes/proposed/figures/Prune.pdf

Figure 3.6: Synthesis-aware pruning with a varying pruning threshold Δ_p on (a) Layer 2 and (b) Layer 3.

includes/proposed/figures/rowDrop.pdf

Figure 3.7: Input-driven neural logic minimization with a varying row dropping threshold Δ_h on (a) Layer 2, (b) Layer 3, and (c) Layer 2 and 3 combined.

Exploiting the error-resilient nature of NNs, we proposed two techniques to over-simplify the input-to-output mapping of neurons and a coordinate descent-based optimization algorithm to determine the optimal level of simplification. Compared to the conventional systolic array-based NN, more than 90% power reduction is demonstrated per layer with an accuracy of 82% on CIFAR-10, yet to achieved by previous CLNNs.

4 ENERGY-AWARE MODEL COMPRESSION

The second approximate computing scheme we propose is energy-aware model compression. In the following section, we introduce a software-based model compression technique that optimizes upon previous model compression designs.

Introduction

Despite the unprecedented success of machine learning (ML), bringing intelligence to resource-constrained edge devices has not seen similar success. While neural network (NN) models are rapidly growing in complexity and size to serve more and more sophisticated applications, the gap between their compute requirements and the capabilities of edge devices has only been widening. Specifically, for edge ML, the limited storage and memory capacity has been identified as a major hindrance [39]. The recent emergence of binary neural networks (BNNs) has shed some light on the possibility of making ML models smaller, in which all the weights and activations are binarized to either +1 or -1 [25] Binarized weights and activations require less memory and storage than their full-precision (floating-point or integer) counterparts, and they can also be processed with simple, low-power bit-wise logic units instead of complex, power-hungry arithmetic units. It makes BNNs highly suitable for ML applications on edge devices with small memory and storage, and thin energy budget.

However, despite the dramatic size reduction of binarization, BNN models still require further compression to be ported onto more severely resource-constrained platforms. Compression methods have been proposed to further optimize BNNs, including computation skipping [16], and bit-level data pruning [38]. Weight pruning is a widely applicable model compression technique that removes unnecessary or unimportant weights from the network [21]. In traditional full-precision NN models, unnecessary or unimportant weights can be easily identified by their small magnitude during the forward pass. Removing such near-zero weights has only a minimal impact on the output accuracy, and, in fact, it can even reach a “sweet spot" in the model where accuracy can surpass the original unpruned model accuracy due to the reduction of overfitting from overparameterization of the model, in addition to performance gains due to reduced computations [20]. This form of unstructured pruning comes at a potential hardware overhead cost identifying the sparsity within the weight matrix. Structured pruning has advantages in generalization but lack the fine-grain control of individual weight connection pruning [69]. However, compression methods can overcome potential hardware overhead by employing methods through quantization, encoding, and weight permutation [20, 8]. Additionally a previous work in [17] has considered the communication latency of hardware when implementing full-precision networks onto the device. Some forms of extremely low-power networks, such as the combinational neural network, will require no hardware overhead when identifying sparsity within the matrix by simply removing the circuit component corresponding to the weight element [44, 7]. However, weight pruning for BNN models is not a straightforward problem since the magnitude of all weights is strictly 1, regardless of their sign, and thus magnitude cannot serve as an indicator of the weights’ importance. Therefore, BNN pruning requires a new significance metric to replace the weight magnitude.

In this section, we propose to use *latent weights* for pruning. Latent weights are real-valued weights that are used to obtain the pseudogradient vector during backpropagation as the real gradient vector cannot be obtained from binary weights [22]. We present a model compression technique that identifies the layer that has the greatest potential to improve the compression ratio at a time and prunes the layer based on the latent weights. The proposed technique includes an effective method to find the target layers based on the impact of pruning on the output accuracy without time-consuming model exploration. As a result, the proposed technique can achieve a dramatic reduction in model size and operation count, and reach the pareto-optimal of compressed networks that suffer no accuracy loss.

The contributions of this section are as follows:

- We present a latent weight-based pruning technique that selects layers that can be pruned with the minimum impact on the output accuracy and prune the layers based on latent weights.
- We introduce a multidimensional analysis of pruning layer-by-layer and include an optimization algorithm that intelligently minimizes a BNN that selectively prunes error-tolerant insensitive layers.
- We show experimental results that indicate a highly-optimized form of BNN pruning that decrease BOPs (binary operations) and model size by 46% and 27%, respectively, while incurring a small accuracy gain of +0.4% on the CIFAR-10 dataset, and similar results on other datasets. Our work is the first that can achieve a significant reduction in model size even without any accuracy loss.

Background and Related Work

The ever-increasing size of NN models not only poses a challenge to fast and energy-efficient processing, but is a major barrier to the deployment on devices with small memory and storage, which calls for effective solutions for efficient model compression and operation count reduction. In this section, we overview some key notions related to BNNs and BNN model compression.

Binary Neural Networks

The high error resilience of NNs allows for aggressive quantization for computation at reduced precision such as fixed-point or ternary weights instead of complex full precision [34, 28]. BNNs are an extreme case of quantized NNs, where weights and activations are restricted solely to two values, +1 or -1 [25]. This binarization leads to a simplification of multiply-and-accumulate (MAC) operations, which is the most fundamental but expensive aspect of the convolutional operation, to extremely simpler XNOR and popcount operations. This leads to a significant reduction in power consumption and model size. The complexity of a BNN is measured by the number of binary operations (BOPs), instead of the number of floating-point operations (FLOPs).

A key observation is that the derivative of the binarization function at all spots is zero or undefined, making backpropagation gradient calculation impossible. Therefore, the straight through estimator is used to allows the gradient to pass exactly as an identity, generating a *pseudogradient* [2, 25]. Also, having only binary values for weights, it is impossible to distinguish distinct magnitudes between the weights in BNNs. Thus, traditional magnitude-based weight pruning is ineffective, as there is no way to determine which weight affects classification accuracy more.

Flip Frequency-based Channel Shrinking

For the weight pruning of a BNN model, a new indicator of weight significance that substitutes the weight magnitude is required. A recent work has proposed to exploit the amount of weight flips (+1 to -1 or vice versa) that occur during training [37]. In this work, they conjecture that the weights can be determined as “unstable” if they flip frequently during training. Unstable weights are considered to have little contribution in the minimization of loss within the network. When the final stage of training is near (i.e., when the loss is stabilizing), the occurrence of flips is counted for each weight kernel as f. If the number of weight flips is above a predetermined threshold, the corresponding weight is determined as negligible and thus prunable. The portion of the prunable weights represents the portion of channels that can be potentially removed. Therefore, the number of channels is reduced by the portion of the prunable weights, and the entire BNN is retrained. This is repeated until the predetermined accuracy threshold has been reached.

Latent Weights in BNN

In a BNN model, the optimizer cannot directly compute the gradients required to update the weight kernels during backpropagation because the gradient of the sign function is zero almost everywhere. Therefore, a real-valued weight vector, $\tilde{w}$, is used instead of the binary weights for training [2, 25]. Also called the *latent weight* [22], it is used to calculate the pseudogradient during backpropagation. During the forward pass, the binarized weights, $w_{\mathtt{bin}}$, is simply the sign of the latent weight:

$$w_{\mathtt{bin}} = \operatorname{sign}(\tilde{w}) = \begin{cases} +1 & \text{if } \tilde{w} \geqslant 0 \\ -1 & \text{if } \tilde{w} < 0 \end{cases} . \tag{4.1}$$

The sign and magnitude can be thought of separately as follows [22]:

$$\tilde{w} = \operatorname{sign}(\tilde{w}) \cdot |\tilde{w}| =: w_{\mathtt{bin}} \cdot m, w_{\mathtt{bin}} \in \{-1, +1\}, m \in [0, \infty). \tag{4.2}$$

Since there now exists a magnitude value of the latent weight, m, different techniques typically reserved for floating-point models can now be applied to BNNs. Weights build inertia m over time. The higher the inertia, the stronger the gradient signal that is required in order to make the weight flip. Weights in the forward pass can only flip and not adjust their magnitude, unlike their floating-point counterparts. However, in the backpropagation stage, m for each latent weight can adjust during each training epoch, distinguishing individual weights in the kernel from one another. This real-valued vector allows for optimization methods to be applied to the BNN. Each BNN model that is trained contains the pseudogradient information along with latent weight information.

Proposed Latent Weight-based Pruning

We propose a new method to prune BNN models based on latent weights that dramatically reduces the model size and operation count, while maintaining accuracy. Specifically, we address the challenges in BNN pruning mentioned in Section 4: i) identify which layer should be pruned and determine how heavily it should be pruned, and ii) select weight kernels within the identified layer to be pruned.

Design Flow Overview

We first describe our pruning method in which a BNN model is pruned based on latent weights. Unlike flip frequencies [37] which are an "indirect" significance metric induced from the latent weights, latent weight-based pruning offers a more "direct" indicator of significance since the magnitude of the latent weight drives the inertia of the weight flipping. This enables us to use the source of weight kernel optimization, which offers additional granularity as we can tune pruning of real-valued weights.

The overall model optimization flow around the proposed latent weight-based pruning is presented in Figure 4.1. During training, we initialize all pruning percentages from zero and begin iterative pruning. From zero, we begin pruning on the least accuracy-responsive layer by increasing the pruning percentage on each layer iteratively. We prune each layer to a predefined accuracy threshold and prune the next least accuracy-responsive layer afterward. Our iterative pruning ends when we no longer can prune and maintain accuracy above the threshold. The following subsections

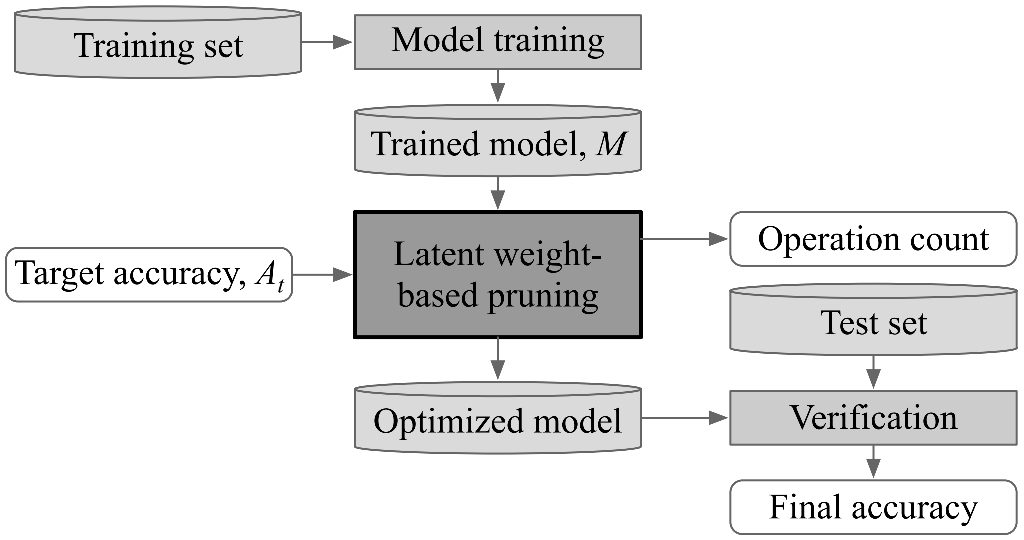


Figure 4.1: Proposed latent weight-based pruning method integrated in the model optimization flow.

describe the latent weight-based pruning highlighted in Figure 4.1 and its subroutines of the algorithm in detail.

Iterative Pruning Optimization

Algorithm 2 describes the main routine of the proposed latent-weight based pruning method, which is highlighted in Figure 4.1. The inputs to the pruning algorithm are M, A_t, and δ, where M is the trained BNN with unpruned weights, A_t is the target accuracy after pruning, and δ is the incremental increase in pruning percentage upon each iteration. Since BNNs are easily overfitted [21], A_t can be set to the accuracy of the original model before pruning, but it can also be any accuracy level that meets the application's requirement.

The pruning algorithm is performed by the iterative execution of getSensitivity to select a target layer through sensitivity analysis and

PRUNELAYER to actually prune the target layer. The algorithm is iterated over each unpruned layer of M until all layers have been pruned. We first find an unpruned layer that is most robust to pruning using GETSENSITIVITY, which is described in Section 4, and set it as the target layer l_p. The target layer l_p is gradually pruned until further pruning violates the accuracy requirement A_t. The pruning percentage is gradually incremented by δ each time. As mentioned in Section 4, the initial pruning percentage for the target layer, $t(l_p)$, is initialized to zero, and it is updated after every iteration of pruning of the layer.

The value of δ should be set small enough not to miss the fine-grained optimal point of the pruning percentage, but not too big in order to minimize computational overhead. We find $\delta = 10\%$ to be reasonable for most cases. The layer-wise pruning is repeated from the least sensitive layer to the most sensitive layer. We conclude the iterative procedure once all layers have been pruned or further pruning violates the target accuracy A_t.

Layer Sensitivity Analysis

When pruning convolutional layers, certain layers react with more volatility than others due to the low operation count after max-pooling or intrinsic small weight kernel size. Therefore, in order to get the most BOPs reduction without hurting accuracy, we determine the *sensitivity*, s, for every layer and prune the least sensitive layers first. For layer l, its sensitivity s_l is defined as the amount of accuracy loss, ΔA, over the operation count reduction, $\Delta BOPs$, as:

$$s_l = \frac{\Delta A}{\Delta BOPs}, \tag{4.3}$$

after pruning p percentage of the weights of layer l in isolation while other layers remain unpruned. The value p must be high enough to introduce accuracy instability within the network, generating a sufficient accuracy

Algorithm 2 Latent-weight based pruning

```
1: procedure PRUNE($M, A_t, \delta$)
2:     while exists an unpruned layer in $M$ do
3:         $s_{max} \leftarrow 0$
4:         $t(l_p) \leftarrow 0$
5:         for each unpruned layer $l$ do
6:             $s_l \leftarrow$ GETSENSITIVITY($l$)
7:             if $s_l > s_{max}$ then
8:                 $s_{max} \leftarrow s_l$; $l_p \leftarrow l$
9:             end if
10:        end for
11:        while $A_p > A_t$ do
12:            PRUNELAYER($l_p$, $t(l_p) + \delta$)
13:            $t(l_p) \leftarrow t(l_p) + \delta$
14:            $A_p \leftarrow$ Accuracy of pruned $M$
15:        end while
16:        Mark $l_p$ as pruned
17:    end while
18:    return $M$
19: end procedure
```

response. We find $p = 95\%$ to be reasonable in most cases to provoke a negative accuracy response within the network. This metric allows us to see which layers are less sensitive and likely to fluctuate less in accuracy when pruned. Effectively, this metric tells us how much accuracy loss we can expect a layer to contribute for a given amount of BOPs reduction. Therefore, the less sensitive a layer is, the better candidate for pruning it is.

The operation count for binarized layer l is calculated as the following:

$$\text{BOPs}_l = \left(\prod_{i=1}^{n} w_i \right) \times i_h \times i_w \tag{4.4}$$

where n is the dimension of the weights ($n = 4$ for convolutional layers

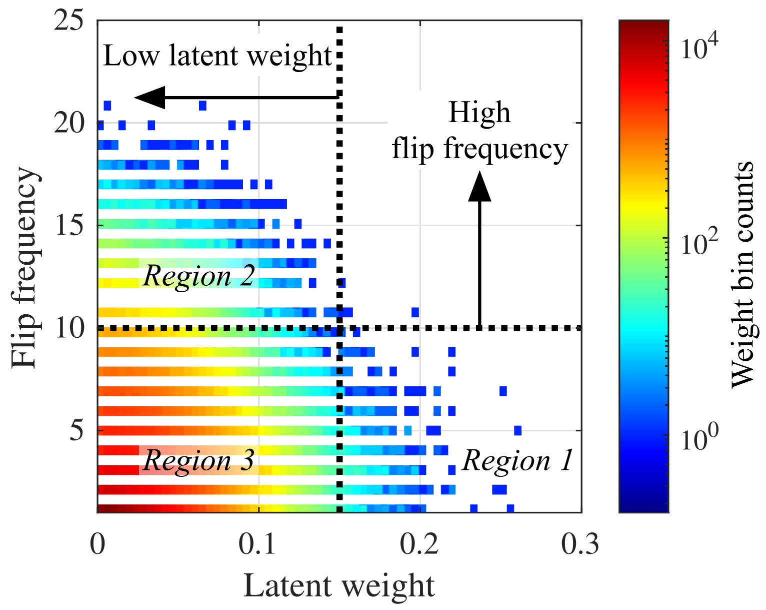


Figure 4.2: Comparison of the magnitude of latent weights and flips of layer 3 of BinaryNet's weight kernel.

and $n = 2$ for dense layers), w_i is the i-th index of the weight kernel, and i_h, i_w is the height and width of the output, respectively.

Pruning based on Latent Weights

Within the function PRUNELAYER(l_p, p), we prune the target layer l_p by removing the p percentile of the weight kernels with the lowest latent weight magnitude. Using latent weights offers distinct advantages over using the flip frequency. First, latent weights are a better indicator of the significance of weight kernels, which is often not correctly captured by flip frequencies. As described in Section 4, the larger the magnitude of the latent weights, the less likely the weight is unstable. Figure 4.2 shows the relationship between latent weights and flip frequencies captured from the BinaryNet as an example. It shows that the *maximum* latent weight is inversely proportional to flip frequency, but the near-zero latent weights, which are the majority of the weights (Region 3), show widely varying flip frequencies, between 0 and 21 in this example. In other words, a low flipping frequency does not always represent an important weight kernel with a high latent weight, and removing only high flip frequency weights (Region 2) may lead to ineffective pruning of weights. On the other hand, our method keeps the high latent weights (Region 1) for better pruning results, as we show in the experimental results.

Second, the real-valued nature of the latent weights allows us to perform more fine-grained pruning. We can distinguish almost every individual weight within the kernel and prune by the percentage of weights that fall below a certain threshold as opposed to pruning on discrete integer values. The ability to prune based on real-valued weights allows us to distinguish individual layers based on our sensitivity analysis as well. To illustrate the disadvantages of discretized pruning, Figure 4.3 shows that an overwhelming majority of flip frequencies have stabilized and are at $f = 0$, making them impossible to discern. Furthermore, flip frequencies that comprise the remainder of weights in the kernel comprise a small fraction of the overall weights. Therefore, there is no way to distinguish sensitivities and selectively choose layers to prune for a baseline flip frequency of $f > 2$. Simply pruning intermediate f values at $f = 1$ and $f = 2$ is unable to produce a sufficient accuracy response for sensitivity analysis. This is in contrast to real-valued pruning on BNNs using latent weights, where we can adjust the entire pruning threshold on a real-valued scale and effectively observe sensitivities.

As a result, the proposed latent weight-based pruning achieves higher reduction in BOPs while maintaining high accuracy, as demonstrated in the following section.

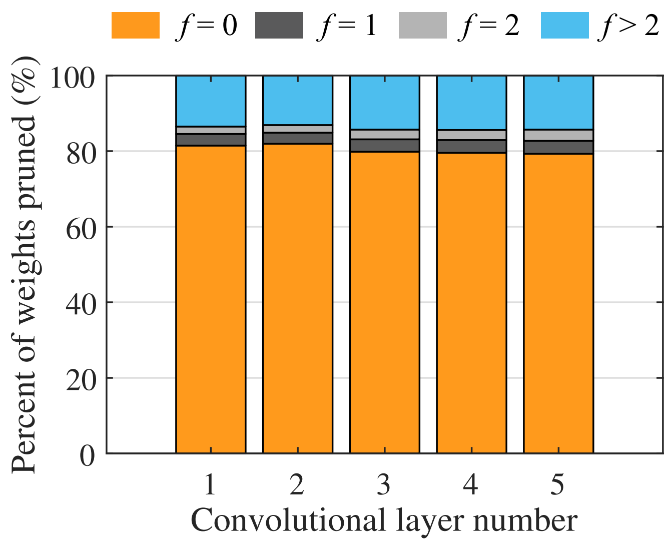


Figure 4.3: Discrete pruning for flip frequencies $f = 0$, $f = 1$, $f = 2$, and $f > 2$ of BinaryNet.

Table 4.1: Models and datasets used in the experiments.

Model	Dataset	Bin. conv. layers	Total BOPs	Base acc.
3ConvNet	MNIST [32]	2	4.5×10^6	97.4%
BinaryNet [25]	CIFAR-10 [29]	5	5.1×10^8	80.5%
XNOR-net [50]	Imagenette [24]	5	2.9×10^9	63.0%

Experiments

In this section, we demonstrate the efficacy of our BNN pruning algorithm in comparison to the baseline unpruned networks as well as network shrinking based on flip frequency [37].

We consider three neural networks, a simple three-layer CNN with two binarized convolutional layers (3ConvNet), BinaryNet [25], and XNOR-Net [50], trained for classification of the MNIST [32], CIFAR-10 [29], and Imagenette [24] datasets, respectively, to demonstrate the proposed

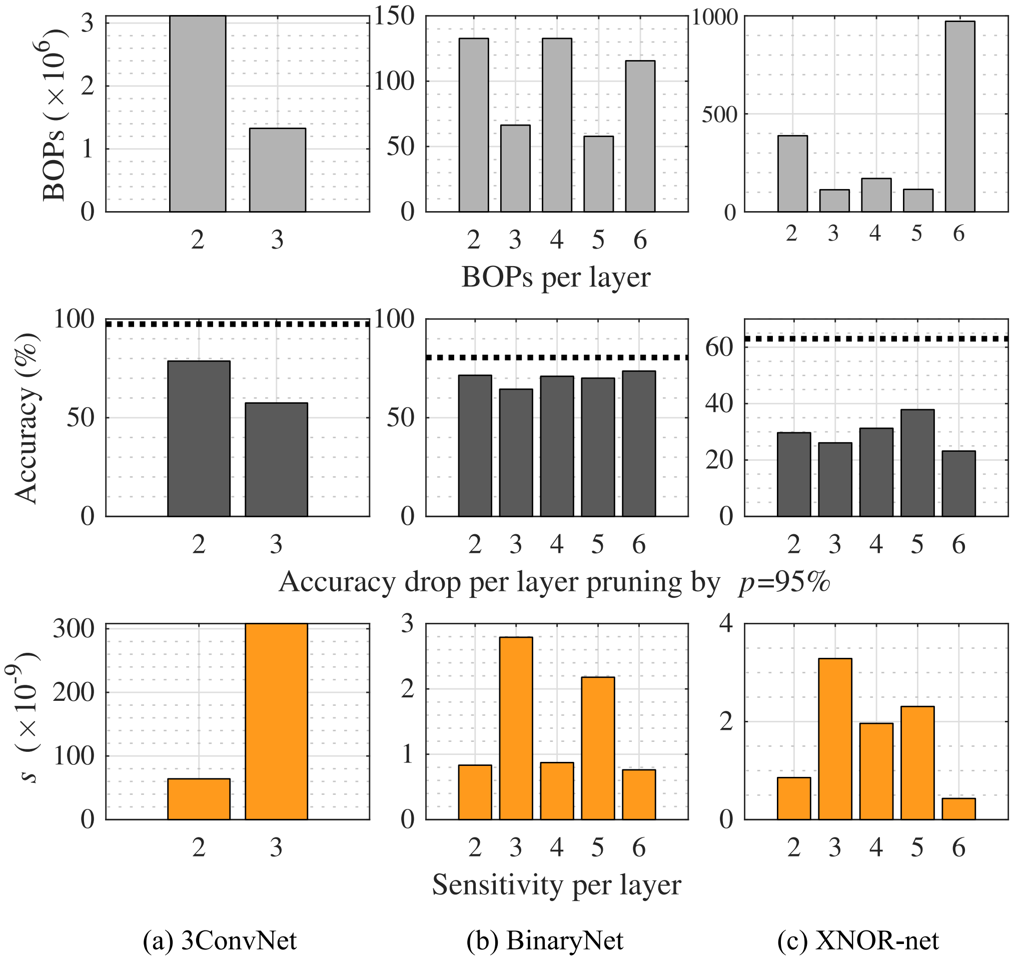


Figure 4.4: BOPs and sensitivities of the binarized convolutional layers of the three models. The dashed line indicates the baseline classification accuracy without any pruning.

pruning method on different sized image inputs and network complexities. Table 4.1 summarizes their complexities. The models are trained using Tensorflow and each BNN is built using the Larq API [1] and pruned using the proposed method. We use batch normalization after every layer and use 20% of the training set as a validation set used to determine the accuracy. Accuracy of the final result is evaluated once on the test set previously unseen to the algorithm. We do not retrain on the validation set and only determine the accuracy threshold strictly on the validation set. Fine tuning is done post-pruning with ten epochs with a learning rate at 10% of the original training learning rate. As discussed in Section 4, A_t is set to the baseline accuracy of the unpruned network to enable iso-accuracy comparison.

Results

We describe results on all three datasets with different networks.

We first implement a small 3-layer CNN with two binary convolutional layers to classify MNIST. The first layer is a floating point activation layer with 32 output channels followed by two binary convolutional layers with 32 and 64 output channels. The small network used is immediately responsive to pruning on both layers. The initial sensitivity analysis reports that the second convolutional layer is the least sensitive to pruning with $s = 64 \times 10^{-9}$ opposed to $s = 308 \times 10^{-9}$ as displayed in Figure 4.4(a). We first iterate through layer 2 with $\delta = 10\%$ and then proceed onto layer 2 with the same granularity. Results indicate a 42% reduction in BOPs and 30% reduction in model size compared to the unpruned base model with no accuracy loss as demonstrated in Figure 4.5(a). Latent weight pruning outperforms flip frequency pruning by 19.4% in BOPs reduction and by 12.5% in model size reduction.

The second model classifies CIFAR-10 using the BinaryNet architecture [25]. The initial sensitivity analysis reports that convolutional layer 4 is

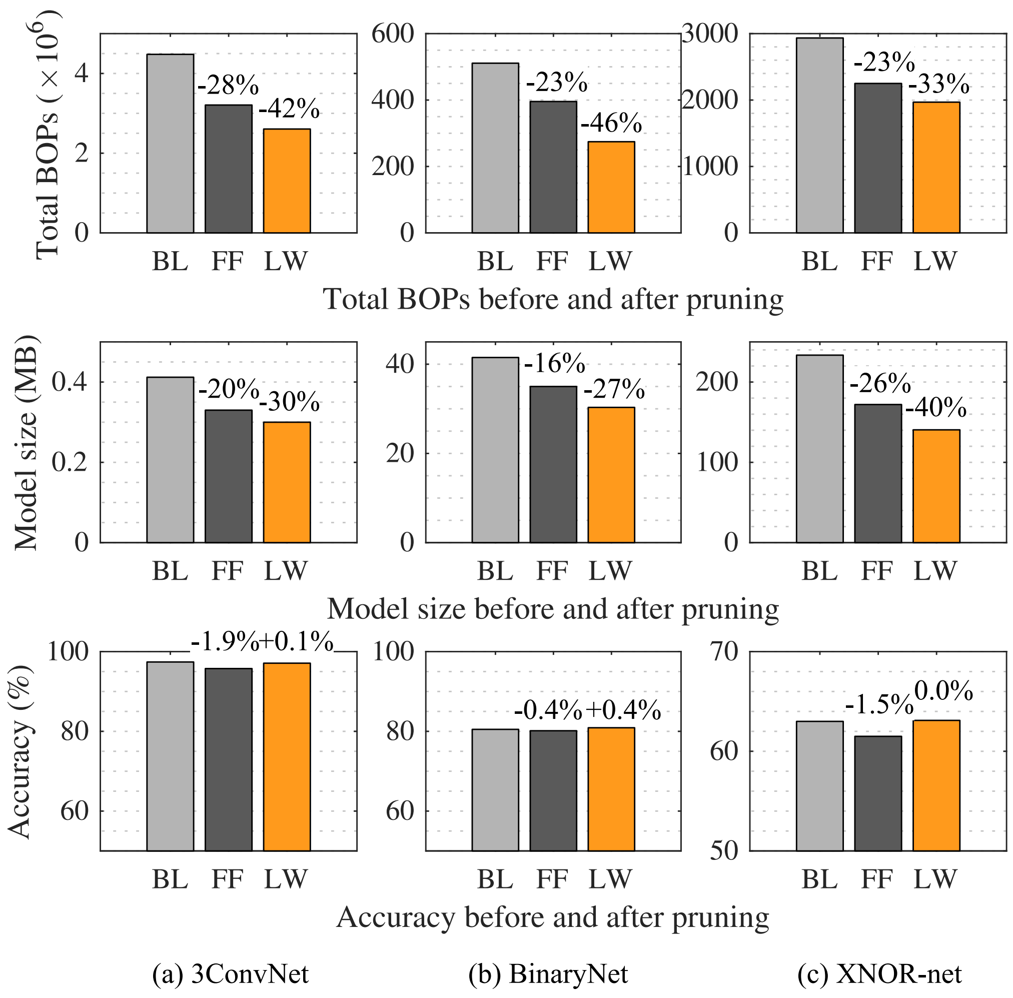
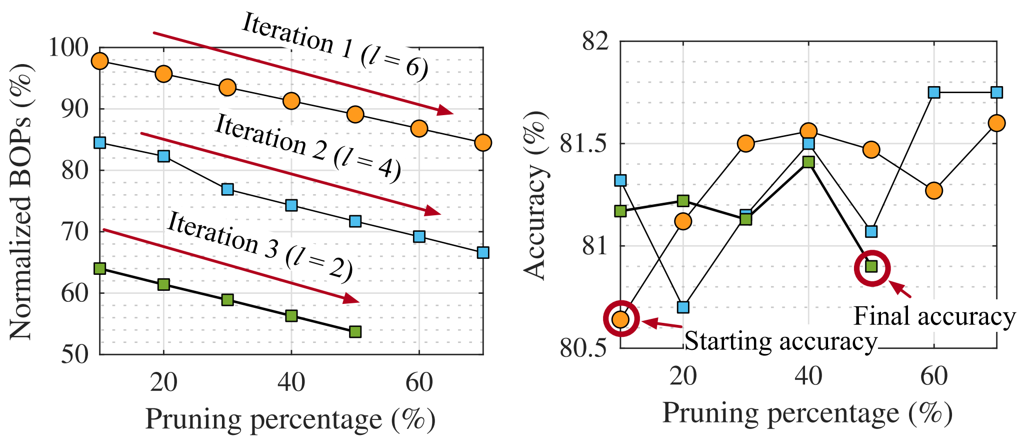


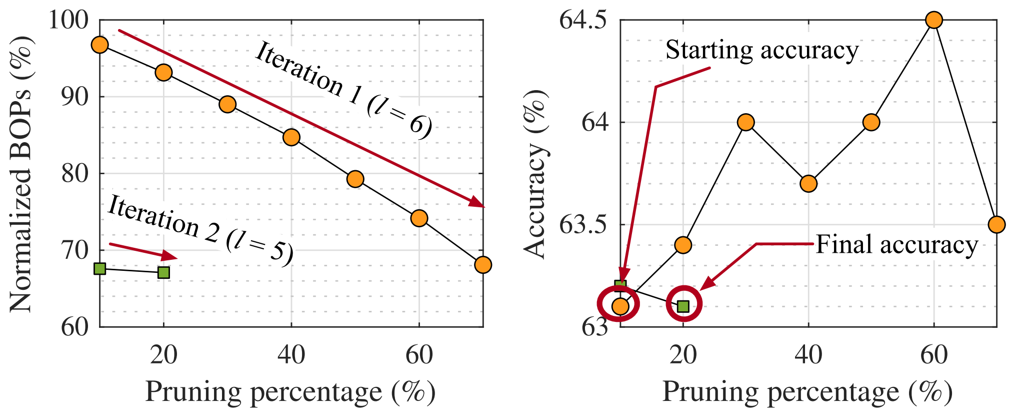
Figure 4.5: Comparison of total BOPs, model size, and accuracy before and after pruning. BL: baseline before pruning, FF: flip frequency-based pruning with channel shrinking [37], LW: latent weight-based pruning (proposed).

the least sensitive to pruning with $s = 0.71 \times 10^{-9}$ in Figure 4.4(b). The initial sensitivity analysis indicates binary convolutional layer 6 is the least sensitive to pruning, with the lowest magnitude s value. Since each layer contributes differently to the total number of operations within the network due to max pooling layers and increasing channel output width, imbalances occur as evidenced in Figure 4.4(b) on the right-hand side. This proves to be beneficial in the case of layer 6, as there is a very high BOPs count within this layer, allowing great BOPs reduction at the cost of little accuracy degradation. Therefore, we prune layer 6 to its entirety until the A_t is reached after fine-tuning. At each end of each pruning iteration, we recalculate s for every unpruned layer. In BinaryNet's case with CIFAR-10 dataset, we prune layers four and six as they are the least sensitive layers. In Figure 4.6(a), the first three iterations with multiple pruning steps are displayed, with layers 6, 4, and 2 only being pruned once before reaching the accuracy threshold. Beyond these first two iterations, we degrade accuracy beyond the A_t threshold. Results indicate a 46% reduction in BOPs and 27% reduction in model size compared to the unpruned base model with no accuracy loss as demonstrated in Figure 4.5(b). Latent weight pruning outperforms flip frequency pruning by 29.9% in BOPs reduction and by 13.1% in model size reduction.

We implement the XNOR-net architecture to classify the Imagenette dataset for our third network analysis [50, 24]. We initialize pruning on the fourth convolutional layer due to its sensitivity being the lowest at $s = 0.43 \times 10^{-9}$ as displayed in Figure 4.4(c). Layer 6 of XNOR-net contains a 6×6 convolutional filter, allowing for this layer to be significantly reduced by pruning over 80% of the original weights. Additionally, pruning layer 6 has massive implications on overall model size, since layer 6 comprises 62% of the total model storage size, allowing us to greatly reduce the model size more than the other designs. Following the pruning of layer 6, layer 5 is pruned according to it having the lowest sensitivity, which



(a) BinaryNet



(b) XNOR-net

Figure 4.6: BOPs reduction and classification accuracy during pruning of (a) BinaryNet and (b) XNOR-net.

is again re-evaluated at the end of each layer pruning iteration. Within Figure 4.4(c), the sensitivities only indicate what layer will be pruned on the first iteration, which is layer 6. The sensitivity is recalculated at the end of each iteration, meaning that while layer 2 has a lower sensitivity than layer 5 on the first iteration, layer 2 will not necessarily be pruned before layer 5. We increase pruning at $\delta = 10\%$ and prune layer-by-layer until the accuracy threshold is reached at 63%. In Figure 4.6(b), the first three iterations with multiple pruning steps are displayed, with layer 1 only being pruned once before reaching the accuracy threshold.

Results indicate a 33% reduction in BOPs and 40% reduction in model size compared to the unpruned base model with no accuracy loss as demonstrated in Figure 4.5(c). Latent weight pruning outperforms flip frequency pruning by 13.0% in BOPs reduction and by 18.9% in model size reduction.

Discussion

We compared our method to purely using a channel shrinking method based on weight flipping frequencies described in [37]. Our method provides a distinct advantage where channel shrinking is removing complexity from the network without exploiting the sparse resilience and lottery-ticket behavior of the network. In this method, we demonstrate the importance of exploiting sparsity within the network, as only a select few connections within the network are shown to be major contributors to the final classification accuracy [15].

Our results using iterative pruning incurs additional offline costs but produce a more efficient final pruned model. These offline costs can be handled by GPUs, producing a portable model for loading onto storage and computation-constrained systems. In particular, we have greatly reduced the amount of BOPs within the network. Our goal is to take computationally demanding training and not have it be handled by the edge device. The only edge device responsibility is inference with the ported model. While the accuracy gain is mild, this unstructured pruning regularizes the network efficiently. Channel shrinking, on the other hand, does not take advantage of the inertia of the binary weights and provides no guarantee for regularization. Thus, we conclude that with our method, we produce a more efficient and accuracy-robust pruned model than the previous work.

Conclusions

Latent weight-based BNN pruning is a promising approach which mixes two popular neural network compression techniques: quantization and weight pruning. The classical unstructured pruning that has been used in floating point models is difficult to integrate with BNNs due to their binary nature and the lack of weight magnitude in the forward pass. We demonstrated that latent weights that exist during backpropagation are a promising alternative that allows pseudogradient weights to represent how negligible a weight is. We presented a pruning solution that is precision-tuned to each layer, querying the sensitivities of individual components of the network to prune in a coordinated manner. In particular, our method focuses on reducing computational complexity and memory storage overhead of the pruned model. Compared to the previous work of pruning binary neural networks, we achieve a lower OPs count and smaller model size. On all three datasets with three different architecture sizes, we demonstrated 33%–46% reduction in operation count and a 27%–40% reduction in model size with no accuracy loss or up to a +0.4% gain.

5 EFFICIENT EDGE OFFLOADING

In the final thrust of the project, efficient edge offloading, we design a new machine learning scheme to reduce transmission latency between an edge client and server. We intelligently scale the resolution of transmitted data in order to reduce the amount of total data transmitted over a communication link. Our method uses a dynamic online reconfigurable scaling optimization that allows us to determine the best scaling for a transmitted image based on accuracy constraints.

Introduction

Extended Reality (XR) has been rapidly gaining traction across various domains, spanning from entertainment [13] to education [42] and healthcare [70]. Its utilization is extending into increasingly critical applications, as evidenced by its incorporation into mission-critical contexts [46, 3]. Given that XR devices are worn on the head, they must adhere to stringent size, weight, and power constraints, similar to other wearable technologies, and these constraints inevitably limit the computational capabilities of XR devices. Moreover, due to the interactive nature of XR applications, low latency is a critical requirement, which worsens the challenge. XR devices are typically equipped with a low-power system-on-chip (SoC) commonly found in smartphones, which fall short in meeting the demands of compute-intensive deep learning (DL) tasks. The gap between the computing power required and what can be provided is expected to widen as DL models continue to grow in complexity, facilitating more advanced applications.

Computation offloading stands out as a promising solution for enabling compute-intensive DL applications on resource-constrained devices [49]. It involves splitting the workload and leveraging resource-rich edge servers to handle the computationally intensive segments. Consequently, the qual-

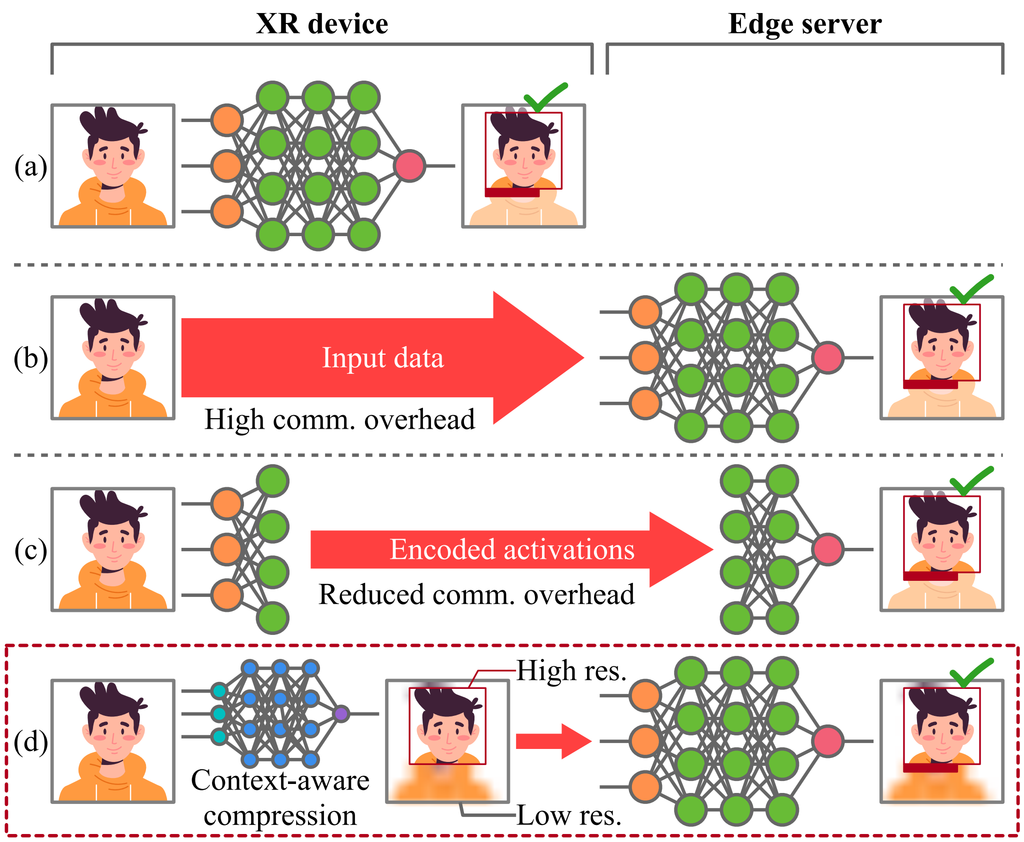


Figure 5.1: Computation offloading of face identification. (a) No offloading. (b) Full offloading of workload. (c) Partial offloading by partitioning workload. (d) Proposed context-aware offloading.

ity of service (QoS), such as latency and accuracy, can be significantly enhanced. However, it is essential to exercise caution during the workload partitioning process to ensure that it alleviates the burden on the XR device while minimizing communication overhead. Inefficient computation offloading can lead to limited or even negative QoS improvements, owing to excessive communication overhead and potential data quality deterioration.

In Figure 5.1, we illustrate the concept of computation offloading using

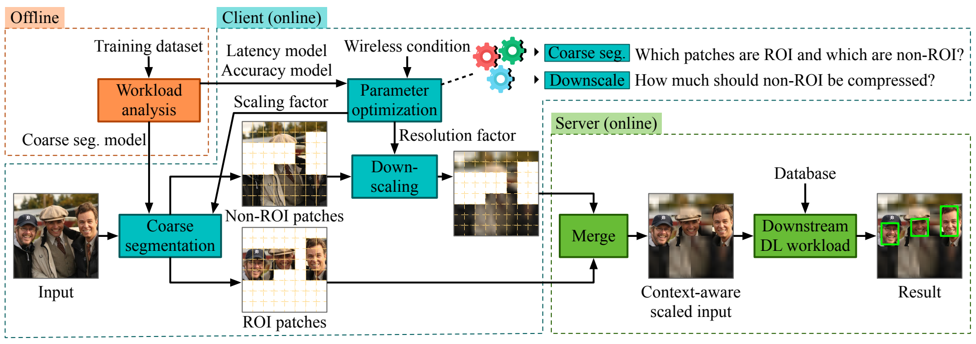


Figure 5.2: Computation offloading of face identification from an XR device (client) to a server. Image from IMDb-Face dataset [63].

DL-based face identification as an example. Figure 5.1(a), is the baseline scenario where the XR device handles the task entirely, which might not meet performance requirements. Figure 5.1(b) shows complete offloading, where the entire task is executed on the edge server. Although the edge server offers more computational power, transmitting the raw input data results in substantial communication overhead. In partial offloading shown in Figure 5.1(c), the DL model is partitioned into two segments so the transmitted intermediate data is minimized after the XR device executes part of the original model. Here, the DL model is divided into two segments, minimizing the transmission of intermediate data after the XR device completes the first segment. In this section, we propose *context-aware computation offloading* as illustrated in Figure 5.1(d), where the XR device intelligently compresses input data using a lightweight model in a context-aware manner, thereby significantly reducing the communication overhead.

More specifically, our framework introduces an intelligent input scaling mechanism on the XR device using a lightweight model. The compressed data is then transmitted to and processed by the full DL model hosted on the edge server. We refer to this approach as *coarse segmentation*, which efficiently identifies regions-of-interest (ROIs) within the input while reducing the fidelity of non-ROIs to minimize communication overhead, all while preserving the integrity of ROIs. The lightweight model executes a simplified version of the original model's task but remains distinct from it. For instance, in a face identification scenario, the lightweight model focuses on "face detection," a less complex task than "face identification." This process involves preserving the resolution of detected faces while downscaling the background, generating compressed input data. Subsequently, the edge server performs the more intricate identification task using this compressed, but ROI-preserved data.

The section's contributions can be summarized as follows:

- We introduce an efficient DL workload offloading approach centered on context-aware coarse segmentation, enabling ROI-preserving data scaling for XR devices. We introduce a novel training objective, coarse segmentation, to train a lightweight model for the efficient identification of ROIs.

- We propose an optimization framework capable of dynamically managing system operations to adhere to latency constraints in varying wireless conditions. This framework comprises offline characterization of DL workloads and online adaptation of data scaling.

- We implement a comprehensive end-to-end pipeline of our proposed framework in the context of face identification, showcasing improved accuracy when compared to baseline computation offloading.

Related Work

Computation offloading is a promising strategy for facilitating resource-intensive tasks on devices with limited resources. Achieving effective computation offloading hinges on the efficient reduction of communication overhead between the client and the server. In the context of video analysis, leveraging inter-frame similarities can substantially decrease this communication overhead. Since a video frame closely resembles a previously processed or transmitted frame, there is considerable potential to diminish both computation and communication requirements, as noted in [40, 36]. A recent work presents a technique to encode intermediate features to compress the data [68].

Accurate ROI detection plays a pivotal role in optimizing data scaling and encoding. It is essential that this detection process remains computationally efficient to ensure that the advantages of computation offloading are not undermined. One of the fundamental techniques involves straightforward frame-to-frame subtraction to identify significant changes between frames. Advanced ROI detection methods leverage various strategies, including exploiting historical frame data [40], lightweight local object detection technique [67, 26], and the integration of multi-camera networks to reduce overlap [18, 65].

Context-Aware Input Scaling

Our aim is to design a compute offloading method for XR systems for efficient DL workloads under latency constraints. We propose a new technique called *context-aware input scaling* to determine key ROI and scale the fidelity of the input data in order to lessen the communication costs. For the rest of this section, we focus on a face identification task as the application of the system, but the proposed method can be generally applicable to any compute-heavy DL workloads.

Let us consider a scenario where an XR device (the client) performs a face identification task that compares a face in a captured image to a pre-existing database of faces to find a matching identity. First, XR devices generally do not have the compute power to efficiently perform such a complex task. In addition, due to the memory and storage constraints, storing the entire face database in the client will be inefficient or infeasible due to the large storage requirement. Moreover, a face database contains sensitive personal data, which cannot be in distributed devices due to privacy concerns. For these reasons, the captured image cannot be processed on the client but must be sent to an edge server (the server) for processing.

Under a latency constraint, computation offloading must be performed with additional overheads taken into account, including data compression and transmission. In order to minimize the overhead the compression should have three crucial properties:

- Compressive: It should reduce the amount of data transmission substantially to reduce communication overhead.
- Efficient: The compression process itself should be lightweight to minimize compute overhead.
- Adaptive: The compression ratio must be should be able to adapt to varying wireless conditions to meet the latency constraint.

In the rest of this section, we describe how we design context-aware input scaling to meet these requirements.

Context-Aware Input Scaling Pipeline

The proposed context-aware input scaling pipeline is composed of four main components as illustrated in Figure 5.2.

- *Workload analysis* is an offline step that analyzes the latency and accuracy of the target DL workload. We also generate a coarse segmentation model in this step, which is used in the next component.
- *Coarse segmentation* is an online step performed by the client. The input is partitioned into ROI patches and non-ROI patches using the coarse segmentation model.

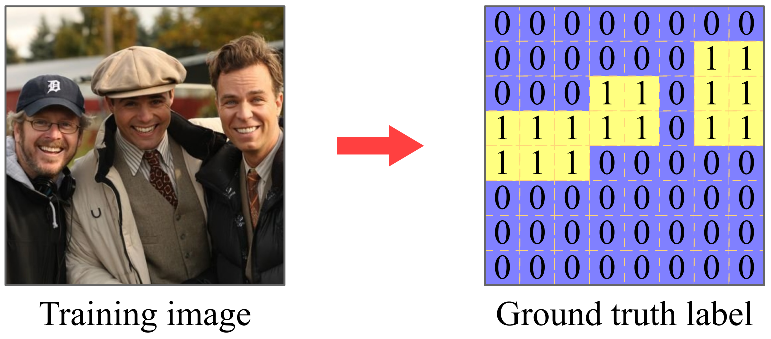


Figure 5.3: Training coarse segmentation model.

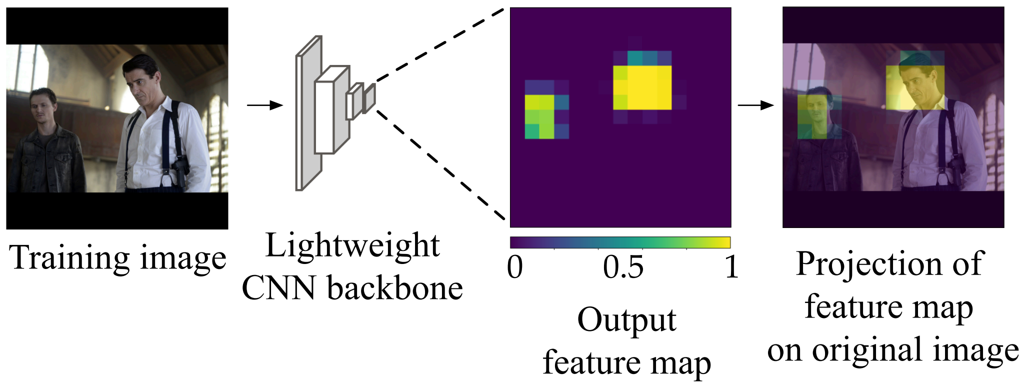


Figure 5.4: Design flow of client-side coarse segmentation model.

- *Parameter optimization* derives two control paramerters, scaling factor and resolution factor, used in the coarse segmentation and downscaling steps, respectively.
- *Downscaling* is where the non-ROI patches are downscaled based on the resolution factor determined in the parameter optimization step.

The ROI patches and the downscaled non-ROI patches are then transmitted to the server, where they are merged and processed by the downstream DL workload (face identification in this example). The following subsections describe each components of the pipeline in detail.

Coarse Segmentation

The coarse segmentation step is to partition an input image into ROI patches that contain ROIs and non-ROI patches that do not. In order to meet the goals discussed in Section **??**, we introduce a new task called *coarse segmentation* and propose a lightweight convolutional neural network (CNN)-based coarse segmentation method.

In coarse segmentation, the objective is to predict if a patch contains the pixels of ROI or not, where a patch is one piece of an image equally divided into $N \times N$. For the downstream task of face identification, we consider face pixels as our ROI. Each patch is labeled either '1' if the patch contains any face pixels or '0' otherwise. Thus, the ground truth label is a 2-D binary mask of dimensions $N \times N$ as shown in Figure 5.3.

In order to perform coarse segmentation, we redesign and adapt a lightweight CNN backbone image classifier to predict the ROI via its feature maps. Different layers in a CNN classifier learn a hierarchy of features, from simple and local features such as edges to more complex and global features such as object parts and, eventually, entire objects or patterns. The deeper layers capture increasingly abstract representations, making the network capable of recognizing and classifying complex patterns in the input data. Analysis of intermediate feature maps also shows that the regions that are activated exhibit a strong correlation with the regions containing semantic objects [63]. This indicates the possibility of approximating the positions of important regions within the image by analyzing the activations of feature maps. Therefore, we redesign the model and training objective for a lightweight CNN backbone of an image classifier such that it predicts ROI via its feature maps.

Figure 5.4 shows our approach to perform coarse segmentation. We train a lightweight CNN backbone without a classification head, such that the last feature map outputs high activations at the spatial locations of ROI and low activations otherwise. After applying the activation function, the output is interpreted as the probability of block containing ROI pixels. When this feature map is projected on the original image dimensions, we obtain the ROI and non-ROI patches as shown in Figure 5.4.

For training the model, the original RGB image serves as the input training image and the coarse segmentation mask serves as the ground truth. Loss is calculated by comparing the output feature map and the ground truth mask and the model weights are updated using backpropagation.

Next, we determine the number of patches that will be downscaled in resolution using a parameter called *scaling factor* $0 \leqslant S_f \leqslant 1$. It is defined as

$$S_f = 1 - \frac{N_{ds}}{N^2}, \tag{5.1}$$

where N_{ds} is number of downscaled patches. These patches are selected based on the output feature map of coarse segmentation model. The patches are sorted based on their probability scores from the feature map. A low probability score indicates that the patch is unimportant for the downstream task and hence can be downscaled in resolution. Therefore, we downscale N_{ds} patches with the lowest probability scores. Figure 5.5 shows coarse segmentation of an image with three different scaling factors. Within Figure 5.5, we see that as scaling factor increases, the amount of high resolution original input image increases. Our system tunes a balance between high accuracy, high scaling factor and low latency, low scaling factor inputs. This parameter will be explicitly defined within our system design and dynamically adjusted depending upon application constraints.

Workload Analysis

The workload analysis is an offline step to understand the computation and communication requirements of the workload in the context of the DL task. This step builds a latency model and an accuracy model for a given training dataset. It also produces a coarse segmentation model, which has been described above within coarse segmentation.

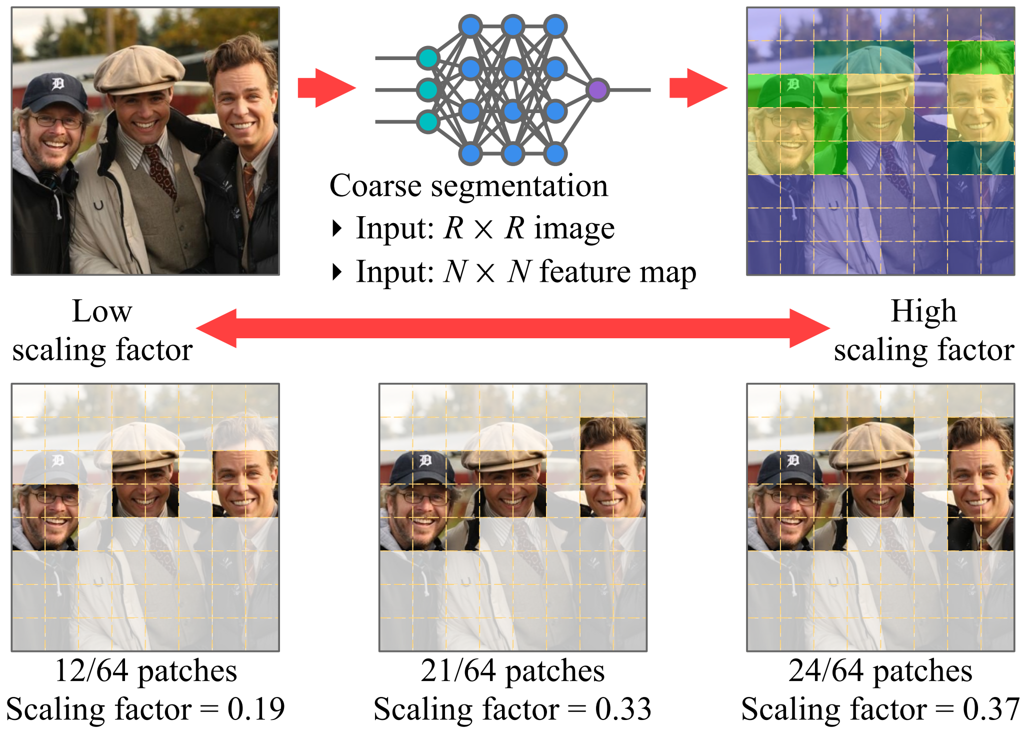


Figure 5.5: Example of coarse segmentation with three different scaling factors. $N = 8$.

First, a latency model is built based on empirical measurement and analytical estimation. The total latency on the client side to process an image is the sum of the acquisition latency L_{acq} to obtain the source image, the processing latency L_{proc} to split the image into ROI and non-ROI patches, and the transmission latency L_{tx}. We do not consider the processing latency on the server side, since it is generally negligible due to its high performance. The total latency should be less than or equal to the latency constraint L_c.

$$L_{acq} + L_{proc} + L_{tx} \leqslant L_c \tag{5.2}$$

Since the image resolution R is fixed, L_{acq} is constant. The majority of L_{proc} is for coarse segmentation with some for image partitioning, and this is also constant for a fixed R on a given client device. Therefore, theses two latency factors can be empirically measured on the target client platform.

On the other hand, L_{tx} is most the dominant and variable latency factor. First, we downscale non-ROI patches by a resolution factor R_f, which is the percentage of how much we downscale the individual patch resolution height and width by. Without downscaling, the baseline transmission latency L_{tx}^{base} would be directly proportional to the size of the image and inversely proportional to the datarate.

$$L_{tx}^{base} = c\frac{R^2}{\gamma}, \tag{5.3}$$

where c is a constant that relates the image size to data size, and γ is the data rate. Then, with the proposed scaling, for a given S_f and R_f, L_{tx} is reduced to

$$L_{tx} = c\frac{R^2}{\gamma}\left(S_f + {R_f}^2(1 - S_f)\right). \tag{5.4}$$

Since $0 \leqslant R_f \leqslant 1$, L_{tx} increases as either S_f or R_f increases.

This step also generates an accuracy model. It characterizes the accuracy of the downstream task by running it with different S_f or R_f settings. As the quality of the image also increases as S_f or R_f increases, the accuracy also increases. Considering these latency and accuracy characteristics, the next step is to find the optimal values of S_f or R_f that maximizes the accuracy while meeting the constraint (5.2) as discussed in the next subsection.

Parameter optimization

In order to determine optimum S_f and R_f, we use a combination of latency analysis combined with downstream task accuracy $A_s(S_f, R_f)$ calculated offline. While the parameters are selected online, the latency and accuracies

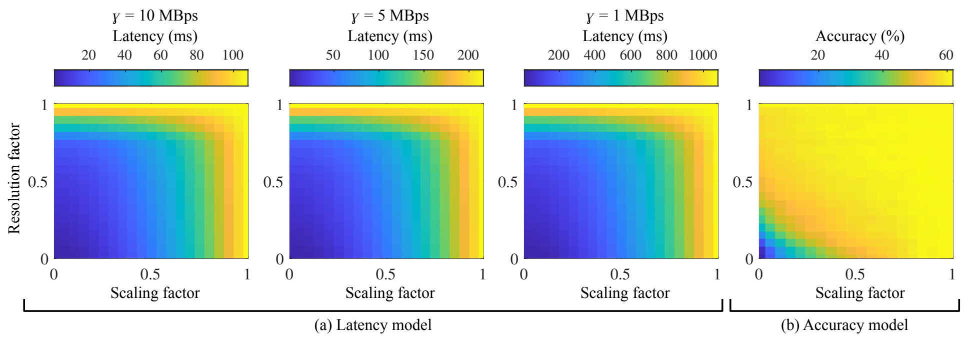


Figure 5.6: (a) Latency model $L_{tx}(S_f, R_f)$ for three data rates, 10, 5, and 1 MBps. (b) Accuracy model $A_{tx}(S_f, R_f)$.

are already calculated, ready to conform to constraints. We dynamically choose parameters (S_f, R_f) such that we scale the input image to match application-defined latency constraint L_c while maintaining the highest level of accuracy based on validation data accuracy for the downstream task.

To do this, we use C_a to represent all possible (S_f, R_f) configurations corresponding to unique latencies, L:

$$\text{Let } C_s \text{ be the set of elements: } C_s = \{A_i : L_{tx}(S_f, R_f) < L_c\}, \tag{5.5}$$

where C_s is the subset of configurations satisfying the latency constraint within C_a, and A_i is the accuracy of the particular configuration. Thus, the optimization problem is to find the configuration that maximizes accuracy within C_s:

$$(S_{fo}, R_{fo}) = \max_{A_i \in C_s} (A_o) \tag{5.6}$$

where (S_{fo}, R_{fo}) is the optimum scaling and resolution factors that satisfy constraints, and A_o is the highest accuracy within the subset of configurations C_s. With this, we can say that (S_{fo}, R_{fo}) is the most latency-constrained, highest-accuracy configuration for context-aware resolution

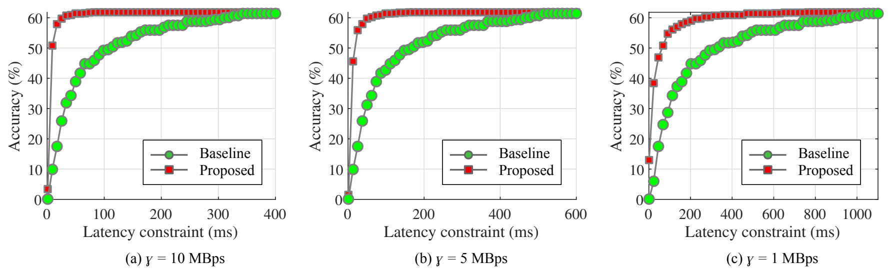


Figure 5.7: Accuracy of face identification after computation offloading for different latency constraints and data rates.

scaling for a given data rate.

Experiments

In this section, we demonstrate the efficacy of our context-aware efficient edge offloading scheme for object detection.

Within our system model, we use the Raspberry Pi 4 as our XR client running a redesigned MobileNetV3-Small backbone as our coarse segmentation model. We use the Raspberry Pi for flexibility in software modification, but the proposed methodology is largely hardware-agnostic.

Face identification involves detecting faces in an input image and predicting the individuals by matching them against a gallery of labeled images of individuals. While we use face identification for this experiment, coupled similar lightweight models and downstream tasks can also be used in our agnostic design. We use a subset of 100 identities from IMDb-Face dataset [63] which comprises of annotated faces of celebrities as our test dataset for the downstream face identification model. We construct an annotated gallery of images of at most ten images per individual against which the test images are matched for identification. The test set comprises of 1190 images that do not overlap with the gallery. For the coarse segmentation model, we use 5000 training images and 2000 test images from IMDb-Face dataset. We ensure that the training data for coarse segmentation does not overlap with the test data for the face identification downstream task. The images in this dataset are official photos, lifestyle photos, and movie snapshots of celebrities sourced from the IMDb website. The images display large variations in terms of scale, pose, lighting, and occlusion of faces, as well as number of faces per image. Specifically, movie snapshots provide a diverse dataset for testing the robustness of our methodology.

As a baseline design, we consider an image downscaling without context awareness. It downscales the entire input image to meet the latency constraint without partitioning into ROI and non-ROI.

Client-side Coarse Segmentation Model

We redesign MobileNetV3-Small [23] classification model to adapt to the task of coarse segmentation. MobileNetV3 architecture design is well-suited for deployment on resource-constrained environments due to its efficiency and competitive performance as compared to other CNN backbones [23]. Post-training dynamic range quantization is used to convert the model from tensorflow to TFLite.

As described above, we remove the classification head and the final three bottleneck layers. The ground truth coarse segmentation mask was created using the bounding box annotations provided in the dataset. Here, we treat the region inside face bounding box as the region of interest. Thus, the regions within a bounding box for a face were labeled as "1" and "0" otherwise. We add a point-wise convolution layer to this trimmed model to produce a feature map with a channel dimension of 1. The input resolution is 600 $\times$ 600 pixels ($R = 600$), and the output feature map resolution is 15 $\times$ 15 ($R = 15$). The output feature map is selected from one of the internal layers of the coarse segmentation model. Multiple layers can potentially be chosen, resulting in different coarse segmentation feature map sizes.

We use binary cross-entropy loss to train the model for 500 epochs with Adam optimizer and learning rate of 0.001. On evaluating on the test data, we obtain a PR-AUC (area under the precision-recall curve) score of 0.82. We use the value of cross-validation to select the best weights with minimum validation loss. The model is trained in TensorFlow framework and converted to TFLite for portability on Raspberry Pi. The execution time of the coarse segmentation model and image partitioning (L_{proc}) is 46 milliseconds per image. This short execution time is compared to full face identification on Raspberry Pi, taking well over 2.8 seconds for a small subset of 3 faces to identify. We can conclude that coarse segmentation is much faster than full face identification on the Pi without any of the privacy concerns that accompany a face database on the client.

Server-side Latency Optimization

In our design, since typical Wi-Fi connections greatly depend on varying transmission upload rates, we analytically find latency according to the methods described in the previous section. We can visualize the latency surface when plotting the surface of (5.4) in Figure 5.6(a) with an image resolution of $r = 600$, and an estimated client upload data rate of 10 Mbps, 5 MBps, and 1 MBps. There are 420 total different configurations within Figure 5.7 for each data rate. We choose 20 different S_f equidistant from $S_f = 0$ to $S_f = 1$. We choose 21 different R_f equidistant from $R_f = 0$ to $R_f = 1$. Therefore, we can conclude that transmission latency is the dominant source of overall system latency since transmission latency, during low data rate transmission (1 MBps), is over 25$\times$ model and preprocessing latency. Figure 5.6(a) provides analytical latency calculations given different data rates, such that we observe S_f and R_f increase proportionally to the latency of transmission of individual frames. Conversely, Figure 5.6(b) displays the accuracy of our face identification model given different S_f and R_f configurations. We use off-the-shelf ArcFace [14] model for face identification, since our methodology remains model-agnostic such that accuracy values remain "as is" for off-the-shelf models. Latency disproportionately decreases by a large percentage for a small decrease in accuracy when adjusting scaling and resolution factor. Therefore, our system is accuracy-insensitive but remains latency-sensitive when doing context-aware resolution scaling. Similarly, we notice that accuracy increases as S_f and R_f increases, as the image lies closer to its full resolution.

Parameter Optimized for Accuracy

Given an application-specified latency constraint, our design selects the best S_f and R_f such that our downstream task accuracy is the highest. We use the method described within our workload analysis for multiple latency constraints and plot them in Figure 5.7. In Figure 5.7, we outperform the baseline resolution scaling for multiple data rates giving us 16.52%–17.34% higher accuracy on average across three different data rates than simple baseline resolution scaling across sample latency constraints. The unique corner case of $R_f = 0$ where only ROI are sent is displayed in Figure 5.7. Accuracy drastically declines when R_f decreases to near-zero (only ROI) values. Therefore, we optimally wish to select R_f that is both nonzero, indicating that resolution scaling still preserves original task accuracy. Note that as data rate decreases, the accuracy sensitivity decreases, with dramatic changes in accuracy for small latency constraint adjustments at higher data rates. For lower data rate scenarios, our method outperforms the baseline resolution scaling at greater latency percentages, due to dominance of transmission latency over the entire workload. Therefore, we can conclude that context-aware resolution scaling is more critical in low-bandwidth scenarios.

Conclusions

Computation offloading, if properly designed, can greatly improve the performance and latency of DL applications on XR systems. We introduced a method to improve computation offloading from an XR device to an edge server. Our method optimizes scaling of wireless transmitted images by identifying ROI to preserve its fidelity while reducing the fidelity of non-ROI background. We used face identification as the downstream DL task as an example and demonstrated a significant improvement of accuracy under the same latency constraint. We achieved 16.52%–17.34% higher downstream task accuracy for three different data rates under the same latency constraints as the baseline naive resolution scaling.

For future work, our method can be extrapolated to a variety of different downstream tasks, such as pose segmentation. Additionally, multiple other control knobs, such as variable resolution adjustment for higher R_f around ROI or variable data rate fluctuatio, can be examined for accuracy preservation.

6 CONCLUSION

Edge machine learning is growing, with more applications being ported on energy-constrained devices. In order to satisfy energy constraints, some of form of low-power computing must be employed to enable edge machine learning. Approximate computing is a design paradigm new to machine learning that leverages both software and hardware elements by relaxing the accuracy requirement so long as the output meets a certain quality threshold. The highly error-tolerant probabilistic nature of machine learning lends itself to approximate computing being used to enable machine learning.

In this thesis, we first introduce a hardware-based memory-aware logic synthesization. This proposed scheme has been successfully implemented and optimized with CLNNs and shift-and-add multiplication schemes. Next, we introduce a software-based energy-aware model compression with latent weight based pruning. Lastly, we introduce an efficient edge offloading technique that allows for low-latency communication between a client and server for couple machine learning tasks. We hope to combine elements of hardware and software based machine learning by strategically by adding approximate multipliers and profiling multiplications using machine learning.

BIBLIOGRAPHY

[1] Bannink, T., Hillier, A., Geiger, L., de Bruin, T., Overweel, L., Neeven, J., and Helwegen, K. Larq compute engine: Design, benchmark and deploy state-of-the-art binarized neural networks. *Proceedings of Machine Learning and Systems (MLSys) 3* (2021), 680–695.

[2] Bengio, Y., Léonard, N., and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432* (2013).

[3] Bhattarai, M., Jensen-Curtis, A. R., and Martínez-Ramón, M. An embedded deep learning system for augmented reality in firefighting applications. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2020), IEEE, pp. 1224–1230.

[4] Castellano, G., Fanelli, A. M., and Pelillo, M. An iterative pruning algorithm for feedforward neural networks. *IEEE transactions on Neural networks 8*, 3 (1997), 519–531.

[5] Chandrasekar, K., Akesson, B., and Goossens, K. Improved power modeling of ddr sdrams. In *2011 14th Euromicro Conference on Digital System Design* (2011), IEEE, pp. 99–108.

[6] Chen, T., Anderson, N., and Kim, Y. Latent weight-based pruning for small binary neural networks. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference (ASP-DAC)* (2023), pp. 751–756.

[7] Chen, T., Kemp, T., and Kim, Y. Synthnet: A high-throughput yet energy-efficient combinational logic neural network. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)* (2022), IEEE, pp. 232–237.

[8] Chen, X., Zhu, J., Jiang, J., and Tsui, C.-Y. Tight compression: compressing cnn model tightly through unstructured pruning and simulated annealing based permutation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)* (2020), IEEE, pp. 1–6.

[9] Chen, Y.-H., Krishna, T., Emer, J. S., and Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits 52*, 1 (2016), 127–138.

[10] Cheng, M., Xia, L., Zhu, Z., Cai, Y., Xie, Y., Wang, Y., and Yang, H. Time: A training-in-memory architecture for memristor-based deep neural networks. In *Proceedings of the 54th Annual Design Automation Conference (DAC)* (2017), pp. 1–6.

[11] Chi, C.-C., and Jiang, J.-H. R. Logic synthesis of binarized neural networks for efficient circuit implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (ICCAD) 41*, 4 (2021), 993–1005.

[12] Chi, P., Li, S., Xu, C., Zhang, T., Zhao, J., Liu, Y., Wang, Y., and Xie, Y. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. vol. 44, ACM New York, NY, USA, pp. 27–39.

[13] Dargan, S., Bansal, S., Kumar, M., Mittal, A., and Kumar, K. Augmented reality: A comprehensive review. *Archives of Computational Methods in Engineering 30*, 2 (2023), 1057–1080.

[14] Deng, J., Guo, J., Xue, N., and Zafeiriou, S. Arcface: Additive angular margin loss for deep face recognition. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2019), pp. 4690–4699.

[15] Frankle, J., and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635* (2018).

[16] Gao, J., Liu, Q., and Lai, J. An approach of binary neural network energy-efficient implementation. *Electronics 10*, 15 (2021), 1830.

[17] Goksoy, A. A., Li, G., Mandal, S. K., Ogras, U. Y., and Marculescu, R. Cannon: Communication-aware sparse neural network optimization. *IEEE Transactions on Emerging Topics in Computing* (2023).

[18] Guo, H., Yao, S., Yang, Z., Zhou, Q., and Nahrstedt, K. Crossroi: Cross-camera region of interest optimization for efficient real time video analytics at scale. In *Proceedings of the 12th ACM Multimedia Systems Conference* (2021), pp. 186–199.

[19] Han, J., and Orshansky, M. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)* (2013), IEEE, pp. 1–6.

[20] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[21] Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems (NIPS) 28* (2015).

[22] Helwegen, K., Widdicombe, J., Geiger, L., Liu, Z., Cheng, K.-T., and Nusselder, R. Latent weights do not exist: Rethinking binarized neural network optimization. *Advances in Neural Information Processing Systems (NIPS) 32* (2019).

[23] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision* (2019), pp. 1314–1324.

[24] Howard, J. Imagewang, 2019.

[25] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. *Advances in Neural Information Processing Systems (NIPS) 29* (2016).

[26] Jiang, S., Lin, Z., Li, Y., Shu, Y., and Liu, Y. Flexible high-resolution object detection on edge devices with tunable latency. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking* (2021), pp. 559–572.

[27] Kim, Y., San Miguel, J., Behroozi, S., Chen, T., Lee, K., Lee, Y., Li, J., and Wu, D. Approximate hardware techniques for energy-quality scaling across the system. In *2020 International Conference on Electronics, Information, and Communication (ICEIC)* (2020), IEEE, pp. 1–5.

[28] Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342* (2018).

[29] Krizhevsky, A., Nair, V., and Hinton, G. The cifar-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html 55*, 5 (2014).

[30] LeCun, Y., Bengio, Y., and Hinton, G. Deep learning. *nature 521*, 7553 (2015), 436–444.

[31] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[32] LeCun, Y., and Cortes, C. MNIST handwritten digit database.

[33] Lee, E. H., Miyashita, D., Chai, E., Murmann, B., and Wong, S. S. Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2017), IEEE, pp. 5900–5904.

[34] Li, F., Zhang, B., and Liu, B. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[35] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).

[36] Li, Y., Padmanabhan, A., Zhao, P., Wang, Y., Xu, G. H., and Netravali, R. Reducto: On-camera filtering for resource-efficient real-time video analytics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM)* (2020), pp. 359–376.

[37] Li, Y., and Ren, F. Bnn pruning: Pruning binary neural network guided by weight flipping frequency. In *2020 21st International Symposium on Quality Electronic Design (ISQED)* (2020), IEEE, pp. 306–311.

[38] Li, Y., Zhang, S., Zhou, X., and Ren, F. Build a compact binary neural network through bit-level sensitivity and data pruning. *Neurocomputing 398* (2020), 45–54.

[39] Lin, J., Chen, W.-M., Lin, Y., Gan, C., Han, S., et al. MCUNet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems (NIPS) 33* (2020), 11711–11722.

[40] Liu, L., Li, H., and Gruteser, M. Edge assisted real-time object detection for mobile augmented reality. In *The 25th annual international conference on mobile computing and networking (MOBICOM)* (2019), pp. 1–16.

[41] Liu, Z., Yazdanbakhsh, A., Park, T., Esmaeilzadeh, H., and Kim, N. S. Simul: An algorithm-driven approximate multiplier design for machine learning. *IEEE Micro 38*, 4 (2018), 50–59.

[42] López-Belmonte, J., Moreno-Guerrero, A.-J., López-Núñez, J.-A., and Hinojo-Lucena, F.-J. Augmented reality in education. a scientific mapping in web of science. *Interactive learning environments 31*, 4 (2023), 1860–1874.

[43] Mrazek, V., Vasícek, Z., Sekanina, L., Hanif, M. A., and Shafique, M. Alwann: Automatic layer-wise approximation of deep neural network accelerators without retraining. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2019), IEEE, pp. 1–8.

[44] Nazemi, M., Pasandi, G., and Pedram, M. Energy-efficient, low-latency realization of neural networks through boolean logic minimization. In *Proceedings of the 24th Asia and South Pacific design automation conference (ASPDAC)* (2019), pp. 274–279.

[45] Panda, P., Sengupta, A., and Roy, K. Conditional deep learning for energy-efficient and enhanced pattern recognition. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2016), IEEE, pp. 475–480.

[46] Peretti, O., Spyridis, Y., Sesis, A., Efstathopoulos, G., Lytos, A., Lagkas, T., and Sarigiannidis, P. Augmented reality training, command and control framework for first responders. In *2022 7th South-East Europe Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)* (2022), IEEE, pp. 1–5.

[47] Raha, A., and Raghunathan, V. qLUT: Input-aware quantized table lookup for energy-efficient approximate accelerators. *ACM Transactions on Embedded Computing Systems* (2017).

[48] Rai, S., Neto, W. L., Miyasaka, Y., Zhang, X., Yu, M., Yi, Q., Fujita, M., Manske, G. B., Pontes, M. F., da Rosa, L. S., et al. Logic synthesis meets machine learning: Trading exactness for generalization. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2021), IEEE, pp. 1026–1031.

[49] Ran, X., Chen, H., Liu, Z., and Chen, J. Delivering deep learning to mobile devices via offloading. In *Proceedings of the Workshop on Virtual Reality and Augmented Reality Network (SIGCOMM)* (2017), pp. 42–47.

[50] Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. XNOR-Net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision (ECCV)* (2016), pp. 525–542.

[51] Razlighi, M. S., Imani, M., Koushanfar, F., and Rosing, T. LookNN: Neural network with no multiplication. *Design Automation and Test in Europe (DATE)* (2017).

[52] Rusci, M., Cavigelli, L., and Benini, L. Design automation for binarized neural networks: A quantum leap opportunity? In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)* (2018), IEEE, pp. 1–5.

[53] Samadi, M., Lee, J., Jamshidi, D. A., Hormati, A., and Mahlke, S. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (2013), pp. 13–24.

[54] Samajdar, A., Zhu, Y., Whatmough, P., Mattina, M., and Krishna, T. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883* (2018).

[55] Sekanina, L., Vasicek, Z., and Mrazek, V. Approximate circuits in low-power image and video processing: The approximate median filter. *Radioengineering 26*, 3 (2017).

[56] Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), pp. 124–134.

[57] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2015.

[58] Tann, H., Hashemi, S., Bahar, I., and Reda, S. Hardware-software codesign of accurate, multiplier-free deep neural networks. *Design Automation Conference (DAC)* (2017).

[59] Umuroglu, Y., Akhauri, Y., Fraser, N. J., and Blott, M. Logicnets: Co-designed neural networks and circuits for extreme-throughput applications. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)* (2020), IEEE, pp. 291–297.

[60] Venkataramani, S., Chakradhar, S. T., Roy, K., and Raghunathan, A. Approximate computing and the quest for computing efficiency. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)* (2015), IEEE, pp. 1–6.

[61] Venkataramani, S., Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Quality programmable vector processors for approximate computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2013), pp. 1–12.

[62] Wang, E., Davis, J. J., Cheung, P. Y., and Constantinides, G. A. Lutnet: Rethinking inference in fpga soft logic. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2019), IEEE, pp. 26–34.

[63] Wang, L., Ouyang, W., Wang, X., and Lu, H. Visual tracking with fully convolutional networks. In *Proceedings of the IEEE international conference on computer vision (ICCV)* (2015), pp. 3119–3127.

[64] Wang, L., and Wang, X. Approximate communication for energy-efficient network-on-chip. In *Advances in Computers*, vol. 124. Elsevier, 2022, pp. 151–215.

[65] Wang, Z., He, X., Zhang, Z., Zhang, Y., Cao, Z., Cheng, W., Wang, W., and Cui, Y. Edge-assisted real-time video analytics with spatial–temporal redundancy suppression. *IEEE Internet of Things Journal 10*, 7 (2022), 6324–6335.

[66] Wu, D., Chen, T., Chen, C., Ahia, O., San Miguel, J., Lipasti, M., and Kim, Y. Seco: A scalable accuracy approximate exponential function via cross-layer optimization. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)* (2019), IEEE, pp. 1–6.

[67] Yang, Z., Wang, X., Wu, J., Zhao, Y., Ma, Q., Miao, X., Zhang, L., and Zhou, Z. Edgeduet: Tiling small object detection for edge assisted autonomous mobile vision. *IEEE/ACM Transactions on Networking* (2022).

[68] Yao, S., Li, J., Liu, D., Wang, T., Liu, S., Shao, H., and Abdelzaher, T. Deep compressive offloading: Speeding up neural network inference by trading edge computation for network latency. In *Proceedings of the 18th conference on embedded networked sensor systems (SenSys* (2020), pp. 476–488.

[69] Zhang, B., Davoodi, A., and Hu, Y. H. Efficient inference of cnns via channel pruning. *arXiv preprint arXiv:1908.03266* (2019).

[70] Zhang, J., Lu, V., and Khanduja, V. The impact of extended reality on surgery: a scoping review. *International Orthopaedics 47*, 3 (2023), 611–621.

[71] Zhang, Q., Wang, T., Tian, Y., Yuan, F., and Xu, Q. Approxann: An approximate computing framework for artificial neural network. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2015), IEEE, pp. 701–706.