©Copyright 2020
Joshua Wolff Fromm

Implementing Binary Neural Networks.

Joshua Wolff Fromm

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

Doctor of Philosophy

University of Washington

2020

Reading Committee:

Shwetak Patel, Chair

Matthai Philipose

Joshua Smith

Program Authorized to Offer Degree:
Department of Electrical and Computer Engineering

University of Washington

Abstract

Implementing Binary Neural Networks.

Joshua Wolff Fromm

Chair of the Supervisory Committee:
Professor Shwetak Patel
Electrical and Computer Engineering

The recent renaissance of deep neural networks has lead to impressive advancements in many domains of machine learning. However, the computational cost of these neural models increases in line with their performance, with many state-of-the-art models only being able to run on expensive high-end hardware. The need to efficiently deploy neural networks to commodity platforms has made network optimization a popular field of research. One particularly promising technique is network binarization, which quantizes the weights and activations of a model to only one or two bits. Although binarization offers theoretical operation count reductions of up to 32×, no actual measurements have been reported. This is a symptom of the gap between theory and implementation of binary networks that exists today. In this work, we bridge the gap between abstract simulations and real usable high speed networks. To do so, we identify errors in the existing literature, develop novel algorithms, and introduce Riptide, an open source system that can train and deploy state-of-the-art binary neural networks to multiple hardware backends.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

ACTIVATION: The inputs and outputs of a neural network layer.

BATCH NORMALIZATION: A common type of normalization that is inserted between dense or convolutional layers. Often referred to as BatchNorm [30].

BINARIZATION: Quantization to extremely low bitwidth integers, typically with only 1 or 2 bits.

BITSERIAL: A method for computing n-bit weight m bit activation binary products. A bitserial kernel performs one xnor-popcount matrix multiply for each mn combination of bitplanes.

CNN: Convolutional Neural Network; the dominant architecture used in computer vision. Composed of many consecutive convolutional layers.

COMPUTE LAYER: A dense or convolutional layer; layers that require significant computation.

CONVOLUTIONAL LAYER: A modification of dense layers better suited to vision workloads. Weights are replaced with kernels (although the two terms can be used interchangeably) that are convolved with an input activation tensor to produce an output.

DENSE LAYER: A neural network building block in which activations are matrix multiplied with learned weights to produce an output. Often referred to as a fully connected layer.

FULL PRECISION: The use of 32-bit floating point types to represent a networks weights and activations. State-of-the-art networks are typically trained and deployed at full precision to reach the highest possible accuracy.

GLUE LAYERS: Layers in a network that are found between the compute layers, such as BatchNorm, ReLU and requantization.

PARALLELIZATION: A scheduling primitive that distributes a workload over multiple threads.

QUANTIZATION: The approximation of a network's full precision activations and weights using integer datatypes, most often with 8 bits.

RELU: Rectified Linear Unit; the most common non-linear activation. Sets all negative inputs to 0 while passing positive inputs.

REQUANTIZATION: The quantization of a tensor that was previously quantized. This is often necessary when floating point operations are used on the output of a quantized layer.

SCHEDULING: Techniques that determine how an algorithm reads from memory and executes instructions.

TILING: A scheduling primitive that accesses memory in small windows, often improving locality.

VECTORIZATION: A scheduling primitive that utilizes a piece of hardware's ability to execute multiple copies of an instruction in a single clock cycle.

WEIGHT: The learnable parameters of a neural network.

XNOR-POPCOUNT: When 1-bit binarization is used, multiply-accumulate operations can be replaced with packed xnor (which is equivalent to multiply) and popcount (equivalent to accumulate). This trick is the key to achieving large speedups with binarization.

2A1W: Shorthand to refer to the quantization of a network. In this case referring to a network with 2-bit activations and 1-bit weights.

Chapter 1

INTRODUCTION

1.1 Overview and Motivation

The foundation of computer vision research was thoroughly shook in 2012, when AlexNet [35], a deep Convolutional Neural Network (CNN) [36], dominated the ImageNet challenge [15]; significantly outperforming the old guard techniques based on feature extraction and support vector machines [58] [44]. Since then, new models have been developed at a dizzying pace and state-of-the-art accuracy has risen considerably each year. With CNNs now outperforming humans in vision classification tasks [59], it is clear they will remain a mainstay of machine learning applications in the forseeable future.

However, the strides made in CNN performance haven't come without a cost. As accuracy has improved over the last several years, so too has the computational cost of running models. The relationship between model accuracy and runtime cost is visualized in Figures 1.1 and 1.2 respectively. Unsurprisingly, models that require tens of billions of operations for each inferred frame demand high performance hardware to train and deploy at reasonable speeds. Typically, CNNs are trained using cluster of GPUs: hardware specialized for massively parallel computation that can cost thousands of dollars each. Although easily justified for training, since it is a one time cost, GPUs are simply too expensive to be used for deployment. Unfortunately, more accessible hardware such as CPUs are too slow for many applications. This conundrum has made deploying neural networks quite difficult and has limited the ability of CNNs to be integrated into applications outside of research settings.

The high computational demands of CNNs have driven significant attention to the approximating optimization of models. Approximating optimizations are those that replace a function with one with approximately similar semantics but better performance. A few

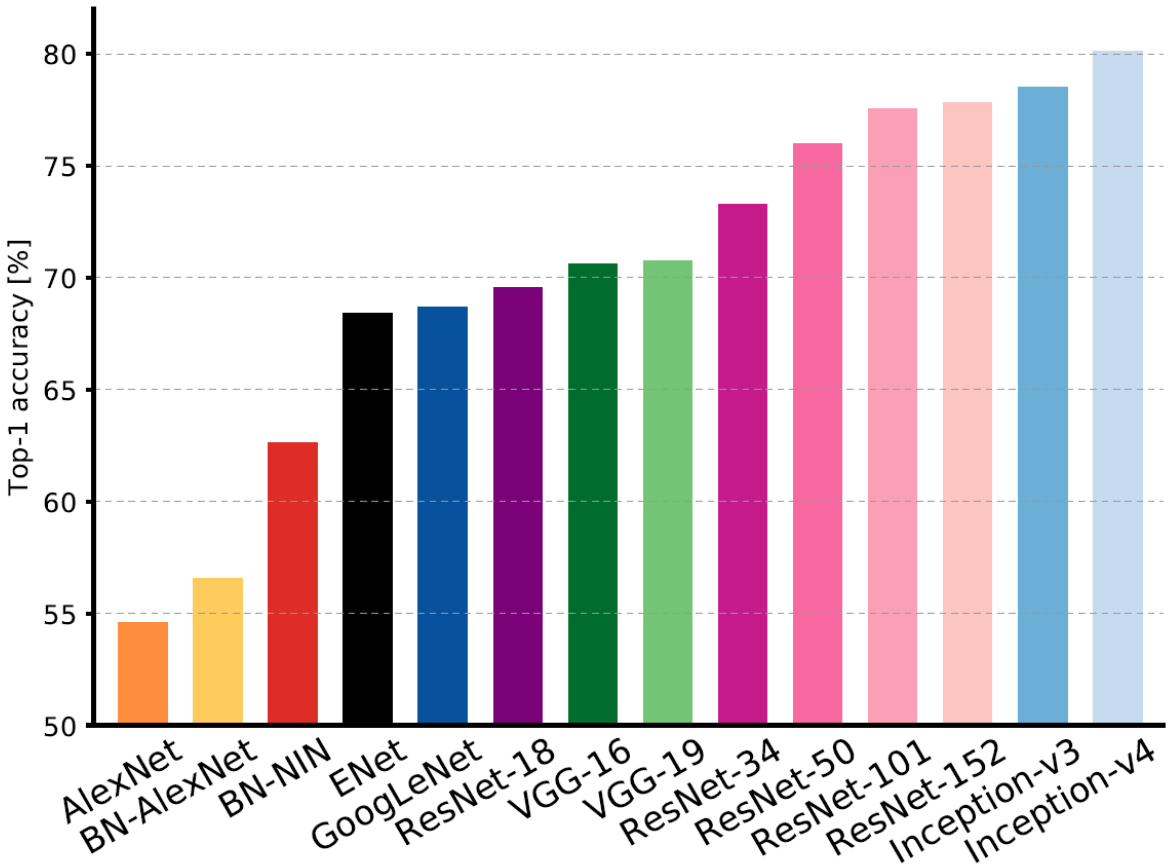


Figure 1.1: Top-1 Accuracy of various models, from Canziani et al. [5]

samples of approximating optimizations that have yielded good results are:

- Hashing [8], which groups weights into buckets and accesses them using a hashing trick.
- Vector compression [21], which uses k-means clustering to factorize weights and reduce dimensionality.
- Weight pruning [22], which removes small magnitude weights from the network.

One particularly promising track of approximating optimization is network binarization [12], which replaces the 32-bit floating point values that are Typically used for a models

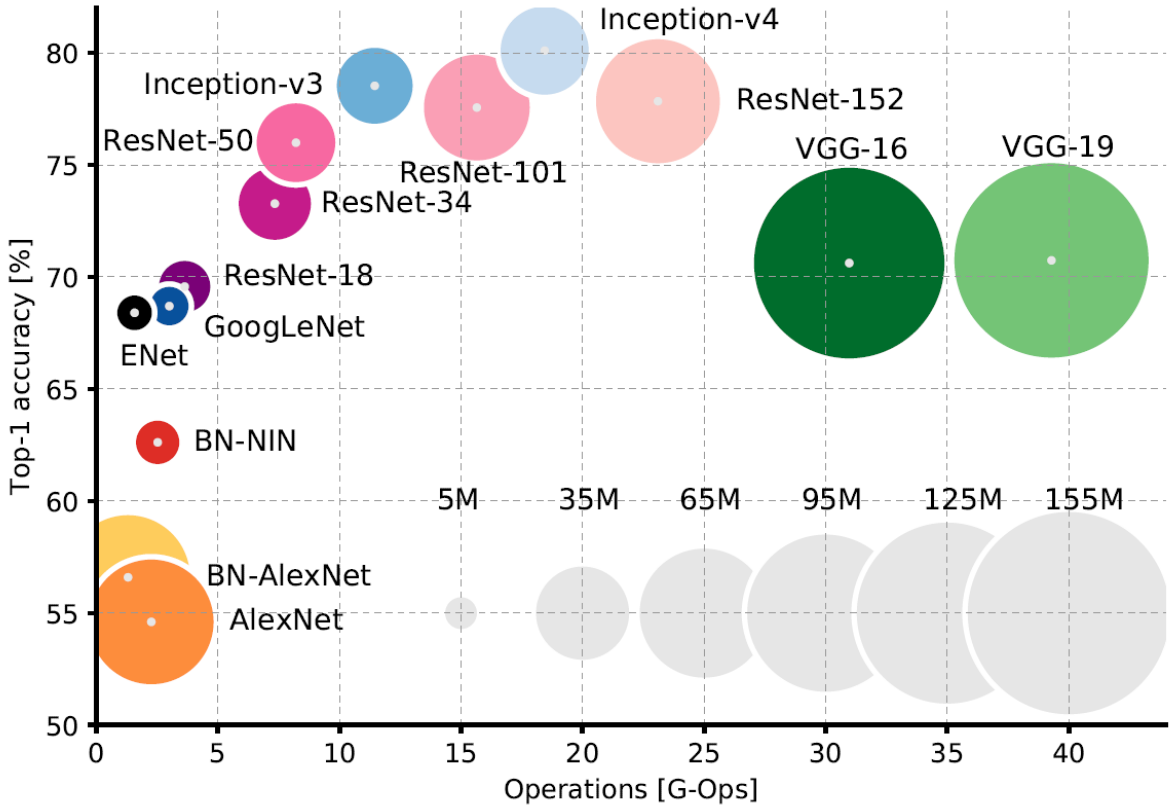


Figure 1.2: Accuracy compared to number of parameters and compute requirements for various models. Note that model accuracy correlates directly with number of operations required. From Canziani et al. [5]

weights and activations with single bits that represent +1 or −1. Not only does this highly lossy approximation reduce model size by a factor of up to 32×, it also allows expensive multiply-accumulates to be replaced by packed bitwise operations that can reduce the number of operations in a model by up to 64×. Unfortunately, such substantial gains don't come for free. Binarization is a highly lossy approximation that significantly reduces model accuracy. Minimizing the gap between binarized models and their full precision equivalents has been the primary focus of research in the field and a slew of recent work has made impressive strides towards this end [13, 64, 4, 27, 60, 17, 20, 10].

Despite considerable progress in developing more accurate models with low theoretical instruction counts, we are aware of no work that has realized measured performance gains on real-world processors. Measurable performance gains on binarized models are hard to achieve for three main reasons. First, for many recent higher-accuracy training techniques, no efficient implementation has been proposed. In some cases (e.g., where bits are scaled using non-linear scaling factors [4]), it is unclear that such implementations even exist. Second, current work has focused on schemes for binarizing the "core" convolutional and fully-connected layers. However, once the dramatic gains of binarization on these layers has been realized, the layers between convolutions ("glue" layers such as batch normalization, scaling and (re-) quantization) become bottlenecks. Bitserial implementations of these layers have traditionally been ignored. Finally, existing floating-point implementations have been carefully scheduled for various processor architectures over many decades via libraries such as BLAS, MKL and CuDNN [62, 29, 9]. No corresponding libraries exist for low-bitwidth implementations. The "number-of-operations" speedup above does not therefore translate to wallclock-time speedups and the true efficacy of binary networks is thus currently unknown.

Another key issue with existing binary networks is their inflexibility. The primary way that a practitioner can trade off speed-ups for higher accuracy in a binary network is to add additional bits, thereby giving the approximation of the true values higher fidelity. However, the cost of extra bits is quite high. Using n bits to approximate just the weights increases the computation and memory required by a factor of n compared to 1-bit binarization. Further using n bits to approximate activations as well requires n^2 times the resources as one bit. There is thus a strong motivation to use as few bits as possible while still achieving acceptable accuracy. However, today's binary approximations are locked to use the same number of bits for all approximated values, and the gap in accuracy between bits can be substantial. For example, recent work concludes 1-bit accuracy is unsatisfactory while 2-bit accuracy is quite high [60].

1.2 Thesis Statement and Research Contributions

The current state of network binarization is insufficient to support its use in any end-to-end system. This motivation directly introduces our thesis statement:

> **By introducing novel quantization algorithms, leveraging recent scheduling tools, and rethinking end-to-end implementation, binarization can be made an effective and practical tool for deploying high speed and accurate neural networks.**

The specific research contributions contained with this thesis are as follows:

- *Techniques for End-to-End Binarization*: We examine existing binarization methods and consider their deployment implications. We find many state-of-the-art methods have fundamental flaws or require new algorithms to be efficiently run. We identify limitations in existing architectures and introduce a new "fused glue" layer that enables truly binary end-to-end models.

- *High Speed Library*: We detail the development of a library of highly performant binary functions and thoroughly evaluate their application to multiple models. We additionally examine several real-world use cases of binary models and demonstrate their Pareto optimality.

- *Heterogenous Bitwidth Networks*: We introduce a novel algorithm for binarizing a model to a fractional bitwidth, allowing much finer grain control of the speed to accuracy tradeoff offered by binarization. We additionally show that this technique provides superlinear accuracy scaling with bitwidth.

Chapter 2
BACKGROUND AND RELATED WORK

Model quantization and binarization is a popular field that has attracted significant research attention. Although the vast majority focuses on closing the accuracy gap between binary models and their full precision counterparts, understanding the various algorithms used helps to motivate our work. In this chapter we provide a brief overview of background information relevant to binarization and discuss the chronology of related work along with state-of-the-art approaches.

2.1 A Brief History of Neural Networks

Neural networks have come a long way from their humble single neuron origins [53] and today see themselves used in many applications. Although research in neural networks has been a relatively active field for decades, until recently use cases tended to be limited and tools such as support vector machines [58] were vastly more popular. Indeed, in the early 2000s prospects were grim for the future of neural networks. That all changed in 2012, when a conjunction of technological spheres foisted neural networks back into the limelight. The dominance of AlexNet [35], an 8 layer neural network, over more in vogue algorithms in the ImageNet challenge [15] kicked off the deep learning craze that continues to drive the rapid progress in the field today.

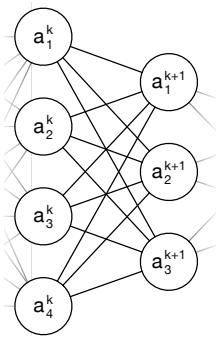
But why did it take over 50 years for neural network to see such success? Deep networks, it turns out, required three foundational technologies to become viable: huge sets of training data, effective techniques to learn parameters, and high speed hardware to make the duration of training feasible. ImageNet, a collection of over one million images each labelled with one of one thousand classes, was introduced in 2009 [15]. AlexNet has roughly

60 million trainable parameters. On a less substantial dataset, such a large model would very quickly and dramatically overfit. Before ImageNet, there simply weren't large enough public datasets to support deep networks. However, the large size of ImageNet introduced other problems, one of which was how to compute optimal parameters given so many training images. Traditional solvers that directly computed parameters based on the entire dataset simply were too computation inefficient to scale to ImageNet proportions. Fortunately, just a year after ImageNet's release, Stochastic Gradient Descent (SGD) began being applied to machine learning algorithms [3]. SGD operates on small batches of random samples at a time and computes updates of a networks parameters via gradient calculation that reduce error on that batch. When run on a large datasets for many epochs (one full iteration through all samples of the dataset), the parameters slowly improve and the loss descends to healthy minimum. Unfortunately, training a network using SGD often requires hundreds of epochs, and when each epoch consists of processing a million images, the prospect of training and adjusting a deep network becomes daunting, and would likely be completely infeasible on a CPU. In 2007, Nvidia released CUDA, a programming model that allowed programs to be run on GPUs, massively parallel computation devices designed for graphical applications. Conveniently, the matrix multiplies that form the basis of all neural network computation are very well suited to parallelization. By leveraging CUDA, AlexNet was able to train in weeks rather than months.

Since AlexNet, deep networks have proliferated quickly and today dominate most machine learning applications. Uses of deep neural networks include visual tasks like classification and detection [49], text understanding [16], generating realistic speech [32], and even more exotic applications like compressing images and videos [50, 51]. Throughout all deep learning applications and variants of networks, the vast bulk of computation is matrix multiplies.

2.2 The Structure of Deep Neural Networks

A single node in a neural network is composed of a set of learnable weights that extract information from a set of input features. The output of that node is referred to as an



$$\begin{bmatrix} w_{11}^k & w_{21}^k & w_{31}^k & w_{41}^k \\ w_{12}^k & w_{22}^k & w_{32}^k & w_{42}^k \\ w_{13}^k & w_{23}^k & w_{33}^k & w_{43}^k \end{bmatrix} \begin{bmatrix} a_1^k \\ a_2^k \\ a_3^k \\ a_4^k \end{bmatrix} = \begin{bmatrix} a_1^{k+1} \\ a_2^{k+1} \\ a_3^{k+1} \end{bmatrix}$$

Figure 2.1: In a fully connected neural network, computing the activations for layer $k+1$ involves a matrix-vector multiplication, which reduces to a series of vector dot products.

activation. In a neural network, many such nodes are combined one after another, with each node now being referred to as a layer. In a fully connected (often referred to as a dense or FC) layer, each weight is multiplied by each incoming activation to produce an output. Using matrix notation, we can write the computation performed in an FC layer as the matrix-vector multiplication $\mathbf{A_{k+1}} = \mathbf{A_k W_k}$, which is visualized in Figure 2.1. For activations A with shape K and weights W with shape N by K, an FC layer requires $2NK$ operations as each weight must be multiplied by each activation and accumulated. In many networks, N and K have values on the order of thousands and so each FC layer in a network will take millions of operations.

Because fully connected layers scale both their computation requirements and number of parameters based on the size of the input tensor, they are not well suited to computer vision applications, which typically have large inputs (images). Instead, convolutional layers are used that spatially slide a learned kernel over input tensors. The first "deep" convolutional network was introduced by Lecun et al. [36] and is visualized in Figure 2.2. The inputs to convolutional layers are 3 dimensional tensors of shape HWC where H is the height of the tensor, W is the width, and C is the number of channels (3 for an RGB image for example). The learnable parameters in convolutional layer are often called kernels and have shape FK^2C where F is the number of kernels, K is the receptive field (often 3), and C is

again the number of input channels. Each step of the convolution (where a step refers to one position of the kernel as it slides across the input tensor) is a matrix-vector product with FK^2C multiply-accumulates that produces one "pixel" of the output tensor. The output shape of the layer is determined by the padding (extra zeros inserted around the input features) of the input and the stride of convolution but is often similar in dimension to the input tensor. In this case, there are a total of $HWFK^2C$ multiply-accumulates. Algorithm 3 provides a detailed description of a convolutional model and the core algorithm.

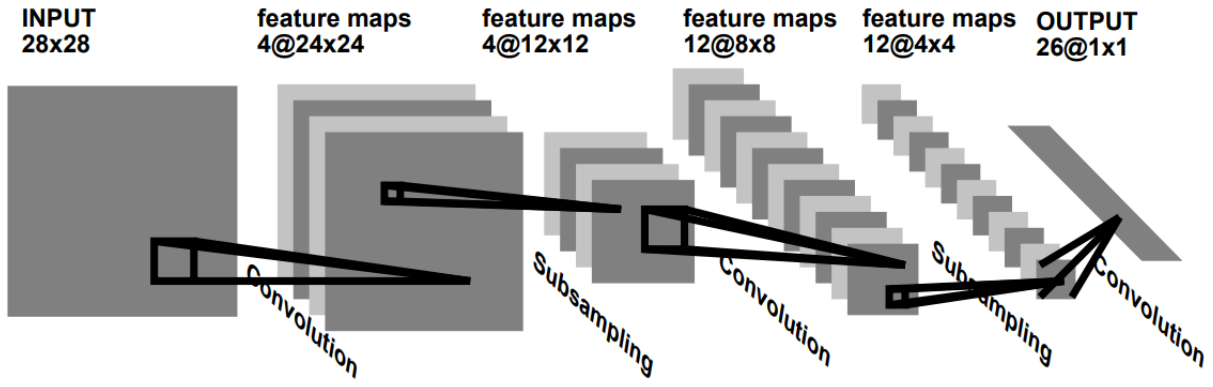


Figure 2.2: The original convolutional neural network in all its glory. Visualizes how learned kernels convolve over an input to generate the output tensor. From Lecun et al. [36]

2.3 Training Versus Inference

There are two very distinct modes in which neural networks are used: training and inference. Much of the work presented in this thesis is in the efforts of optimizing inference rather than training. To understand why we focus on inference it is useful to review the life cycle of a neural network. When a new neural network is developed or an existing architecture is applied to a new application, it must be trained using the procedure described in Algorithm 1. The training loop steps over each sample in a dataset and iteratively refines the network's learnable parameters. One full iteration over the dataset is referred to as one epoch. For a

Algorithm 1: Procedure for training a neural network

Input: Input features X and labels Y from dataset D, learnable parameters W and b, learning rate η

Output: Updated weights W and b

1 **for** $X, Y \in D$ **do**

    `// Compute prediction of network using current parameters`

2 $\hat{Y} = f(X)$

    `// Compute loss between prediction and target`

3 $L = \ell_2(\hat{Y}, Y)$ `// Update parameters using backpropagated gradients`

4 $W = W + \eta(\frac{\partial f}{\partial W} L)$

5 $b = b + \eta(\frac{\partial f}{\partial b} L)$

6 **end**

network designed to perform well on ImageNet, it often takes hundreds of epochs for accuracy to plateau. On top of this, researchers often tune the model by adding layers or changing hyperparameters such as learning rate and must retrain the model with each variation.

Once trained, the network is used for inference (line 2 of Algorithm 1) in whatever application it is designed for. Given that inference requires so much less computation than training, and in fact is itself just one step of training, it may be surprising that in this work we focus only on inference optimization. However, training is a one-time cost while inference scales with uses of the application and remains relevant as long as the model is used in short "train once, run billions of times". Additionally, training typically happens on a single machine, or at least by a single individual, while inference is often distributed across many users. These factors make it appealing to perform training on clusters of expensive GPUs that are extremely efficient at computing neural network workloads. At inference time though, the cost of GPUs cannot be justified in most cases. Thus, the potential impact of optimizing training exists but in our opinion is dwarfed by the need for efficient inference.

All work presented in this thesis focuses only on improving the speed and model size during inference. Although many of the techniques we present could be applied to optimize training, we leave this as future work.

2.4 Binary Neural Networks

First introduced by Courbariaux et al. [13], network binarization attempts to optimize networks by quantizing 32-bit full precision networks to only 1 bit. In a binary network, both weights and activations of the network are quantized to a single bit representing +1 or -1 ($q = \text{sign}(x)$). Of course, there is immediate value in compressing the size of a model's parameters by a factor of 32×, which not only enables the network to consume less memory when running but also may improve speed by reducing the cost of accessing memory. However, the compression of a model requires only that weights are binarized while activations could be left full precision, and indeed some models may use this technique solely for the benefits to memory. When activations are also also quantized to 1-bit representing +1 or -1, the possible combination of weight-activation products becomes quite limited, and can be enumerated as in Equation 2.1.

$$\begin{cases} 0 \times 0 = 1 \\ 0 \times 1 = 0 \\ 1 \times 0 = 0 \\ 1 \times 1 = 1 \end{cases} \tag{2.1}$$

We note that this enumeration is identical to the truth table of an XNOR gate. This equivalence allows multiplications between binarized weights and activations to replaced with bitwise XNOR operations. Unfortunately it is very rare for hardware to support 1-bit datatypes. To efficiently perform bitwise XNOR and accumulation, the bits must be packed into a larger datatype, typically the largest integer type available such as Int64. The definition of the bitpacking procedure is given in Algorithm 2 and visualized in Figure 2.3.

Algorithm 2: Method for packing bits in preparation for binary convolution.

Input: An N-bit quantized tensor of integers Q that should be packed prior to a binary convolution.

Output: BP, a tensor of N sets of packed Int64 planes ready for xnor-popcount based computation.

```
for n = 0 to N − 1 do
    BP_n = 0
    for i = 0 to len(Q)/64 do
        for j = 0 to 63 do
            BP_n(i) |= (Q(i ∗ j) ∧ n) ≪ (j − n)
        end
    end
end
```

After being packed into an Int64, a single bitwise-XNOR operation does the work of 64 equivalent floating point multiplies. Accumulation is replaced by the POPCOUNT operation, which returns the number of set bits in an integer. In this work, we refer to matrix multplications that use this approach as "bitserial". The number of operations (and the theoretical runtime) in the inner loop of a bitserial matrix multiplication reduces from $2C$ to $\frac{3C}{64}$, a reduction of 43×. An example of a packed bitserial dot product is visualized in Figure 2.4.

Memory compression of 32× and speedups of 43× might sound too good to be true. Indeed approximating a 32-bit value down to a single bit causes catastrophic accuracy loss. Early binary networks were only performant on small toy datasets like CIFAR10 [34] and were unable to reach usable accuracies on realistic datasets like ImageNet. Enticed by the promises of huge performance gains at low accuracy loss, all followup work to the early binarization papers has been focused on closing the accuracy gap.

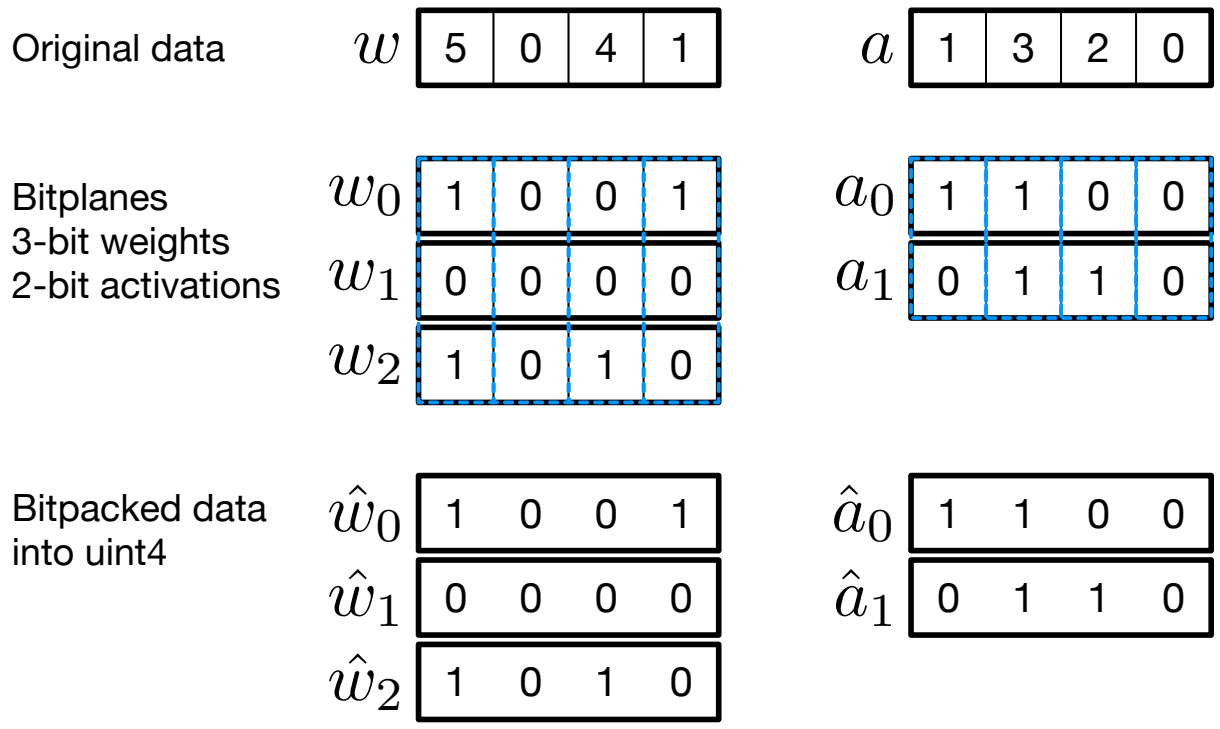


Figure 2.3: Visualization of how quantized data can be split into bitplanes and packed into an integer datatype.

2.5 Binary Accuracy Improvement Techniques

Although 1-bit binary models promise significant performance benefits, the accuracy they have been shown capable of achieving on challenging datasets like ImageNet has been underwhelming. For example, the AlexNet [35] based BNN used by [13] was only able to reach a top-1 accuracy of 27.9% when trained on ImageNet compared to the full precision models 56%. The significant accuracy loss that comes with network binarization has been the focus of research in the space, with most papers introducing modifications to the core algorithm or new training techniques.

One key modification towards higher accuracy binary networks was the addition of the *straight-through estimator*, defined in Equation 2.2, which passes through gradients corre-

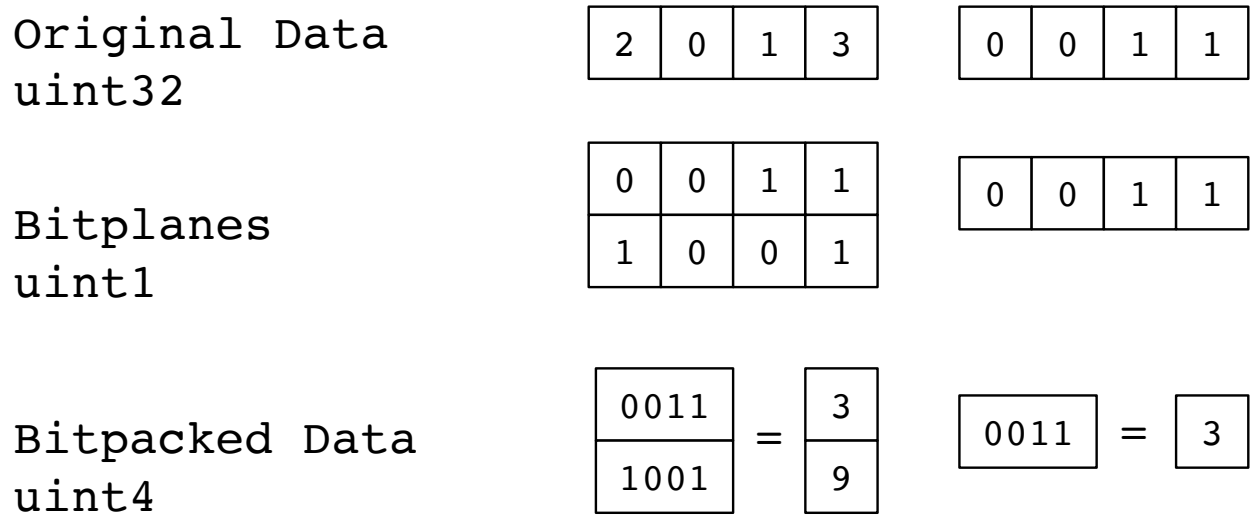


Figure 2.4: Visualization of a bitserial product using 2-bit activations and 1-bit weights. Note that because we pack data and use efficent popcount-and operations, we can compute the output in only 6 instructions instead of 32.

sponding to inputs with magnitude less than one but clips gradients for larger inputs. The logic behind the straight-through estimator is that only values close to zero can impact the output when changed.

$$\frac{dT^B}{dT} = 1_{|T|\leq 1}. \tag{2.2}$$

Because quantization is a rounding operation, it has a derivative of zero at all points. Since neural networks use gradients to learn parameters, this is something of a deal breaker. Early networks redefined the gradient to simply be one instead, however, the mismatch between the forward and backward passes resulted in the low accuracy of training attempts. By replacing the gradient of binarization with the straight-through estimator, it became a much

more accurate reflection of the forward function. Today, the straight-through estimator or some slight variant is used in virtually all binarization algorithms.

[48] introduced XNOR-Net, which improved the accuracy of single bit binary models by adding the WeightScale function on line 5 of Algorithm 5. The term $\alpha_k = \text{mean}(|W_k|)$ was multiplied into the binary convolution output, where W_k are the weights of one of the convolutional layer's filters. Weight scaling proved extremely useful for preserving both the magnitude and relative scale of weights. The authors additionally noted that applying batch normalization directly before quantization ensures maximum retention of information due to the centering around zero. These subtle but important changes allowed an XNOR-Net version of AlexNet to reach 44.2% accuracy on ImageNet.

Although XNOR-Net offered a substantial improvement to accuracy, follow-up works noted that even so, the accuracy achievable with 1-bit activations is simply not compelling and instead focus on using $N \geq 2$ bits. [27] and [64] introduce QNN and DoReFA-Net respectively, both of which use 2-bit activations to achieve higher accuracy on ImageNet. Both works used very similar techniques and had similar results. Here we'll discuss DoReFa-Net's multi-bit implementation as it is more precisely defined. Like XNOR-Net, DoReFa-Net quantizes *weights* using $q = \text{sign}(x)$ and uses weight scale term α. *Activations*, on the other hand, are quantized into linearly spaced bins between zero and one (Equation 2.3). DoReFa-Net uses $\text{clip}(x, 0, 1)$ as activation function, ensuring outputs from Equation 2.3 are less than or equal to one. DoReFa-Net was able to reach an AlexNet top-1 accuracy of 50%, closing quite a bit of the gap between binary and floating point models.

$$\begin{aligned} q_{\text{bits}} &= \text{round}((2^N - 1) * x) \\ q_{\text{approx}} &= \frac{1}{2^N - 1} q_{\text{bits}} \end{aligned} \tag{2.3}$$

[4] introduce Half Wave Gaussian Quantization (HWGQ), a new Quantize function that enables 2-bit activation binary networks to achieve the highest reported AlexNet accuracy (52.7%) to our knowledge. HWGQ uses the same weight quantization function as XNOR-Nets and DoReFa-Nets, quantizing to a single bit representing -1 or 1 and adding scale factor

α. To quantize the networks activations, the authors note that the output of ReLU tends to fit a half Gaussian distribution. The authors suggest that quantization functions should attempt to fit this distribution. To this end, HWGQ uses k-means clustering to find $k = 2^N$ quantization bins that best fit a half Gaussian distribution.

Although the original interest in binarization was due to its potential to enable high speed and low memory models without sacrificing too much accuracy, all follow up work has been focused on reducing the accuracy gap rather than realizing high speedups in practice. To our knowledge, no work has reported a measured end-to-end speedup or described in detail the techniques required to yield one. Additionally, all approaches to date only allow small, discrete number of binarization levels, which makes choosing the right accuracy to speedup tradeoff difficult. To address the lack of efficient binary implementations, we present Riptide in Chapters 3 and 4. In Chapter 5, we present Heterogeneous Bitwidth Neural Networks (HBNNs) that allow a tensor to be binarized with a *continuous* bitwidth and thus dramatically expanding binarization's price-point flexibility.

Chapter 3

END-TO-END BINARY NETWORKS

The focus of much current research in binarization has been on reducing the accuracy gap between binary models and their full-precision counterparts. Researchers usually provide rough estimates of instruction count reductions in linear algebra operations to justify their approach. These estimates ignore whether the algorithms add significant computational overhead beyond matrix operations, interact poorly with other algorithms, and how to efficiently implement them on modern processors. This trend has caused many errors and oversights in the field, which both make it difficult to know what the best choice of algorithm is and prevent fast end-to-end models from being developed. In this chapter, we dig into the existing approaches and consider their viability, we uncover and address several fundamental algorithmic issues, and introduce a new type of "fused glue" operation that enables truly binary end-to-end networks. In our discussion of the various operations of binary networks and how they can be optimized or removed, we will often refer to Algorithms 3 and 5, which define the forward pass of full precision and binary networks respectively. We also consider the core computation loops of convolution in Algorithms 4 and 6. Collectively, we refer to our innovations, algorithms, and design choices as Riptide, which we believe is the first system to allow the training and deployment of fast binarized neural networks in practice.

3.1 Implementing core operations efficiently

The first step in building a binary network is choosing a quantization method. Although it may seem adequate to pick the method with the highest accuracy, it is often challenging to implement the most accurate models in a bitserial fashion (i.e., using logical operations on packed bit-vectors as described above). In particular, proposed algorithms often achieve

Algorithm 3: Typical CNN forward propagation. Lines marked $\star$ are glue layers that use floating point arithmetic.

Input: Image X, network with L convolutional layers

Output: Y, the predictions of the network

1 $c_0 = \text{Conv}(X)$ `// Input layer block`

2 $a_0 = \text{BatchNorm}(c_0)$

3 **for** $k = 1$ ***to*** L **do**

  `//` $\texttt{shape}(a_{k-1}) = HWC$ `and` $\texttt{shape}(L_k) = KKFC$

4 $c_k = \text{Conv}(a_{k-1})$ `//` $KKFHWC$ `ops`

5 $p_k = \text{Pooling}(c_k)$ `//` HWF `ops`

6 $b_k = \star\text{BatchNorm}(p_k)$ `//` $4HWF$ `ops`

7 $a_k = \text{Activate}(b_k)$ `//` HWF `ops`

8 **end**

9 $Y = \text{Dense}(a_L)$

higher accuracy by varying *bit consistency* and *polarity* so as to trade off accuracy for bitserial implementability.

Lines 3 and 4 of Algorithm 6 describe the inner loop of bitserial convolution when values are linearly quantized, as in Equation 2.3. For $n > 2$, the term 2^n (which can be implemented as a left shift) adds the scale of the current bit to the output of popcount before it is accumulated. This is possible because the spacing between incremental bits is naturally linear. Using a non-linear scale would require replacing the efficient shift operation with a floating point multiply.

Additionally, it is imperative that the values of bitwise arithmetic be consistent with the value represented by those bits. For example for $N = 2$, the value of the sum of bit pairs 01 and 10 must equal the value of bit pair 11. If this is not the case, values are effectively being assigned to bits that are conditional on their bit pairs. However, POPCOUNT performs a

Algorithm 4: Definition of Conv function.

Input: Activations A of shape HWC and F kernels W each of shape KKC

Output: c_k, the output activations of the convolution

for $i = 0$ ***to*** K, $j = 0$ ***to*** K **do**
  for $c = 0$ ***to*** C **do**
    c_{hwf} += $A[h+i][w+j][c] * W_f[i][j][c]$
  end
end

Algorithm 5: Convolution block that replaces Conv in Algorithm 3 for an N-bit binary network. Lines marked $\star$ are glue layers that use floating point arithmetic.

Input: Activation tensor x

Output: Output activation tensor y

$q = \star\text{Quantize}(x)$ `// At least 5HWC ops.`

$b = \text{BitPack}(q)$ `// At least 3HWC ops.`

$c = \text{BitserialConv}(b)$ `//` $\frac{NKKFHWC}{43}$ `ops.`

$f = \star\text{Dequantize}(c)$ `// 4HWF ops.`

$y = \star\text{WeightScale}(f)$ `// HWF ops.`

reduction over bitplane before scales can be applied. Using a representation that does not have bit consistency would require multiplying each x_{hwf} by a scaling constant and prevent the use of popcount, removing any reduction in computation benefits that quantization otherwise offers. High accuracy binarization techniques that attempt to better fit non-linear distributions by dropping bit consistency such as HWGQ are thus difficult to implement efficiently.

Quantization polarity describes what the bits of a quantized tensor represent. In **unipolar quantization**, bits with value 0 represent 0 and bits with value 1 represent 1. Conversely, in **bipolar quantization** bits with value 0 represent -1 and bits with value 1 represent 1.

Algorithm 6: Definition of BitserialConv.

Input: Q, a quantized tensor with unpacked size HWC and W a quantized tensor of weights with unpacked size $FKKC$. Both inputs are packed along the channel axis into in64 chunks

```
for n = 0 to N, i = 0 to K, j = 0 to K do
    for c = 0 to C/64 do
        x_hwf = Q_n[h + i][w + j][c] ⊗ W_f[i][j][c]
        y_hwf += 2^n popc(x_hwf) // Sum over bitplanes.
    end
end
y_hwf = 2y_hwf − KKC
```

Early binarization models such as XNOR-Nets and QNNs use bipolar quantization for both weights and activations due to the ability of the xnor operation to elegantly replace multiplication in the inner loop of a binary convolution. Because bipolar quantization must be centered around zero, it is not possible to actually represent zero itself without breaking linearity. Not only does zero have some intrinsic significance to activations, but it also is ubiquitously used to pad convolutional layers. In fact, *this padding issue prevents QNNs and XNOR-Nets from being implemented as proposed by their authors.*

Methods that use unipolar quantization for activations such as DoReFa-Net and PACT-SAWB [10] are able to represent zeros but encounter other implementation issues. Because weights are always bipolar due to their need to be capable of representing inverse correlation (negative numbers), the unset bits in a quantized weight tensor represent -1 while the unset bits in quantized activation tensor represent 0. This polarity mismatch prevents a single bitwise operation and popcount from producing a correct result since the bits effectively represent three values instead of two. The current literature does not provide an answer to this issue and it is not clear how to efficiently and correctly implement mixed polarity

models.

It is worth noting that there also exist Ternary Weight Networks [37] that use bipolar quantization and a mask tensor that specifies some bits as representing 0. Although ternary quantization is able to represent both zero and negative numbers, it is effectively using an extra bit via the mask tensor to do so. Instead of being able to represent 2^N unique values, ternary quantization can only represent $2^{N-1} + 1$ values. This loss of expressiveness leads to ternary networks not having competitive accuracy with state-of-the-art unipolar models.

Attempting to navigate these numerous complications and implement an end-to-end system could easily lead to poor performance or incorrect output values at inference time. In Section 4.3.4 we examine the impact of polarity on runtime and describe the quantization scheme used in Riptide.

3.2 Binarizing glue layers

In a typical floating point model the vast bulk of computation goes into the core convolutional and dense layers. The interlayer glue operations such as non-linear activations, MaxPooling and BatchNormalization are so minimally compute intensive compared to core layers that their impact on runtime is ignored. However, in BNNs the number of operations in core layers is so greatly reduced that the time spent in glue layers actually becomes a major bottleneck.

To demonstrate the effect of glue layers, we consider the total number of operations in a binarized SqueezeNet [28]. We count the number of operations in all bitserial convolution layers at various assumed speedups and compare those counts to the total number of glue operations. These estimates are visualized in Figure 3.1. We see glue layers make up a whopping 70% of the operations in a network given the optimal reduction in number of operations offered by binarization. Even assuming smaller speedups in practice, glue layers contribute a substantial fraction of the total estimated runtime at all scales where the speedups of binarization can justify its impact on accuracy.

Figure 3.1 makes it readily apparent that a high speed end-to-end implementation must

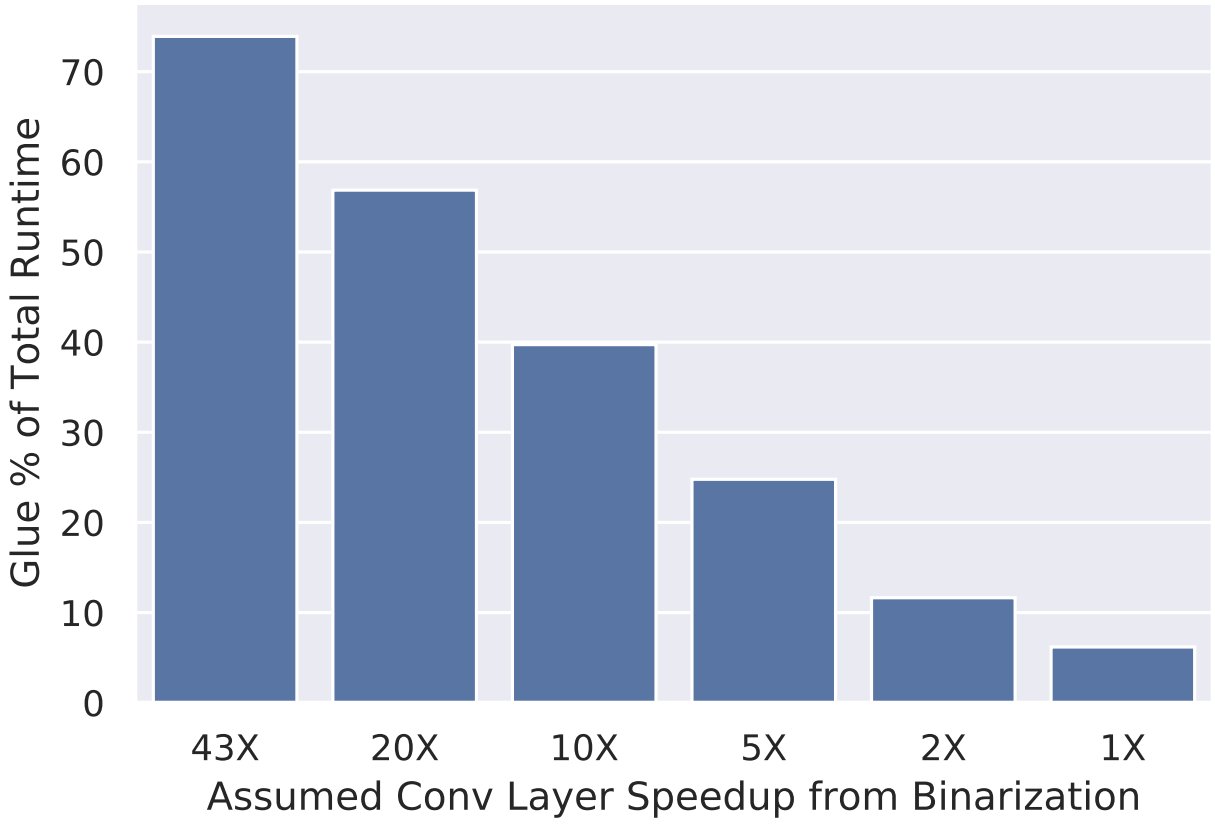


Figure 3.1: End-to-end speedup of SqueezeNet with fixed glue layer costs and theoretical speedups of convolution layers.

minimize or all-together remove glue layers. However, all high accuracy binarization techniques today rely on BatchNormalization and weight scaling in the floating point domain. The centering and normalization effects of BatchNormalization are essential to generating consistent and accurate quantized activation representations and weight scaling has been shown to dramatically increase model accuracy by allowing the magnitude of weight tensors to be efficiently captured. Because these layers require floating point arithmetic, interlayer type casting and requantization must also be inserted. To address this bottleneck, we introduce a novel fusible operation that completely removes the cost of glue layers and yields a speedup of multiple factors without loss of accuracy in Section 4.3.6 .

3.3 System Design

In this section we discuss the methods Riptide uses to overcome the challenges raised in Section 3.1, allowing it to generate fast end-to-end binary models. All the following described innovations are implemented and supported in both TensorFlow [1] and TVM [6]. We use TensorFlow for training binary networks and TVM for compiling efficient machine code. The combination of these two halves makes Riptide an effective one stop solution to training and deploying binary networks.

3.3.1 Quantization Technique and Polarity

As discussed, quantization methods that are not bit-consistent have fundamental issues being implemented in a bitserial way. As bitserial computation is essential to realizing speedups, we are forced to use one of the bit-consistent techniques. It remains an open question whether bit-inconsistent binarization can be implemented efficiently. We choose to use linear quantization in the style of Equation 2.3 as it does not require any floating point multiplication in its inner loop and has been shown to yield high accuracy models. However, there remain major barriers to supporting both it's bipolar and unipolar variants. To provide a deeper understanding of the impact of polarity, and offer as fine a granularity as possible in the trade-off between speed and accuracy, Riptide supports both unipolar and bipolar

quantization.

Supporting unipolar activation quantization requires solving the polarity mismatch described in Section 3.1. There are a few possible solutions to this dilemma. Perhaps the most direct solution would be to get rid of the polarity mismatch by quantizing both activations *and* the weights unipolarly. Although this would allow a fast implementation, it would also require that weight values be strictly positive. Because weights represent correlation with specific patterns, removing negative weights is similar to preventing a network from representing inverse correlation, which is highly destructive to accuracy.

Instead, we can treat the weight values *as if* they're unipolar. Then, the bitwise-and operation between activations and weights is correct except when the activation bit is 1 and weight bit is 0. In this case, the product should have been -1 but is instead 0. To handle these cases, we count them and subtract it from the accumulation. This solution is given in Equation 3.1

$$a \cdot w = \sum_{n=0}^{N-1} 2^N (popc(a_n \wedge w) - popc(a_n \wedge !w)) \tag{3.1}$$

Here, we use two popcounts and bitwise-and operations and a bitwise invert (!) instead of the single popcount-xnor used in bipolar quantization. While the unipolar representation requires double the compute of the bipolar representation, the number of memory operations is the same.

Bipolar activation quantization can be implemented as described in Algorithm 5, but requires padding with -1 instead of the ubiquitous 0. Ideally, this would be a simple argument swap, however, frameworks such as TensorFlow do not support constant non-zero padding. Although it is possible to implement a custom padding operation that directly pads with non-zero constants, Algorithm 7 can also be used to simulate padding with -1 during training.

Algorithm 7: Simulated padding with non-zero constants.

Input: Activation tensor A, weight tensor W with f filters, and padding constant p.

Output: Activation tensor Y padded with p.

1 $\hat{A} = A - p$

2 $\hat{Y} = \text{Conv}(\hat{A}, W)$

3 $p_f = p * \text{sum}(W_f)$

4 $Y = \hat{Y} + p_f$

3.3.2 Fused Binary Glue

Figure 3.1 demonstrates that the significant speedups (up to 43×) offered by binary layers pushes the cost of convolution and dense layers so low that it causes glue layers (lines marked with $\star$ in Algorithms 3 and 5) to become a major bottleneck at inference time. We seek to replace each such glue layer with bitserial operations. To this end, we introduce a novel operator that completely replaces all floating point glue layers while requiring only efficient bitserial addition and shifting. This new **Fused Glue** operator allows Riptide to simplify the forward pass of a binary model to the definition in Algorithm 8, where line 5 is our fused glue operation. Here we introduce the fused glue layer and explain how it works.

The glue layers in a traditional binary network perform three key functions: the WeightScale operation in line 5 of Algorithm 5 propagates the magnitude of weights into the activation tensor while the BatchNorm layer in line 6 of Algorithm 3 normalizes and centers the activations. Because these operations require floating point arithmetic, the remaining glue layers exist to cast and convert activations from integer to float and back again. If we could simply remove weight scaling, activation normalization, and activation centering, the rest of the glue layers wouldn't be required. Unfortunately, all three functions are essential to generating high quality quantizations. Instead, we seek to replace these floating point operations with efficient integer versions. Indeed, the three constants wb, sb, and cb in Algorithm 8 represent weight scaling, normalization, and centering terms respectively.

Algorithm 8: Riptide inference with N-bit activations.

Input: Input tensor X, binary layers L, weight scaling bits wb, shiftnorm scaling bits sb, and combined centering term cb.

Output: Binary network predictions Y

$c_0 = \text{NormalConv}(X)$ `// Full precision first block.`

$b_0 = \text{BatchNorm}(c_0)$ $q_0 = \text{LinearQuantize}(b_0)$ $a_0 = \text{BitPack}(q_0)$

for $k = 1$ ***to*** L **do**

  $c_k = \text{BinaryConv}(a_{k-1})$ `//` $\frac{NKKFHWC}{42}$ `ops.`

  $q_k = (c_k + cb) \gg (wb + sb)$ `// 2HWF ops.`

  $l_k = \text{clip}(q_k, 0, 2^N - 1)$ `// HWF ops.`

  $p_k = \text{Pooling}(l_k)$ `// HWF ops.`

  $a_k = \text{BitPack}(p_k)$ `// At least` $3HWF$ `ops.`

end

$Y = \text{BinaryDense}(a_L)$

3.3.3 Weight Scaling

Multiplying the output of a bitserial operation by the scale term $a_k = \text{mean}(|W_k|)$ where k is the number of filters or units in weight tensor W has been a staple of BNNs since it was introduced in XNOR-Nets. This simple modification allows the relative magnitude of weight tensors to be preserved through quantization and gives a dramatic boost to accuracy while adding few operations. To maintain this functionality and preserve the integer domain, we replace weight scaling with an approximate power of two (AP2) bitwise shift. AP2 and its gradient g_x is defined in Equation 3.2.

$$\begin{aligned} AP2(x) &= 2^{\text{round}(log_2(|x|))} \\ g_x &= g_{AP2(x)} \end{aligned} \tag{3.2}$$

This allows us to approximate the multiplying of tensor A with weight scale α as $A \cdot \alpha_k \approx A \gg -log_2(AP2(\alpha_k))$ where $\gg$ is a bitwise right shift. Note that the term $-log_2(AP2(\alpha_k)$

is constant at inference time, so this scaling requires only a single shift operation, which is much more efficient than a floating point multiply on most hardware. However, right shifting is equivalent to a floor division when we'd prefer a rounding division to preserve optimal quantization bins. Fortunately, round$(x) =$ floor$(x + 0.5)$ so we need only add the integer domain equivalent of 0.5 to a_k before shifting. Thus, Riptide's full weight scaling operation is defined in Equation 3.3.

$$\begin{aligned} wb &= -log_2(AP2(\alpha_k)) \\ q(a) &= (a + (1 \ll (wb - 1))) \gg wb \end{aligned} \tag{3.3}$$

Although the addition of the term $(1 \ll (wb-1))$ increases the amount of compute used, we will soon show that it can be fused with a centering constant without requiring any extra operations.

3.3.4 Normalization

We can extend Equation 3.3 to support activation normalization by approximating the variance of the activation tensor using AP2. Then, instead of dividing by activation tensor A by its filter-wise variance σ_k, we can perform a right shift by sb bits, where sb is defined in Equation 3.4. Thus, we can perform a single right shift by $wb + sb$ to both propagate the magnitude of weights, and normalize activations. Equation 3.3 thus becomes Equation 3.4.

$$\begin{aligned} sb &= log_2(AP2(\sqrt{\sigma_k^2 + \epsilon})) \\ q(a) &= (a + (1 \ll (wb - 1))) \gg (wb + sb) \end{aligned} \tag{3.4}$$

We keep track of the running average of variance during train time so that the term $wb + sb$ is a constant at inference time.

3.3.5 Centering

Finally we extend Equation 3.4 to center the mean of activations around zero. The simplest way of centering a tensor is by subtracting it's mean. Because this is a subtraction rather

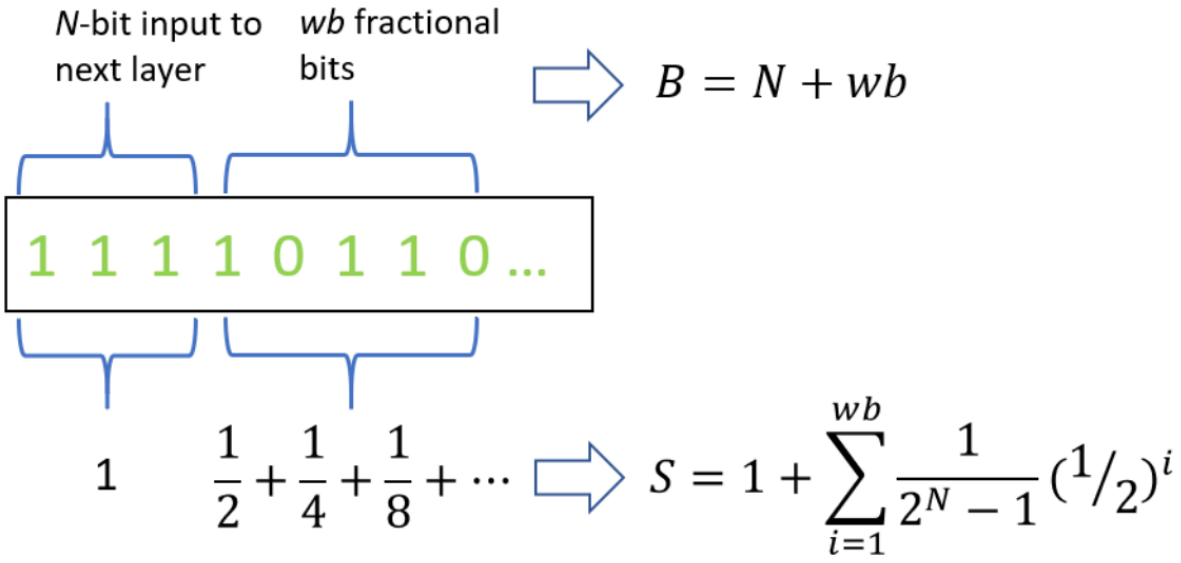


Figure 3.2: Visualization of bits corresponding to one element output of a bitserial operation and how to compute the corresponding Fixed Point Quantization (Equation 9) parameters.

than a division or multiplication, we can not simply add more shift bits. Instead, we must quantize the mean of activation tensor A in an equivalent integer format so that it can be subtracted from the quantized activations. To this end, we use fixed point quantization (FPQ) as defined in Algorithm 9. The number of relevant bits in the output of an N-bit bitserial layer is $N+wb$, where the top N bits form the quantized input to the next layer and the remaining wb bits are effectively fractional values. Thus we set $B = N+wb$ in Algorithm

Algorithm 9: Fixed point quantization (FPQ) function.

Input: a tensor X to quantize to B bits with scale S

Output: B-bit quantized tensor Y

1 $\hat{X} = \text{clip}(X, -S, S)$

2 $g = \frac{S}{2^B - 1}$ `// Compute granularity`

3 $Y = \text{round}(\frac{\hat{X}}{g})$

9. Next we must determine the proper range, or scale, term to use in the quantization. This value should be equal to the floating point value that setting all $N + wb$ bits represents. By linear quantization's construction, setting the top N bits represents a value of 1 and the least significant of those N bits represents the value $\frac{1}{2^N-1}$. The value of setting all remaining wb bits is the geometric sum $\sum_{i=1}^{wb} \frac{1}{2^N-1}(\frac{1}{2})^i$ which simplifies to $\frac{1}{2^N-1}(1 - \frac{1}{2^{wb}})$. Thus, setting all $N + wb$ bits is equivalent to the floating point value $S = 1 + \frac{1}{2^N-1}(1 - \frac{1}{2^{wb}})$. The computation of S and B are visualized in Figure 3.2.

With S and B properly defined, we can compute a quantized mean $\hat{\mu}$ from a floating point mean μ as $\hat{\mu} = FPQ(\mu, B, S)$ and directly subtract the result from binary activations. Conveniently, $\hat{\mu}$ can be subtracted from $(1 \ll (wb - 1))$ to create a new centering constant. Equation 3.5 is thus the final form of Equation 3.4 and allows weight scaling, normalization, and centering in just two integer operations.

$$\begin{aligned} cb &= (1 \ll (wb - 1)) - \hat{\mu} \\ q(a) &= (a + cb) \gg (wb + sb) \end{aligned} \tag{3.5}$$

As in the case of variance, we again keep track of the running mean of activations during train time so that during inference cb is a constant.

We thus have fully derived the fused glue operation used in Algorithm 8. The only other point worth noting is that we use $\text{clip}(q_k, 0, 2^N - 1)$ as the activation for our network. This has a similar effect as a saturating ReLU that bunches large activations into the highest quantization bin.

By identifying and addressing issues in existing quantization and introducing new operations that remove interlayer inefficiencies, Riptide opens the door to deploying functional high speed networks. However, there remains the looming question of how to translate a binary model into efficient machine code and the end-to-end speedups that can be realized. We address both of these questions in Chapter 4.

Table 3.1: Accuracy related binarization work and our results

	Model	Name	1-bit	2-bit	3-bit	full precision
		ImageNet top-1 accuracy				
1	AlexNet	Xnor-Net [48]	44.2%	—	—	56.6%
2	AlexNet	BNN [12]	27.9%	—	—	—
3	AlexNet	DoReFaNet [64]	43.6%	49.8%	48.4%	55.9%
4	AlexNet	QNN [27]	43.3%	51.0%	—	56.6%
5	AlexNet	HWGQ [4]	—	52.7%	—	58.5%
6	VGGNet	HWGQ [4]	—	64.1%	—	69.8%
7	AlexNet	Riptide-unipolar (ours)	44.5%	52.5%	53.6%	56.5%
8	AlexNet	Riptide-bipolar (ours)	42.8%	50.4%	52.4%	56.5%
9	VGGNet	Riptide-unipolar (ours)	56.8%	64.2%	67.1%	72.7%
10	VGGNet	Riptide-bipolar (ours)	54.4%	61.5%	65.2%	72.7%

3.4 Riptide Accuracy

Although Riptide offers clear benefits in terms of reducing the amount of computation in a full binary model, it is important that we confirm that our approximations do not cause unacceptable accuracy loss relative to other state-of-the-art binarization techniques. The vast bulk of previous binarization work is evaluated using AlexNet [35] and VGGNet [23] with the understanding that good performance on these models is likely to transfer to more modern architectures. To directly compare against these results, we train both models with a variety of bitwidths for both polarity configurations. In these comparisons, we consider HWGQ [4] the current state-of-the-art for high accuracy binary models. For each model, we binarize all layers except the input layer as is common practice in the literature. It is worth noting that unlike recent work [60, 64], however, we find that binarizing the output dense layer does not negatively impact the accuracy of Riptide models.

The results of our training experiments and the accuracy reported by previous binarization works are reported in Table 3.1. Models are binarized using all Riptide optimizations and trained on the ImageNet dataset for 100 epochs using SGD with an initial learning rate of 0.1 that is decreased by 10× every 30 epochs. We train variants of AlexNet and VGGNet with 1-bit, 2-bit, and 3-bit activations, in all cases weights are quantized bipolarly to 1-bit. For baselines we train full precision versions of Alexnet and VGGNet using the same settings as above. Across all models and bitwidth configurations, we find that Riptide matches accuracies of other approaches. Thus, we are confident that the benefits of Riptide come without an accuracy cost.

Chapter 4

GENERATING FAST CODE AND APPLYING IT

Having identified shortcomings of existing binarization techniques and introduced algorithms that allow the creation of an end-to-end binary network, we now turn our attention to converting binary networks to high quality code that can be run on commodity hardware. It may at first seem like generating code should be straight forward, however a naive implementation of a neural layer can be up to a 100× slower than one that is carefully implemented. A *schedule* defines how memory is utilized during computation and how instructions are executed on the target hardware. In the days of yore, scheduling required world experts to painstakingly hand craft each algorithm, that are then made available in linear algebra libraries like OpenBLAS [62]. Because binarization uses a fundamentally different core operation than typical networks, we are unable to reuse these existing libraries. Although binarization in theory provides large speedups, improperly scheduled binary layers are often much slower than full precision layers using BLAS libraries. Creating a highly optimized binary BLAS library by hand would be a colossal engineering effort that is difficult to justify for a researcher.

The difficulty of creating high quality binary kernels is a significant factor into why its so rare in the literature to actually evaluate the runtime of binary models. It would even likely be too much for us to take on if not for the recent emergence of tools that make scheduling much more approachable. Halide [46] introduced the concept of separation of compute definition and schedule. Halide removes the need to completely rewrite the algorithm for each variant of a schedule, which dramatically simplifies the effort required to explore and refine performance. However, Halide was intended for traditional image processing algorithms which are similar enough to neural network type operations to allow

them to be implemented but different enough to make it challenging. TVM [6] was built using many of the lessons of Halide as a compiler specifically for machine learning. Not only does TVM provide an excellent ecosystem of tools for creating new neural network algorithms and schedules, it also makes deploying to commodity hardware like the Raspberry Pi straight forward. In this chapter, we discuss how we build off of and contribute to TVM to realize end-to-end speedups with Riptide.

We perform all measurements in this chapter on a Raspberry Pi 3b [45]. The Raspberry Pi uses an ARM Cortex A53 processor which is the same architecture used in many other internet-of-things class devices such as Microsoft's Azure Sphere [56]. This leads us to believe that the Raspberry Pi is a suitable representative of mobile class platforms in general and reflects the environment that binary networks are best suited to. Although some of the optimizations presented are specific the to Pi, the majority can be transferred seamlessly to other hardware backends.

4.1 Compilation and Scheduling

To compile our described algorithms to efficient machine code, we extend TVM [6] to support bitserial operations including convolution and matrix multiplication. This allows Riptide to directly convert its TensorFlow training graph to a TVM based representation that can leverage LLVM to compile to multiple backends such as ARM CPUs, GPUs, and even smartphones. Additionally, supporting bitserial operations in TVM allows Riptide to apply TVM's scheduling primitives to bitserial operations. These scheduling primitives include:

- **Tiling**, which splits loops over a tensor into small regions that can better fit into the cache, thereby reducing memory traffic and increasing compute intensity.

- **Loop unrolling** which replicates the body of loops to reduce the overhead of maintaining induction variables and checking exit conditions.

- **Vectorization**, which enables the use of hardware SIMD instructions to operate on

LLVM 8.0

```
    | vmovl.8     q0, d0
    | vmovl.8     q2, d1
x8  | vand        q0, q0, q2
    | vcnt.8      q0, q0
    | vpaddl.8    q0, q0
    | vadd.16     q1, q0, q0

    vst1.16      q1, addr
```

Total: 49

Synthesized

```
x8  | vand.8      d0, d0, d1
    | vcnt.8      d0, d0

    vadd.8      d0, d0, d1
    vadd.8      d0, d0, d1
    vadd.8      d0, d0, d1
    vadd.8      d0, d0, d1

    vpadd.8     d0, d0, d1
    vpadd.8     d0, d0, d1

    vpadal.8    q1, {d0,d1}

    vst1.16      q1, addr
```

Total: 24

Figure 4.1: Microkernels for 8 × 8 by 8 × 1 matrix multiply with 1-bit values generated by (a) TVM's default LLVM output and (b) our fast popcount synthesis override. The fast popcount is half the length ("8×" code is unrolled 8 times) and twice as fast.

multiple tensor elements simultaneously.

- **Parallelization**, Which takes advantage of hardware MIMD facilities. For low power use cases such as on the Raspberry Pi this enables multiple cores on a multiprocessor to be used.

We also found that LLVM had difficultly compiling popcount operations for ARM class CPUs. Although it produced correct outputs, it was eagerly promoting values to 16 bit datatypes and was not vectorizing popcounts in the inner loop. To remedy this, we overrode the default LLVM mapping of popcount using TVM tensorization. A sample of our generated code versus the TVM default is shown in Figure 4.1. Here we find that our so called "fast popcount" operation requires half as many operations as the default.

It's important to note that each of these scheduling primitives (except perhaps loop

unrolling) is highly dependant on the hardware of the target platform. For example, it is important to have a detailed understanding of cache and memory parameters to determine the proper tile size. Similarly, choosing the proper amount of vectorization and parallelization requires a detailed understanding of the processors capabilities. Although the Raspberry Pi is a decent surrogate for other low power platforms, the optimal scheduling parameters is highly likely to vary between each platform. This would make it difficult to port Riptide to a new platform. Fortunately, TVM recently added support for autotuning [7], which uses gradient boosted decision trees to explore a schedule space and automatically find a near optimal set of schedule parameters. We incorporate AutoTVM into Riptide to make the process of applying Riptide to a new environment straight forward.

4.2 Bitpack Fusion

When discussing the amount of memory used in a deep neural network, it is common practice to simply count the number of parameters and multiply by the number of bytes per parameter (four for full precision for example). However, this assumes that the size of parameters is dramatically larger than intermediate activations, which of course use memory as well. In binary networks, the parameters require only one or two bits, significantly reducing the amount of memory they require. This causes the intermediate activations to actually take a sizeable amount of the total memory in a network. For embedded platforms that have only a few hundred kilobytes of storage, reducing intermediate memory usage is essential. It may seem that binary networks immediately provide this activation activation, however it is important to remember that the output of popcount is actually int16 datatype tensors. The output of popcount is then repacked into much more efficient binary tensors, but the full int16 tensor still is temporarily stored in memory, contributing to the peak memory usage of the model.

To address this, we introduce bitpack fusion, visualized in Figure 4.2. By computing the output of a binary layer in small chunks, we can fold packing into the inner loop and reduce intermediate memory used from $2NHWC$ bytes to $\frac{NHWC}{8}$ bytes, where N is the number of

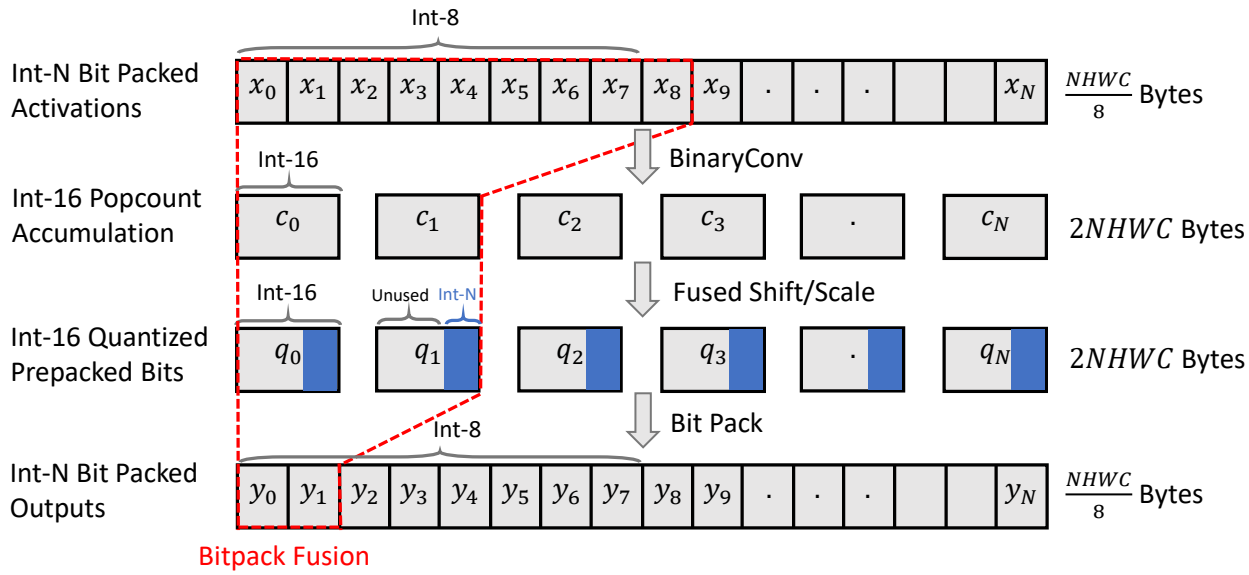


Figure 4.2: Scheduling of N-bit binary layer demonstrating intermediate memory used. By fusing computation within tiles, such as the region highlighted red, memory use can be reduced.

bits. Not only is this memory reduction useful in resource constrained environments, it also can improve runtime by reducing memory traffic.

4.3 Runtime Evaluation

In this section we analyze the speedups offered by Riptide and study the impact of the many optimizations introduced in this dissertation. Although previous chapters have focused primarily on AlexNet and VGGNet, in large part due to the need to compare to previous work, we note that they are bulky and outdated models that would not be run on commodity hardware in any real world application. We instead consider the optimization of SqueezeNet [28], an incredibly parameter efficient network that was designed for low power applications.

Table 4.1: Runtime Measurements Yielded by Riptideon Popular Architectures

	Model	Name	1-bit	2-bit	3-bit	full precision
			ImageNet top-1 accuracy / Runtime (ms)			
1	AlexNet	Xnor-Net [48]	44.2% / —	— / —	— / —	56.6% / —
2	AlexNet	BNN [12]	27.9% / —	— / —	— / —	— / —
3	AlexNet	DoReFaNet [64]	43.6% / —	49.8% / —	48.4% / —	55.9% / —
4	AlexNet	QNN [27]	43.3% / —	51.0% / —	— / —	56.6% / —
5	AlexNet	HWGQ [4]	— / —	52.7% / —	— / —	58.5% / —
6	VGGNet	HWGQ [4]	— / —	64.1% / —	— / —	69.8% / —
7	AlexNet	Riptide-unipolar (ours)	44.5% / 150.4	52.5% / 196.8	53.6% / 282.8	56.5% / 1260.0
8	AlexNet	Riptide-bipolar (ours)	42.8% / 122.7	50.4% / 154.6	52.4% / 207.0	56.5% / 1260.0
9	VGGNet	Riptide-unipolar (ours)	56.8% / 243.8	64.2% / 387.2	67.1% / 610.0	72.7% / 2420.0
10	VGGNet	Riptide-bipolar (ours)	54.4% / 184.1	61.5% / 271.4	65.2% / 423.5	72.7% / 2420.0
11	ResNet18	Riptide-unipolar (ours)	47.9% / 76.2	58.4% / 112.0	61.8% / 152.3	70.9% / 380.8

4.3.1 End-to-End Results

In Table 4.1, we extend the accuracy measurements presented in Table 3.1 with runtime measurements at various bitwidths and quantization polarities. The runtime of baseline models is measured using optimized TVM implementations. The speedup results for each model and bitwidth is visualized in Figure 4.3.

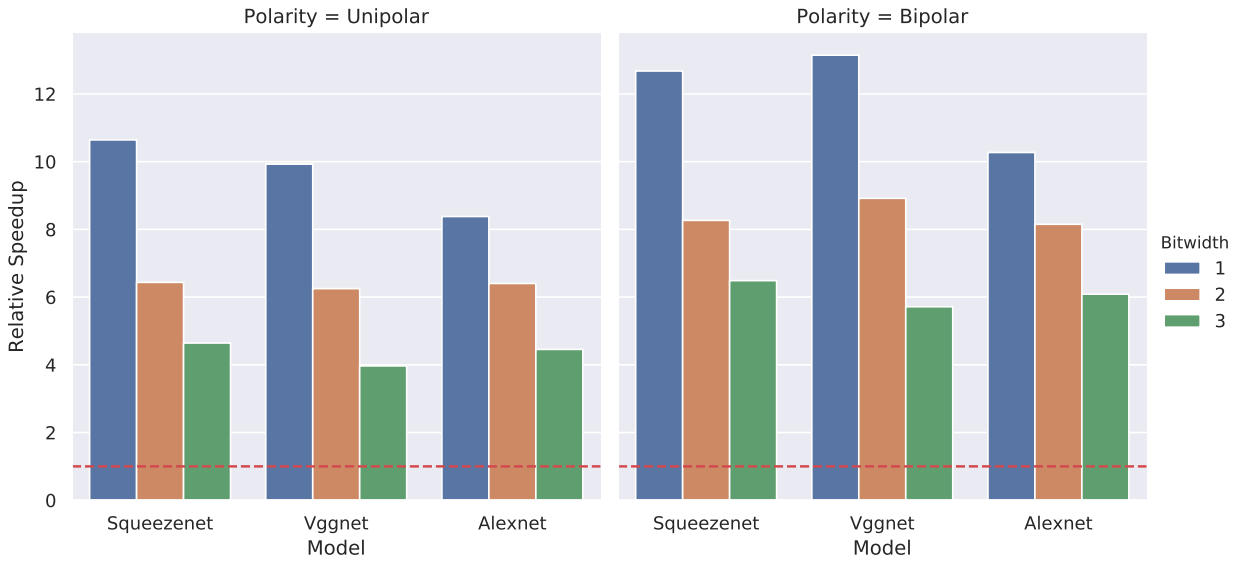


Figure 4.3: Visualization of end to end speedups of all explored models and bitwidths compared to floating point baselines.

There are three key takeaways from these results:

- Riptide is able to generate the first ever reported end-to-end speedups of a binary model and achieves accuracy comparable to the state-of-the-art across all configurations, confirming that Riptide's optimizations do not cause a drop in accuracy.

- We observe end-to-end speedups across all models and bitwidths and have a wide range of points in terms of the speed to accuracy trade-off; ranging from high accuracy 3-bit unipolar models with 4× speedup to high speed bipolar models with 12× speedup.

- Although unipolar models yield higher accuracy and slower runtimes than bipolar models as expected, they are only about 25% slower despite having twice the number of operations. This suggests our mixed polarity implementation (Equation 3.1) is quite efficient.

Taking these points together, we are confident that Riptide provides a high quality implementation of bitserial networks.

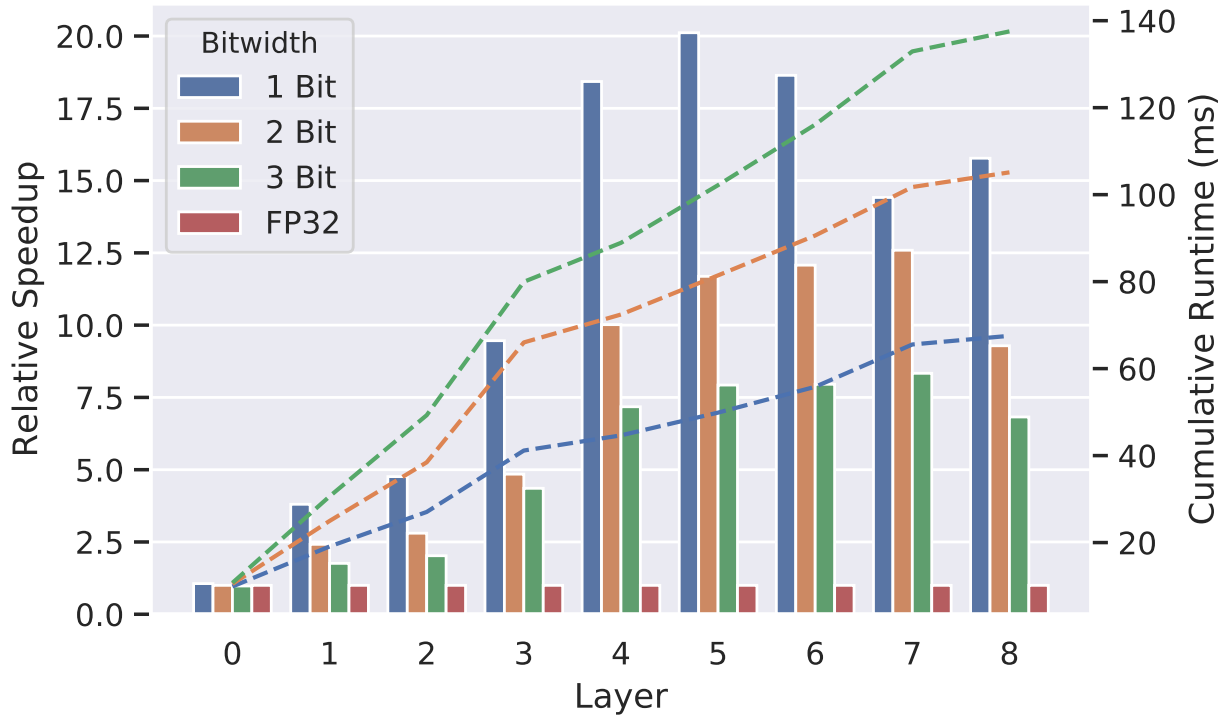


Figure 4.4: Layerwise speedup of SqueezeNet quantized with varying bitwidth versus the floating point baseline model.

4.3.2 Layerwise Analysis

We measure the per-layer runtime of SqueezeNet unipolarly quantized at each bitwidth and visualize the results in Figure 4.4. The bars indicate the relative speedup versus a floating

point baseline and the dotted lines tracks the cumulative runtime over the layers. There are a few interesting takeaways from this measurement. We see that not all layers benefit equally from quantization; those towards the end of the model have speedup of up to 20× compared to early layers' 3×. Early layers tend to be spatially larger but have fewer channels than later layers, suggesting that binarization scales better with the number of channel than it does spatially. Leveraging this knowledge, it may be possible to design architectures that are able to achieve higher speedups when binarized even if they are less efficient in full precision. We also note that the output dense layer achieves speedups inline with convolutional layers, suggesting our techniques apply well to both types of layer. Although we leave the input layer in full precision, we see that it takes a relatively small amount of the total runtime, suggesting that this is not a significant bottleneck.

4.3.3 Optimization Ablation

We perform a one-off ablation study of each of Riptide's optimizations. The results of this study are shown in Figure 4.5. Although not all optimizations contribute equally, its clear that they are all essential to our final highly performant model, with the smallest reduction lowering the end-to-end speedup from 10.6× to 8.4× and the largest reduction lowering speedups to only 2.9×. In the subsequent sections we drill down further into these optimizations to better understand their impact.

4.3.4 Polarity

To better examine the impact of polarity, we consider the runtime of the first layer of SqueezeNet for a baseline FP32 model, a unipolar model, and a bipolar model with the latter two being quantized unipolarly with 1-bit in Figure 4.7. Here, we have added a yellow line to indicate the runtime if the layer were entirely compute bound (calculated by dividing the number of operations by the RPi's frequency × core count). We see that the baseline model is extremely close to its hypothetical ceiling, suggesting it is in fact quite compute bound. The binarized models on the other hand are far from compute bound. Because

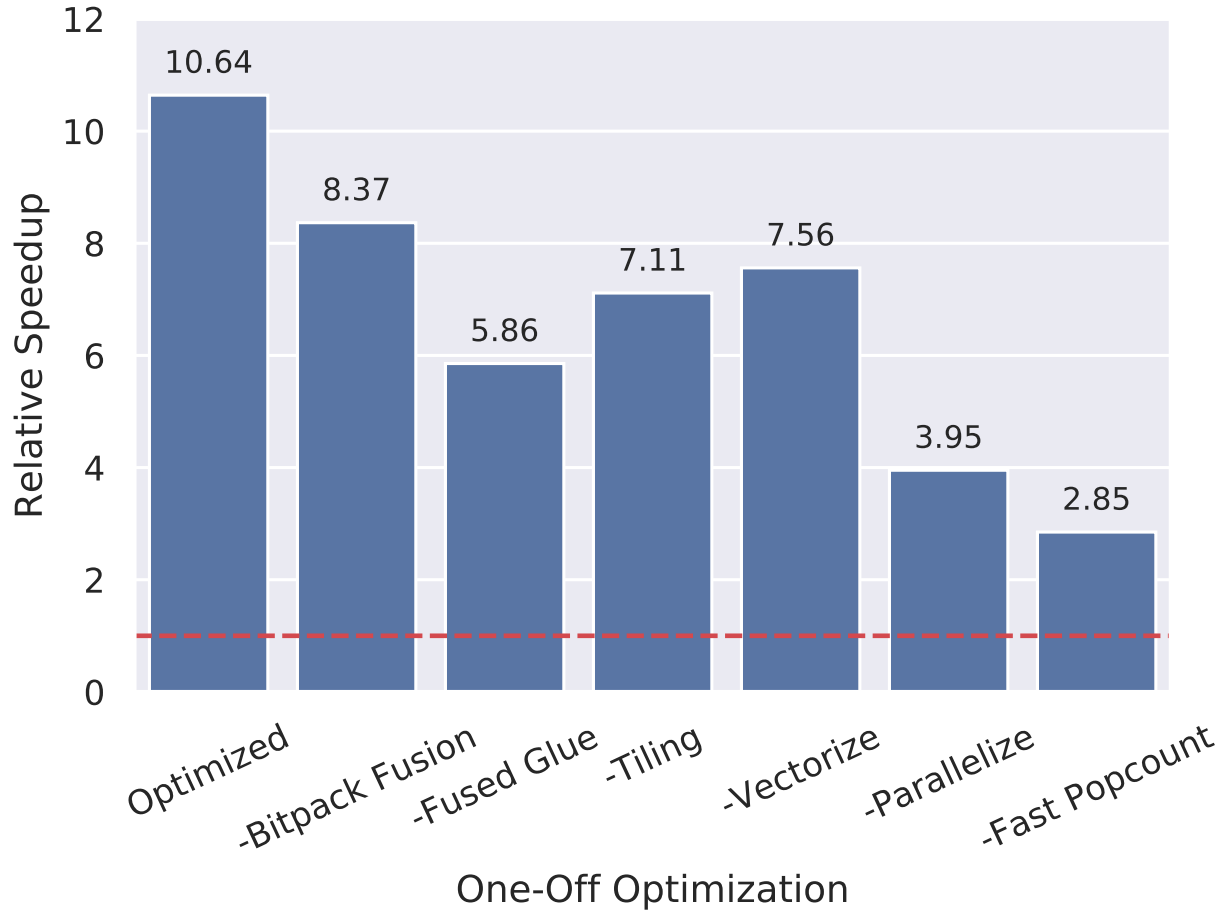


Figure 4.5: Ablation study of the effect of Riptide optimizations versus the baseline floating point model.

Riptide's unipolar quantization requires no more memory operations than bipolar, it is only marginally slower. Thus in general, unipolar quantization tends to give better accuracy to speedup trade-offs. However, in situations where speed is more important than accuracy, bipolar models may be more attractive.

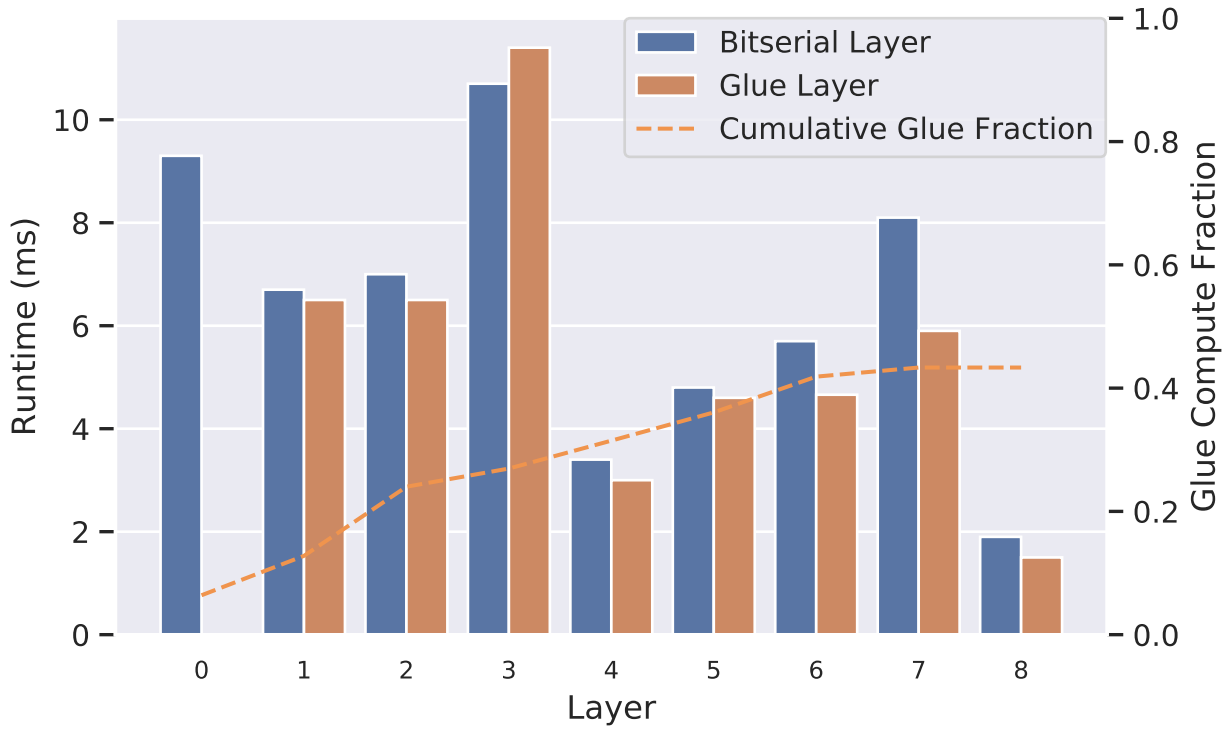


Figure 4.6: Layerwise runtime breakdown of SqueezeNet quantized with 1-bit weights and activations.

4.3.5 Scheduling

To demonstrate the impact of proper machine code generation, we take a SqueezeNet binarized unipolarly with 1-bit and compile it with no optimizations then measure its end-to-end runtime. From that unoptimized starting point, we incrementally apply the optimizations discussed in Section 4.1 and measure the resulting runtimes. The speedups of these measurements are shown in Figure 4.8. We note that an unoptimized binary model is actually about 7× **slower** than the full precision baseline. We can see that each optimization contributes significantly to generating performant machine code. Notably, we find that bitpack fusion gives a speed boost of about 30%, all of which comes from the reduction to memory operations that it contributes.

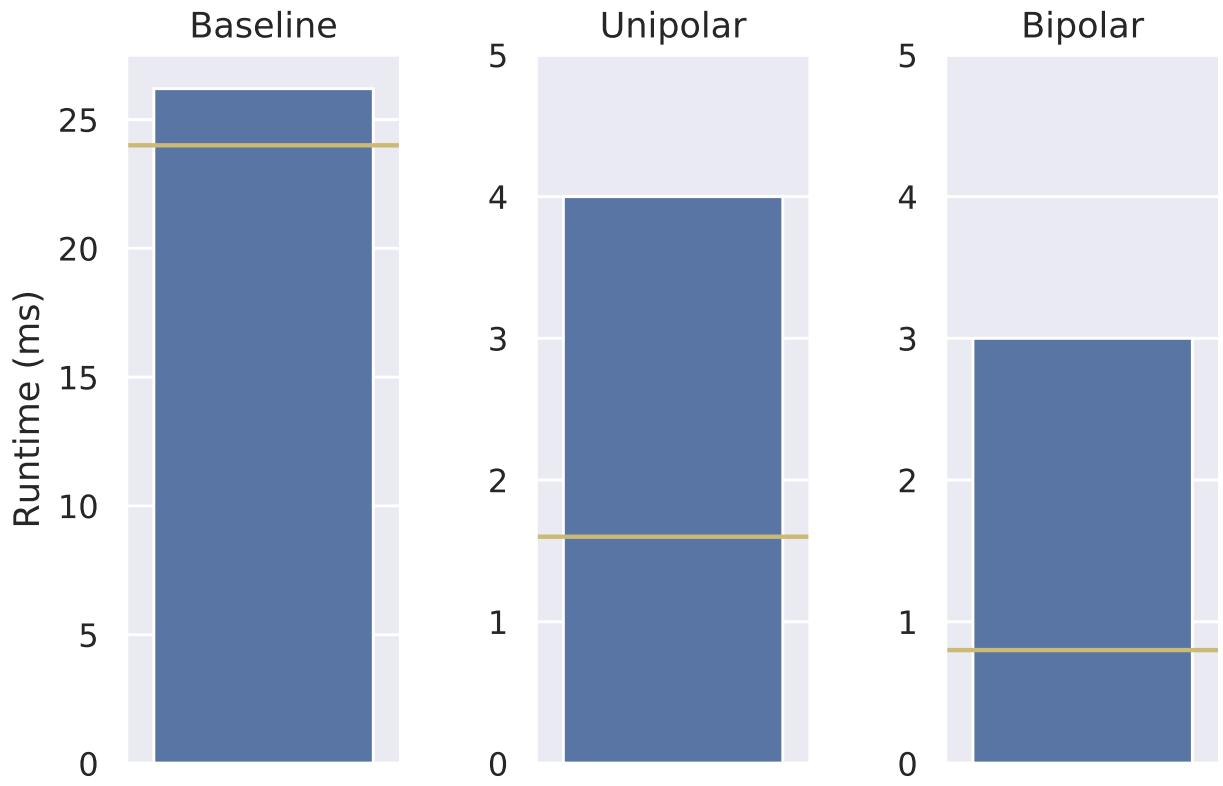


Figure 4.7: Effect of quantization polarity on the runtime of the first fire layer in SqueezeNet. The horizontal yellow line indicates the runtime of the layer if it were run with perfect efficiency.

4.3.6 Glue Layers

To understand the impact of our fused glue operation, we examine the runtime of SqueezeNet when all optimizations except fused glue are applied. We first measure the runtime of each bitserial layer and its corresponding glue and visualize the results in Figure 4.6. We find that glue layers make up a considerable portion of each layer and cumulatively contribute 44% of the total inference time. This confirms that glue layers are a major bottleneck at all points in the model. Next, we examine each individual operation in the first quantized layer of SqueezeNet and visualize the runtimes in Figure 4.9. Here we find that although the Quantize operation is the largest of the glue operations, all contribute non-trivially. This

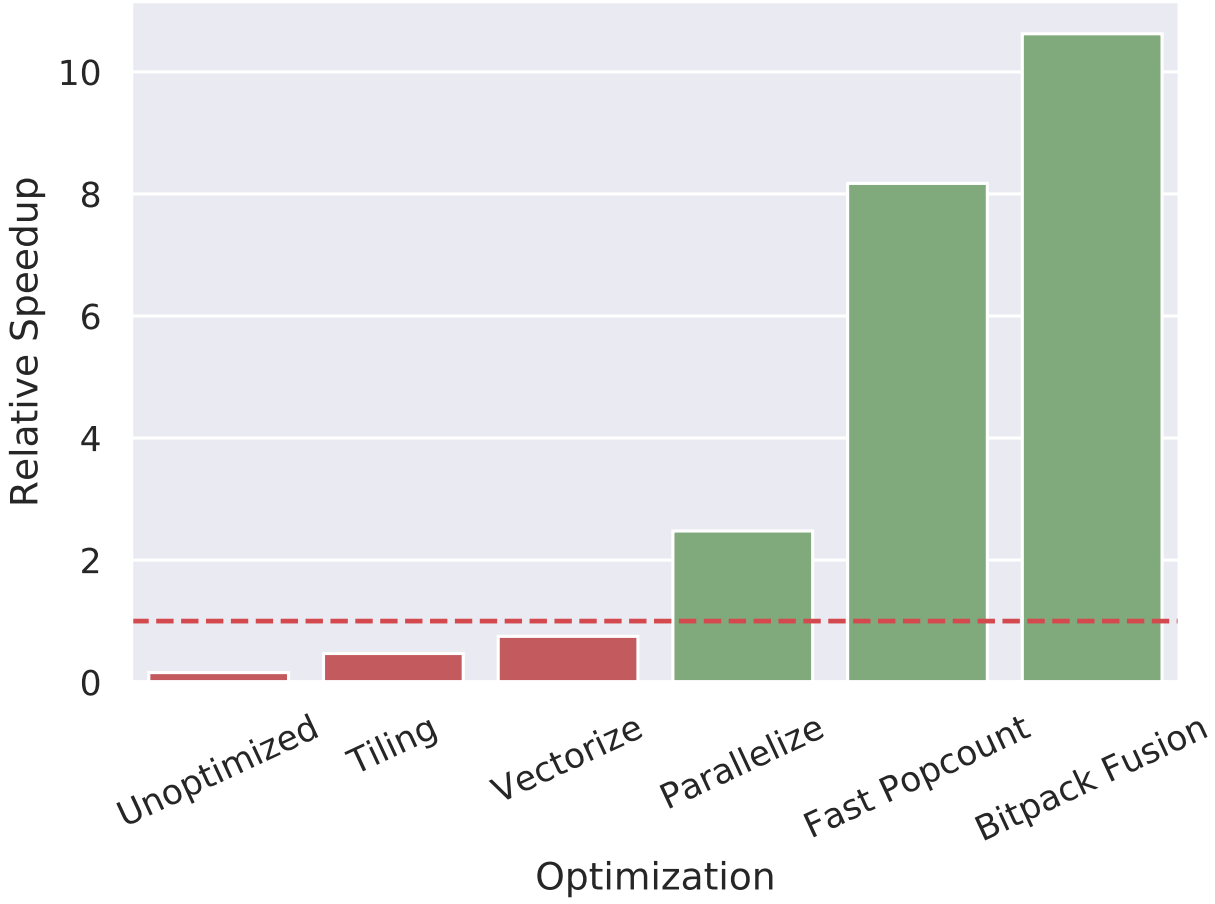


Figure 4.8: End-to-end speedup of quantized SqueezeNet versus the baseline floating point model when scheduling optimizations are incrementally applied.

leads us to conclude that optimizing only a portion of the glue layers would be insufficient and give us confidence that our fused glue operation is essential and highly performant.

4.4 Pareto Optimality

In Section 4.3 we demonstrated that Riptide is capable of producing networks that offer significant speedups with respect to full precision baselines. However, because these speedups come at an accuracy cost, it is not obvious that the networks produced by Riptide are Pareto optimal. In this case, a Pareto optimal model is one that offers the highest possible accuracy

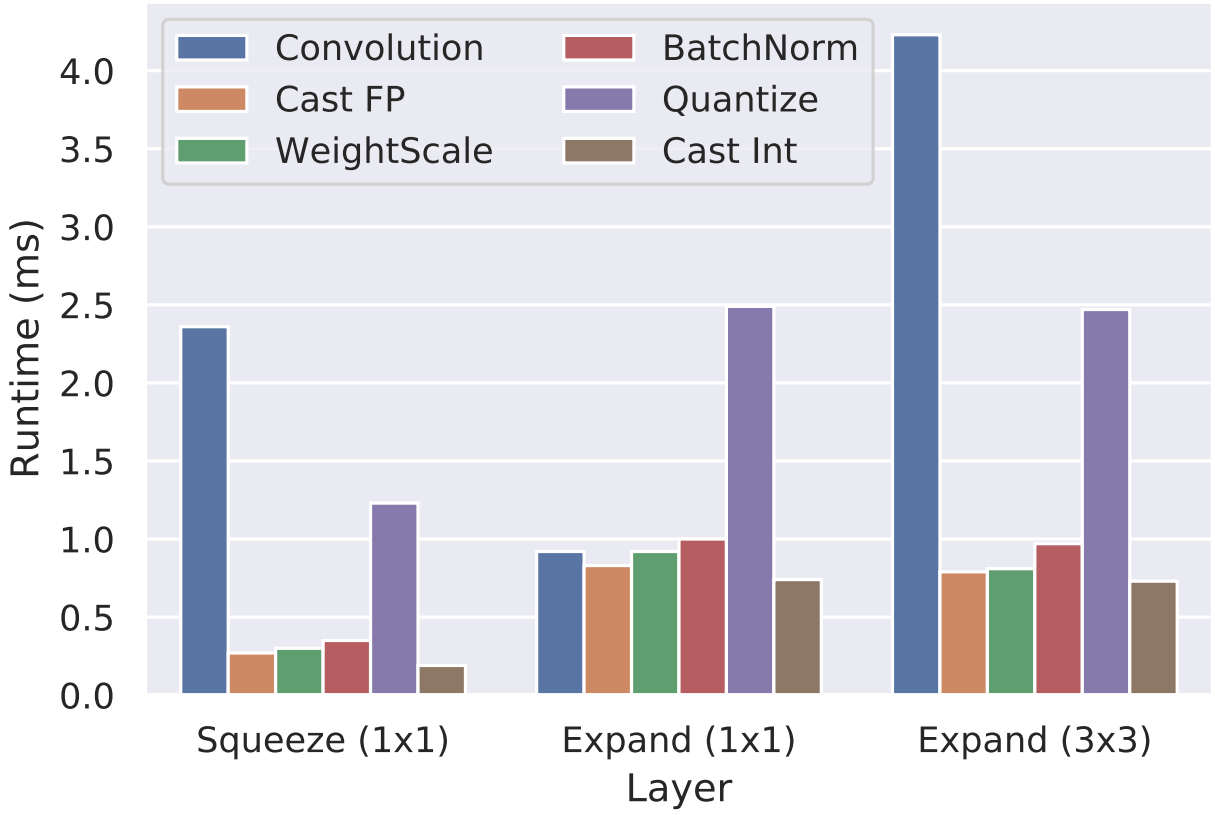


Figure 4.9: Runtime of each operation in the first fire layer of SqueezeNet quantized with 1-bit weights and activations.

at a certain runtime cost. For Riptide to produce Pareto optimal models, binarization must enable better speed to accuracy tradeoffs than more efficient models and other types of optimization. Here, we consider a fast variant of the most popular architecture in computer vision: ResNet18 [24].

A very simple but effective method for reducing a model's runtime is the reduction of the input images dimensions. Because convolutions are spatially invariant, the same parameters can be applied to arbitrary shaped inputs. By reducing the number of pixels that need to be convolved, input dimension reduction directly reduces the amount of compute in each layer. For example, cutting the resolution in half reduces the amount of compute by a factor of

four. However, just as in other approximating optimizations, reducing the input resolution causes an accuracy loss as useful details can be obscured.

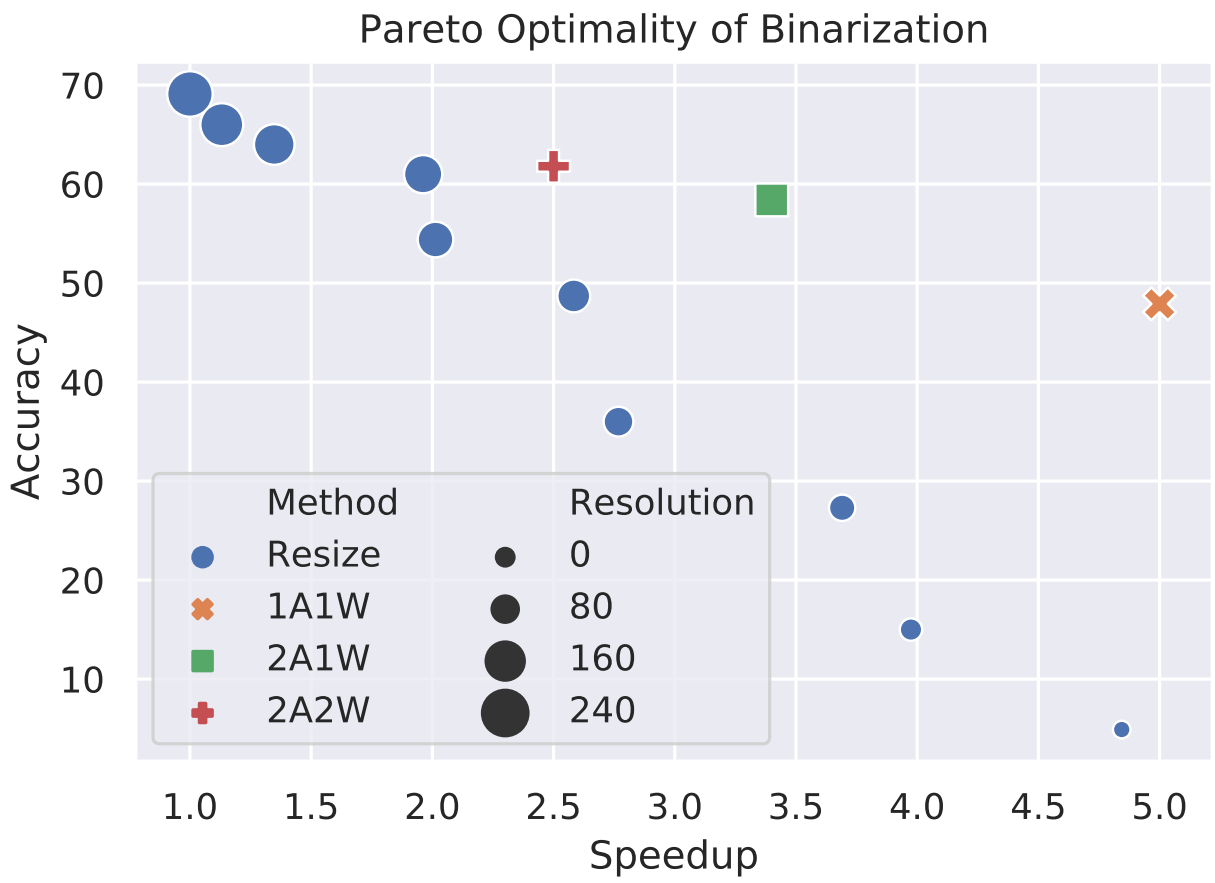


Figure 4.10: Comparison of ResNet18 speed-accuracy tradeoff points offered by reducing input resolution versus those offered by binarization at varying bitwidths. We see that binarization offers points well beyond the Pareto boundary.

To compare Riptide's binarization with input scaling we sweep the runtime and accuracy of ResNet18 with various input resolutions on a Raspberry Pi and compare the resulting speed-accuracy points to those offered by Riptide. The results are shown in Figure 4.10. We see that resolution scaling offers a very clear nearly linear Pareto boundary that binarization methods outperform across the board. This is strong evidence that binarization is a

compelling optimization that may find a home in many real world applications.

Chapter 5

HETEROGENEOUS BITWIDTH NEURAL NETWORKS

It has not escaped hardware designers that the popcount-xnor operations used in a binary network are especially well suited for FPGAs or ASICs. Taking the xnor of two bits requires a single logic gate compared to the hundreds required for even the most efficient floating point multiplication units [18]. The drastically reduced area requirements allows binary networks to be implemented with fully parallel computations on even relatively inexpensive FPGAs [61]. The level of parallelization afforded by these custom implementations allows them to outperform GPU computation while expending a fraction of the power, which offers a promising avenue of moving state of the art architectures to embedded environments. We seek to improve the occupancy, power, and/or accuracy of these solutions.

Our approach is based on the simple observation that the power consumption, space needed, and accuracy of binary models on FPGAs and custom hardware are proportional to mn, where m is the number of bits used to binarize input activations and n is the number of bits used to binarize weights. Current binary algorithms restrict m and n to be integer values, in large part because efficient CPU implementations require parameters within a layer to be the same bitwidth. However, hardware has no such requirements. Thus, we ask whether bitwidths can be fractional. To address this question, we introduce Heterogeneous Bitwidth Neural Networks (HBNNs), which allow *each individual parameter to have its own bitwidth*, giving a fractional average bitwidth to the model.

In this chapter we:

(1) Propose the problem of selecting the bitwidth of individual parameters during training such that the bitwidths average out to a specified value.

(2) Show how to augment a state-of-the-art homogeneous binarization training scheme with a greedy bitwidth selection technique (which we call "middle-out") and a simple hyperparameter search to produce good heterogeneous binarizations efficiently.

(3) Present a rigorous empirical evaluation (including on highly optimized modern networks such as Google's MobileNet) to show that heterogeneity yields equivalent accuracy at significantly lower average bitwidth.

(4) Provide estimates based on recently proposed FPGA/ASIC implementations that HBNNs' lower average bitwidths can translate to significant reductions in circuit area and power.

5.1 Residual Error Quantization

In previous chapters we've focused primarily on linear quantization as in Equation 2.3. However, there are other methods of quantization that trade simplicity for greater representational power. In this chapter, we extend the *residual error binarization* introduced by Tang et al. [60]. Residual error quantization is defined in Equation 5.1.

$$\begin{gathered} T_1^B = \text{sign}(T),\ \mu_1 = \text{mean}(|T|) \\ E_n = T - \sum_{i=1}^{n} \mu_i \times T_i^B \\ T_{n>1}^B = \text{sign}(E_{n-1}),\ \mu_{n>1} = \text{mean}(|E_{n-1}|) \\ T \approx \sum_{i=1}^{n} \mu_i \times T_i^B \end{gathered} \tag{5.1}$$

where T is the input tensor, E_n is the residual error up to bit n, T_n^B is a tensor representing the n^{th} bit of the approximation, and μ_n is a scaling factor for the n^{th} bit. Note that the calculation of bit n is a recursive operation that relies on the values of all bits less than n. Residual error binarization has each additional bit take a step from the value of the previous bit. Figure 5.1 illustrates the process of binarizing a single value to 3 bits. Since every

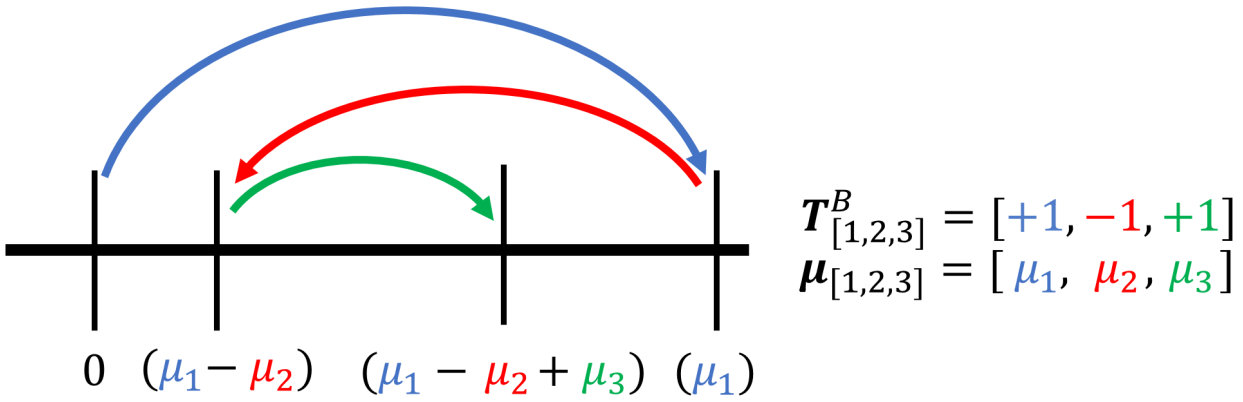


Figure 5.1: Residual error binarization with $n = 3$ bits. Computing each bit takes a step from the position of the previous bit (see Equation 5.1).

binarized value is derived by taking n steps, where each step goes left or right, residual error binarization approximates inputs using one of 2^n values.

5.2 Heterogeneous Binarization

To date, there remains a considerable gap between the performance of 1-bit and 2-bit networks (compare rows 8 and 10 of Table 5.1). The highest full (i.e., where both weights and activations are quantized) single-bit performer on AlexNet, Xnor-Net, remains roughly 7 percentage points less accurate (top 1) than the 2-bit variant, which is itself about 5.5 points less accurate than the 32-bit variant (row 25). When only weights are binarized, very recent results [17] similarly find that binarizing to 2 bits can yield nearly full accuracy (row 2), while the 1-bit equivalent lags by 4 points (row 1). The flip side to using 2 bits for binarization is that the resulting models require double the number of operations as the 1-bit variants at inference time.

These observations naturally lead to the question, explored in this section, of whether it is possible to attain accuracies closer to those of 2-bit models while running at speeds

closer to those of 1-bit variants. Of course, it is also fundamentally interesting to understand whether it is possible to match the accuracy of higher bitwidth models with those that have lower (on average) bitwidth. Below, we discuss how to extend residual error binarization to allow heterogeneous (effectively fractional) bitwidths and present a method for distributing the bits of a heterogeneous approximation.

5.2.1 Heterogeneous Residual Error Binarization via a Mask Tensor

We modify Equation 5.1, which binarizes to n bits, to instead binarize to a mixture of bitwidths by changing the third line as follows:

$$\begin{gathered} T^B_{n>1} = \text{sign}(E_{n-1,j}),\ \mu_{n>1} = \text{mean}(|E_{n-1,j}|) \\ \text{with } j : M_j \geq n \end{gathered} \tag{5.2}$$

Note that the only addition is the *mask tensor* M, which is the same shape as T, and specifies the number of bits M_j that the j^{th} entry of T should be binarized to. In each round n of the binarization recurrence, we now only consider values that are not finished binarizing, i.e, which have $M_j \geq n$. Unlike homogeneous binarization, therefore, heterogeneous binarization generates binarized values by taking *up to*, not necessarily exactly, n steps. Thus, the number of distinct values representable is $\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$, which is roughly double that of the homogeneous binarization.

In the homogeneous case, on average, each step improves the accuracy of the approximation, but there may be certain individual values that would benefit from not taking a step, in Figure 5.1 for example, it is possible that $(\mu_1 - \mu_2)$ approximates the target value better than $(\mu_1 - \mu_2 + \mu_3)$. If values that benefit from not taking a step can be targeted and assigned fewer bits, the overall approximation accuracy will improve despite there being a lower average bitwidth.

5.2.2 Computing the Mask Tensor M

The question of how to distribute bits in a heterogeneous binary tensor to achieve high representational power is equivalent to asking how M should be generated. When computing M, our goal is to take an average bitwidth B and determine both what fraction P of M should be binarized to each bitwidth (e.g., $P = 5\%$ 3-bit, 10% 2-bit and 85% 1-bit for an average of $B = 1.2$ bits), and how to distribute these bitwidths across the individual entries in M. The full computation of M is described in Algorithm 9.

We treat the distribution P over bitwidths as a model-wide hyperparameter. Since we only search up to 3 bitwidths in practice, we perform a simple grid sweep over the values of P. As we discuss in Section 5.3.3, our discretization is relatively insensitive to these hyperparameters, so a coarse sweep is adequate. The results of the sweep are represented by the function $DistFromAvg$ in Algorithm 9.

Given P, we need to determine how to distribute the various bitwidths using a value aware method: assigning low bitwidths to values that do not need additional approximation and high bitwidths to those that do. To this end, we propose several sorting heuristic methods: Top-Down (TD), Middle-Out (MO), Bottom-Up (BU), and Random (R). These methods all attempt to sort values of T based on how many bits that value should be binarized with. For example, Top-Down sorting assumes that larger values need fewer bits, and so performs a standard descending sort. Similarly, Middle-Out sorting distributes fewer bits to values closest to the mean of T, while Bottom-Up sorting assigns fewer bits to smaller values. As a simple we control, we also consider Random sorting, which assigns bits in a completely uninformed way. The definitions for the sorting heuristics is given by Equation 5.3.

$$\begin{aligned} \mathrm{TD}(T) &= \mathrm{sort}(|T|, \mathrm{descending}) \\ \mathrm{MO}(T) &= \mathrm{sort}(|T| - \mathrm{mean}(|T|), \mathrm{ascending}) \\ \mathrm{BU}(T) &= \mathrm{sort}(|T|, \mathrm{ascending}) \\ \mathrm{R}(T) &= \text{a fixed uniformly random permutation of } T \end{aligned} \tag{5.3}$$

To evaluate the methods in Equation 5.3, we performed two experiments. In the first,

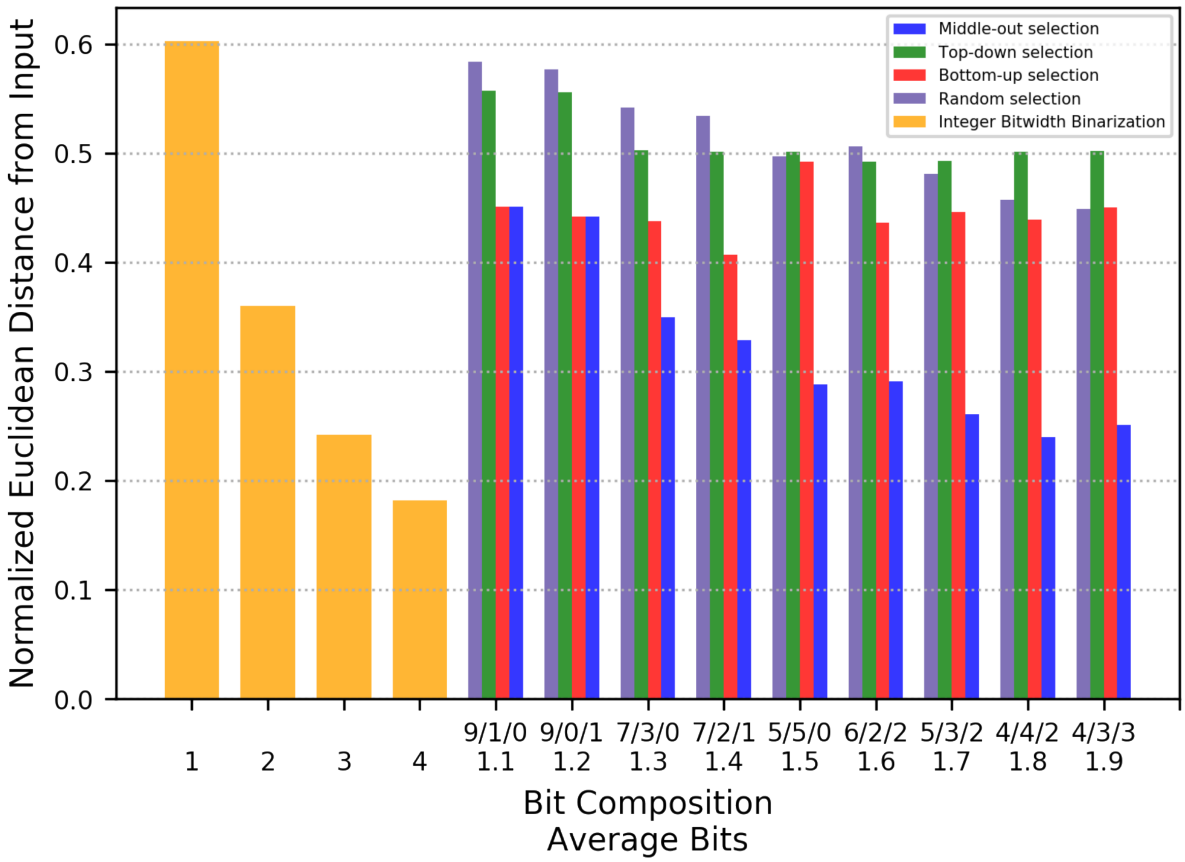


Figure 5.2: Ability of different binarization schemes to approximate a large tensor of normally distributed random values. A bit composition denoted as x/y/z indicates x% of values are binarized to 1-bit, y% to 2-bit, and z% to 3-bit.

we create a large tensor of normally distributed values and binarize it with a variety of bit distributions P and each of the sorting heuristics using Algorithm 9. We then computed the Euclidean distance between the binarized tensor and the original full precision tensor. A lower normalized distance suggests a more powerful sorting heuristic. The results of this experiment are shown in Figure 5.2, and show that Middle-Out sorting outperforms other heuristics by a significant margin. Notably, the results suggest that using Middle-Out sorting can produce approximations with fewer than 2-bits that are comparably accurate to 3-bit integer binarization.

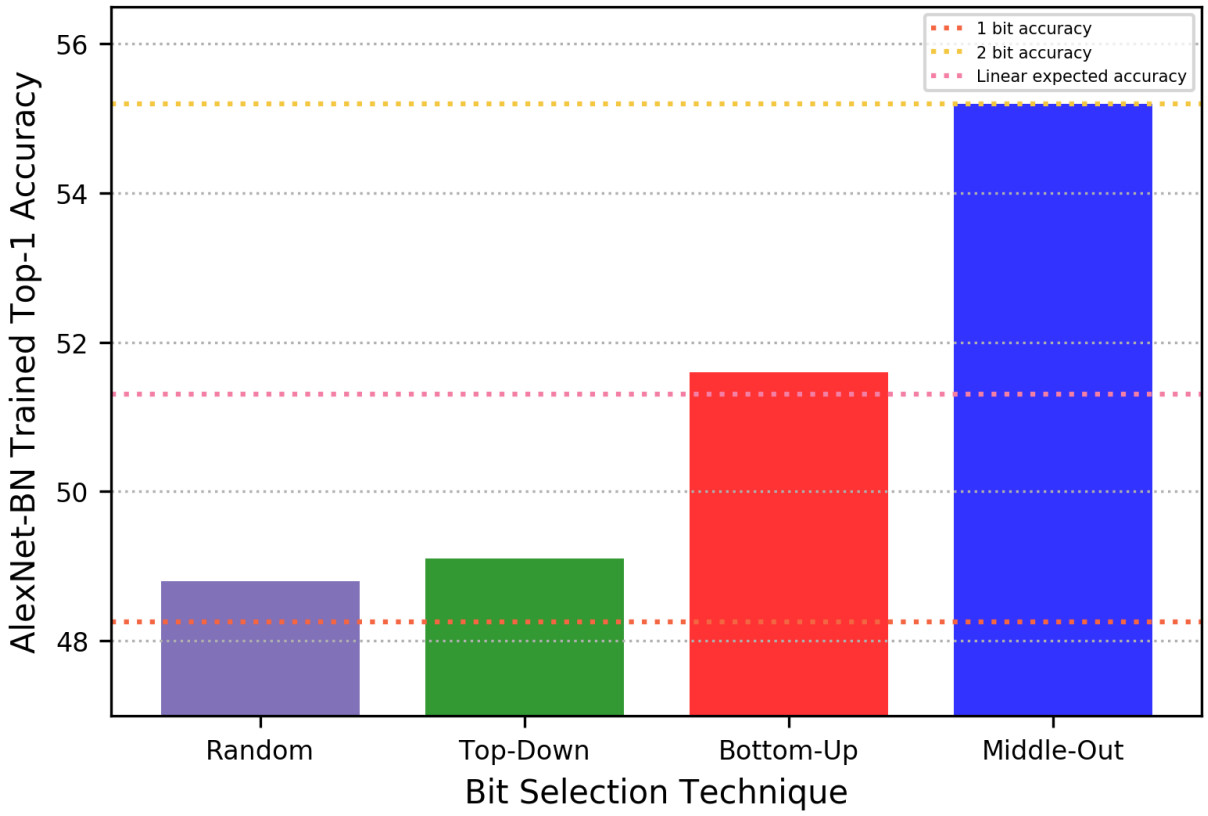


Figure 5.3: Effectiveness of heterogeneous bit selection techniques (a) ability of different binarization schemes to approximate a large tensor of normally distributed random values. (b) accuracy of 1.4 bit heterogeneous binarized AlexNet-BN trained using each bit-selection technique.

To confirm these results translate to accuracy in binarized convolutional networks, we consider 1.4 bit binarized AlexNet, with bit distribution P set to 70% 1-bit, 20% 2-bit, and 10% 3-bit, an average of 1.4 bits. The specifics of the model and training procedure are the same as those described in Section 5.3.1. We train this model with each of the sorting heuristics and compare the final accuracy to gauge the representational strength of each heuristic. The results are shown in Figure 5.3. As expected, Middle-Out sorting performs significantly better than other heuristics and yields an accuracy comparable to 2-bit integer

Algorithm 10: Generation of bit map M.

Input: A tensor T of size N and an average bitwidth B

Output: A bit map M that can be used in Equation 5.2 to heterogeneously binarize T

$R = T$ // Initialize R, which contains values that have not yet been assigned a bitwidth

$x = 0$

$P = \text{DistFromAvg}(B)$ // Generate distribution of bits to fit average

// b is a bitwidth and p_b is the percentage of T to binarize to width b

for *(b, p_b) in* P **do**

  $S = \text{SortHeuristic}(R)$ // Sort indices of remaining values by suitability for b-bit binarization

  $M[S[x : x + p_b N]] = b$

  $R = R \setminus R[S[x : x + p_b N]]$ // Do not consider these indices in next step

  $x \mathrel{+}= p_b N$

end

binarization despite using on average 1.4 bits.

The intuition behind the exceptional performance of Middle-Out is based on Figure 5.1 . We can see that the values that are most likely to be accurate without additional bits are those that are closest to the average μ_n for each step n. By assigning low bitwidths to the most average values, we can not just minimize losses, but in some cases provide a better approximation using fewer average steps. In proceeding sections, all training and evaluation is performed with Middle-Out as the sorting heuristic in Algorithm 9.

5.3 Experiments

To evaluate HBNNs we wished to answer the following three questions:

(1) How does accuracy scale with an uninformed bit distribution?

(2) How well do HBNNs perform on a challenging dataset compared to the state of the art?

(3) Can the benefits of HBNNs be transferred to other architectures?

In this section we address each of these questions.

5.3.1 Implementation Details

AlexNet with batch-normalization (AlexNet-BN) is the standard model used in binarization work due to its longevity and the general acceptance that improvements made to accuracy transfer well to more modern architectures. Batch normalization layers are applied to the output of each convolution block, but the model is otherwise identical to the original AlexNet model proposed by Krizhevsky et al. [35]. Besides it's benefits in improving convergence, Rastegary et al. [48] found that batch-normalization is especially important for binary networks because of the need to equally distribute values around zero. We additionally insert binarization functions within the convolutional layers of the network when binarizing weights and at the input of convolutional layers when binarizing inputs. We keep a floating point copy of the weights that is updated during back-propagation, and binarized during forward propagation as is standard for binary network training. The gradient for our binarization function is a slightly modified, "bitwidth-aware", straight-through estimator:

$$\frac{dT^b}{dT} = 1_{|T| \leq M}. \tag{5.4}$$

where M is the bitwidth map used in Equation 5.2. This allows gradient relaxation for values with bitwidths above 1, since they are capable of more complex expression. For homogeneous bitwidth tensors, M is replaced by the integer bitwidth.

When binarizing the weights of the network's output layer, we add a single parameter scaling layer that helps reduce the numerically large outputs of a binary layer to a size more amenable to softmax, as suggested by Tang et al. [60]. We train all models using an SGD solver with learning rate 0.01, momentum 0.9, and weight decay 1e-4 and randomly initialized weights for 90 epochs on PyTorch.

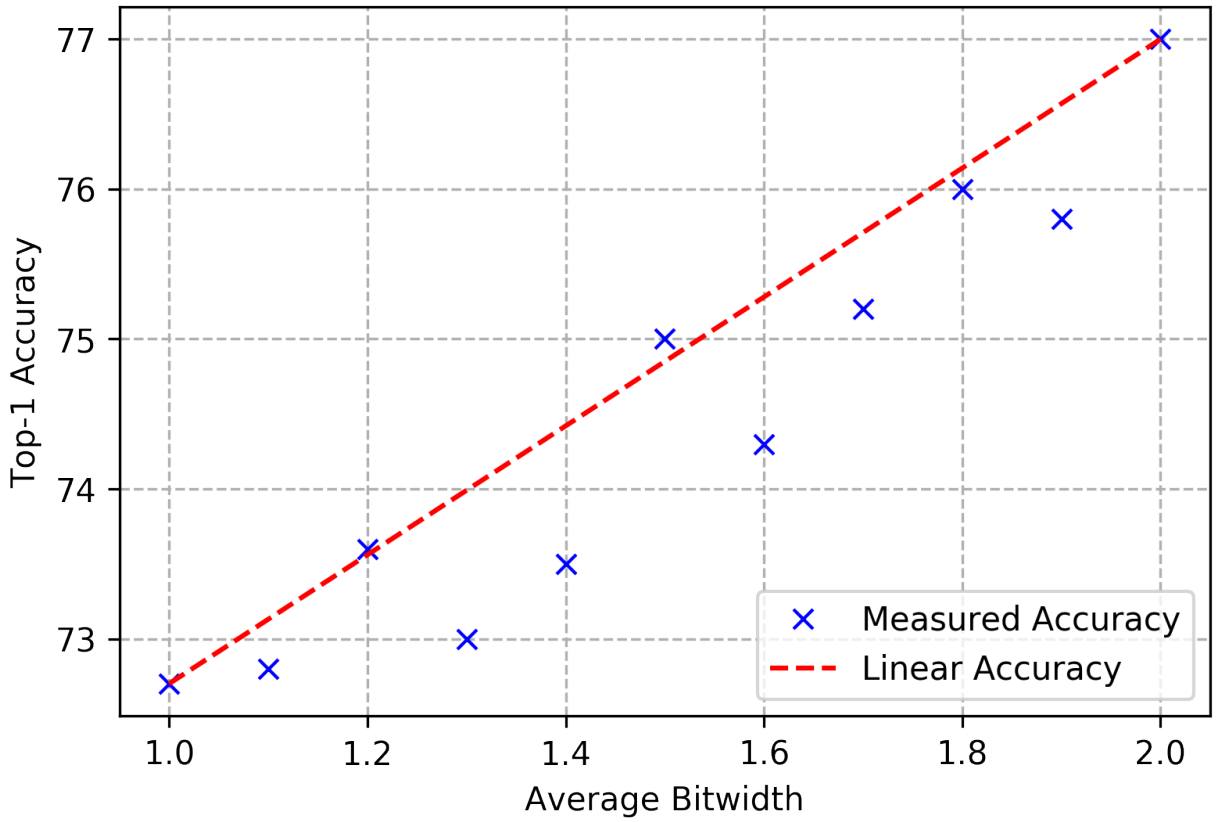


Figure 5.4: HBNN accuracy trained on CIFAR-10 with uninformed layer-level bit distribution.

5.3.2 Layer-level Heterogeneity

As a baseline, we test a "poor man's" approach to HBNNs, where we fix up front the number of bits each layer is allowed, require all values in a layer to have its associated bitwidth, and

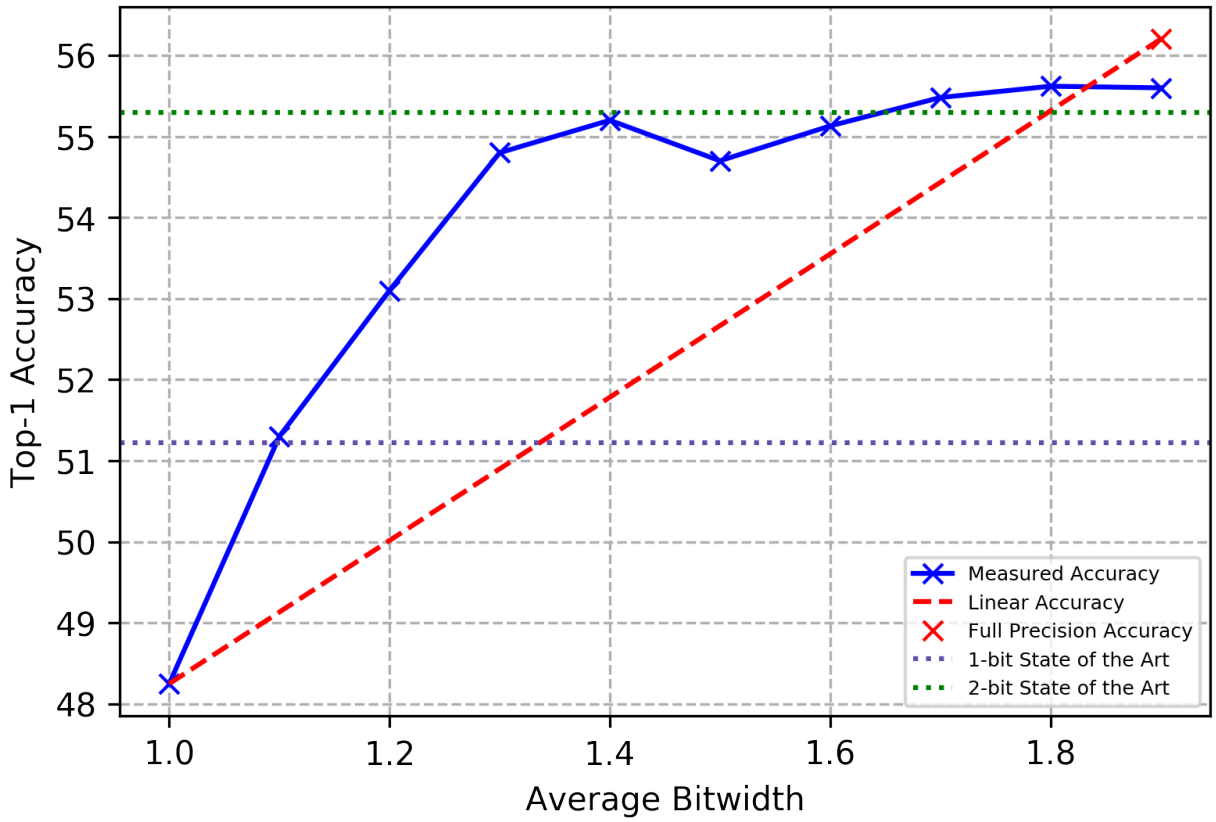


Figure 5.5: Accuracy results of trained HBNN models. (a) Sweep of heterogenous bitwidths on a deliberately simplified four layer convolutional model for CIFAR-10. (b) Accuracy of heterogeneous bitwidth AlexNet-BN models. Bits are distributed using the Middle-Out selection algorithm.

then train as with conventional homogeneous binarization. We consider 10 mixes of 1, 2 and 3-bit layers so as to sweep average bitwidths between 1 and 2. We trained as described in Section 5.3.1. For this experiment, we used the CIFAR-10 dataset with a deliberately hobbled (4-layer fully convolutional) model with a maximum accuracy of roughly 78% as the baseline 32-bit variant. We chose CIFAR-10 to allow quick experimentation. We chose not to use a large model for CIFAR-10, because for large models it is known that even 1-bit models have 32-bit-level accuracy [13].

Figure 5.4 shows the results. Essentially, accuracy increases roughly linearly with average

bitwidth. Although such linear scaling of accuracy with bitwidth is itself potentially useful (since it allows finer grain tuning on FPGAs), we are hoping for even better scaling with the "data-aware" bitwidth selection provided by HBNNs.

5.3.3 Bit Distribution Generation

As described in Section 5.2.2, one of the considerations when using HBNNs is how to take a desired average bitwidth and produce a matching distribution of bits. For example, using 70% 1-bit, 20% 2-bit and 10% 3-bit values gives an average of 1.4 bits, but so too does 80% 1-bit and 20% 3-bit values. We suspected that the choice of this distribution would have a significant impact on the accuracy of trained HBNNs, and performed a hyperparameter sweep by varying $DistFromAvg$ in Algorithm 9 when training AlexNet on ImageNet as described in the following sections. However, much to our surprise, **models trained with the same average bitwidth achieved nearly identical accuracies regardless of distribution**. For example, the two 1.4-bit distributions given above yield accuracies of 49.4% and 49.3% respectively. This suggests that choice of $DistFromAvg$ is actually unimportant, which is quite convenient as it simplifies training of HBNNs considerably.

5.3.4 AlexNet: Binarized Weights and Non-Binarized Activations

Recently, Dong et al. [17] were able to binarize the weights of an AlexNet-BN model to 2 bits and achieve nearly full precision accuracy (row 2 of Table 5.1). We consider this to be the state of the art in weight binarization since the model achieves excellent accuracy despite all layer weights being binarized, including the input and output layers which have traditionally been difficult to approximate. We perform a sweep of AlexNet-BN models binarized with fractional bitwidths using middle-out selection with the goal of achieving comparable accuracy using fewer than two bits.

The results of this sweep are shown in Figure 5.5. We were able to achieve nearly identical top-1 accuracy to the best full 2 bit results (55.3%) with an average of only 1.4 bits (55.2%). As we had hoped, we also found that the accuracy scales in a *super-linear* manner with

Table 5.1: Accuracy of HBNNs at Various Fractional Bitwidths Vs State-of-the-Art

	Model	Name	Binarization (Inputs / Weights)	Top-1	Top-5
		Binarized weights with floating point activations			
1	AlexNet	SQ-BWN [17]	full precision / 1-bit	51.2%	75.1%
2	AlexNet	SQ-TWN [17]	full precision / 2-bit	55.3%	78.6%
3	AlexNet	TWN (our implementation)	full precision / 1-bit	48.3%	71.4%
4	AlexNet	TWN	full precision / 2-bit	54.2%	77.9%
5	AlexNet	HBNN (our results)	full precision / 1.4-bit	55.2%	78.4%
6	MobileNet	HBNN	full precision / 1.4-bit	65.1%	87.2%
		Binarized weights and activations excluding input and output layers			
7	AlexNet	BNN [12]	1-bit / 1-bit	27.9%	50.4%
8	AlexNet	Xnor-Net [48]	1-bit / 1-bit	44.2%	69.2%
9	AlexNet	DoReFaNet [64]	2-bit / 1-bit	50.7%	72.6%
10	AlexNet	QNN [27]	2-bit / 1-bit	51.0%	73.7%
11	AlexNet	our implementation	2-bit / 2-bit	52.2%	74.5%
12	AlexNet	our implementation	3-bit / 3-bit	54.2%	78.1%
13	AlexNet	HBNN	1.4-bit / 1.4-bit	53.2%	77.1%
14	AlexNet	HBNN	1-bit / 1.4-bit	49.4%	72.1%
15	AlexNet	HBNN	1.4-bit / 1-bit	51.5%	74.2%
16	AlexNet	HBNN	2-bit / 1.4-bit	52.0%	74.5%
17	MobileNet	our implementation	1-bit / 1-bit	52.9%	75.1%
18	MobileNet	our implementation	2-bit / 1-bit	61.3%	80.1%
19	MobileNet	our implementation	2-bit / 2-bit	63.0%	81.8%
20	MobileNet	our implementation	3-bit / 3-bit	65.9%	86.7%
21	MobileNet	HBNN	1-bit / 1.4-bit	60.1%	78.7%
22	MobileNet	HBNN	1.4-bit / 1-bit	62.0%	81.3%
23	MobileNet	HBNN	1.4-bit / 1.4-bit	64.7%	84.9%
24	MobileNet	HBNN	2-bit / 1.4-bit	63.6%	82.2%
		Unbinarized (our implementation)			
25	AlexNet	[35]	full precision / full precision	56.5%	80.1%
26	MobileNet	[25]	full precision / full precision	68.8%	89.0%

respect to bitwidth when using middle-out bit selection. Specifically, the model accuracy increases extremely quickly from 1 bit to 1.3 bits before slowly approaching the full precision accuracy.

5.3.5 AlexNet: Binarized Weights and Activations

In order to realize the speed-up benefits of binarization (on CPU or FPGA) in practice, it is necessary to binarize both inputs the weights, which allows floating point multiplies to be replaced with packed bitwise logical operations. The number of operations in a binary network is reduced by a factor of $\frac{64}{mn}$ where m is the number of bits used to binarize inputs and n is the number of bits to binarize weights. Thus, there is significant motivation to keep the bitwidth of both inputs and weights as low as possible without losing too much accuracy. When binarizing inputs, the input and output layers are typically not binarized as the effects on the accuracy are much larger than other layers. We perform another sweep on AlexNet-BN with all layers but the input and output fully binarized and compare the accuracy of HBNNs to several recent results. Row 8 of Table 5.1 is the top previously reported accuracy (44.2%) for single bit input and weight binarization, while row 10 (51%) is the top accuracy for 2-bit inputs and 1-bit weights.

Table 5.1 (rows 13 to 16) reports a selection of results from this search. Using 1.4 bits to binarize inputs and weights ($mn = 1.4 \times 1.4 = 1.96$) gives a very high accuracy (53.2% top-1) while having the same number of total operations mn as a network, such as the one from row 10, binarized with 2 bit activations and 1 bit weights. We have similarly good results when leaving the input binarization bitwidth an integer. Using 1 bit inputs and 1.4 bit weights, we reach 49.4% top-1 accuracy which is a large improvement over Rastegari et al. [48] at a small cost. We found that using more than 1.4 average bits had very little impact on the overall accuracy. Binarizing inputs to 1.4 bits and weights to 1 bit (row 15) similarly outperforms Hubara et al. [27] (row 10).

5.3.6 MobileNet Evaluation

Although AlexNet serves as an essential measure to compare to previous and related work, it is important to confirm that the benefits of heterogeneous binarization is model independent. To this end, we perform a similar sweep of binarization parameters on MobileNet, a state of the art architecture that has unusually high accuracy for its low number of parameters [25]. MobileNet is made up of separable convolutions instead of the typical dense convolutions of AlexNet. Each separable convolution is composed of an initial spatial convolution followed by a depth-wise convolution. Because the vast bulk of computation time is spent in the depth-wise convolution, we binarize only its weights, leaving the spatial weights floating point. We binarize the depth wise weights of each MobileNet layer in a similar fashion as in Section 5.3.4 and achieve a Top-1 accuracy of 65.1% (row 6). This is only a few percent below our unbinarized implementation (row 26), which is an excellent result for the significant reduction in model size.

We additionally perform a sweep of many different binarization bitwidths for both the depth-wise weights and input activations of MobileNet, with results shown in rows 17-24 of Table 5.1. Just as in the AlexNet case, we find that MobileNet with an average of 1.4 bits (rows 21 and 22) achieves over 10% higher accuracy than 1-bit binarization (row 17). We similarly observe that 1.4-bit binarization outperforms 2-bit binarization in each permutation of bitwidths. The excellent performance of HBNN MobileNet confirms that heterogeneous binarization is fundamentally valuable, and we can safely infer that it is applicable to many other network architectures as well.

5.4 Hardware Implementability

Our experiments demonstrate that HBNNs have significant advantages compared to integer bitwidth approximations. However, with these representational benefits come added complexity in implementation. Binarization typically provides a significant speed up by packing bits into 64-bit integers, allowing a CPU or GPU to perform a single xnor operation in lieu of

Table 5.2: HBNN Custom Hardware Implementation Benefits

	Platform	Model	Unfolding	Bits	Occupancy	kFPS	P_{chip} (W)	Top-1
			CIFAR-10 Baseline Implementations					
1	ZC706	VGG-8	1×	1	21.2%	21.9	3.6	80.90%
2	ZC706	VGG-8	4×	1	84.8%	87.6	14.4	80.90%
3	ASIC	VGG-8	-	2	6.06 mm^2	3.4	0.38	87.89%
			CIFAR-10 HBNN Customization					
4	ZC706	VGG-8	1×	1.2	25.4%	18.25	4.3	85.8%
5	ZC706	VGG-8	1×	1.4	29.7%	15.6	5.0	89.4%
6	ZC706	VGG-8	4×	1.2	100%	73.0	17.0	85.8%
7	ASIC	VGG-8	-	1.2	2.18 mm^2	3.4	0.14	85.8%
8	ASIC	VGG-8	-	1.4	2.96 mm^2	3.4	0.18	89.4%
			Extrapolation to MobileNet with ImageNet Data					
9	ZC706	MobileNet	1×	1	20.0%	0.45	3.4	52.9%
10	ZC706	MobileNet	1×	2	40.0%	0.23	6.8	63.0%
11	ZC706	MobileNet	1×	1.4	28.0%	0.32	4.76	64.7%
12	ASIC	MobileNet	-	2	297 mm^2	3.4	18.62	63.0%
13	ASIC	MobileNet	-	1.4	145.5 mm^2	3.4	9.1	64.7%

64 floating-point multiplications. However, Heterogeneous tensors are essentially composed of sparse arrays of bits. Array sparsity makes packing bits inefficient, nullifying much of the speed benefits one would expect from having fewer average bits. The necessity of bit packing exists because CPUs and GPUs are designed to operate on groups of bits rather than individual bits. However, programmable or custom hardware such as FPGAs and ASICs have no such restriction. In hardware, each parameter can have its own set of n xnor-popcount units, where n is the bitwidth of that particular parameter. In FPGAs and ASICs, the

total number of computational units in a network has a significant impact on the power consumption and speed of inference. Thus, the benefits of HBNNs, higher accuracy with fewer computational units, are fully realizable.

There have been several recent binary convolutional neural network implementations on FGPAs and ASICs that provide a baseline we can use to estimate the performance of HBNNs on ZC706 FPGA platforms [61] and on ASIC hardware [2]. The results of these implementations are summarized in rows 1-3 of Table 5.2. Here, unfolding refers to the number of computational units placed for each parameter, by having multiple copies of a parameter, throughput can be increased through improved parallelization. Bits refers to the level of binarization of both the input activations and weights of the network. Occupancy is the number of LUTs required to implement the network divided by the total number of LUTs available for an FPGA, or the chip dimensions for an ASIC. Rows 4-12 of Table 5.2 show the metrics of HBNN versions of the baseline models. Some salient points that can be drawn from the table include:

- Comparing lines 1, 4, and 5 show that on FPGA, fractional binarization offers fine-grained tuning of the performance-accuracy trade-off. Notably, a significant accuracy boost is obtainable for only slightly higher occupancy and power consumption.

- Rows 2 and 6 both show the effect of unrolling. Notably, with 1.2 average bits, there is no remaining space on the ZC706. This means that using a full 2 bits, a designer would have to use a lower unrolling factor. In many cases, it may be ideal to adjust average bitwidth to reach maximum occupancy, giving the highest possible accuracy without sacrificing throughput.

- Rows 3, 7, and 8 show that in ASIC, the size and power consumption of a chip can be drastically reduced without impacting accuracy at all.

- Rows 9-13 demonstrate the benefits of fractional binarization are not restriced to CIFAR, and extend to MobileNet in a similar way.

Chapter 6

CONCLUSION

6.1 Making Binarization a Viable Tool

In this dissertation I present algorithms, systems, and applications that solidify binarization into a a practical and efficient method for improving the speed and memory requirements of neural networks. The ballooning size and computational complexity of state-of-the-art deep models demands that powerful optimization techniques be applied to make deployment on commodity hardware feasible.

Significant research has been directed towards binarization to fill this need due to its promise of order of magnitude speedups and memory compression. However, the bulk of machine learning research today focuses on training rather than inference and deployment, in large part due to the excellent ecosystem of tools for training neural networks that simply doesn't exist for deployment. Thus, previous work attempts to improve the accuracy of binary networks without attempting to implement and measure actual speedups. By taking an implementation first approach, we identify and address many key missing pieces needed to make binarization a viable optimization in practice.

In Chapter 3, we consider the many different possible methods to binarize a network and identify which are suitable for a high speed implementation. Even in these cases, there are many missing equations and algorithms needed which we introduce. Next, we examine the differences between binarizing a layer and binarizing an entire model. We find that in many models, binarizations speeds up core compute layers enough that the often ignored intermediate layers such as activations and batch normalization begin to have a significant impact on runtime. We introduce a novel fused glue operation that completely replaces all intermediate layers and can be directly applied to binary tensors. We denote the combina-

tion of viable quantization techniques and fused glue operations Riptide. By analyzing the computational complexity of a Riptide model, we find that it is dramatically more efficient than other approaches. We additionally demonstrate that Riptide is able to reach state of the art accuracies on numerous models.

We demonstrate how to convert a Riptide model into efficient machine code for commodity hardware, specifically the Raspberry Pi in Chapter 4. We adopt the use of TVM, a new tool that simplifies the compilation of machine learning models, and extend it to support binary networks. We analyze the impact of numerous scheduling primitives on mobile class models such as SqueezeNet. We additionally introduce a type of workload fusion that dramatically lowers intermediate memory usage in binary networks. The combination of these optimizations allows Riptide to produce end-to-end models that are up to 12× faster than full precision baselines. Given these speedups, we address the key question of whether binarization is able to produce a Pareto optimal model compared to other types of optimization. By considering a highly efficient baseline, ResNet18, we show that binarization does in fact provide a better speed to accuracy curve than reducing compute through downsizing of the input. Finally, we perform a case study application of binarization to voice activity detection, and demonstrate that binarization is capable of realizing impactful speedups in a real world application.

Having presented Riptide and carefully measured performance, we note that there are only a small handful of viable bitwidths that can be used in binarization. This significantly limits the points on the speed to accuracy curve that can be selected from. Many applications, however, have a very precise accuracy that needs to be reached at the highest possible speed. To enable a continous tradeoff between speed and accuracy, we introduce Heterogeneous Bitwidth Binary Networks in Chapter 5. We derive the Middle-Out bit selection algorithm and demonstrate that it produces high quality mixed-bit tensors. We train a number of HBNN networks at varying fractional bitwidths and find that they offer accuracy that is super-linear with respect to bitwidth, for example with 1.4 average bits equalling the accuracy of a full 2-bit network. We consider the suitability of HBNNs to be implemented efficiently

on custom hardware such as FPGAs and find that they offer compelling benefits compared to integer bitwitdh networks, indicating that even if heterogeneous bitwidths are difficult to handle in current hardware such as CPUS, that may not always be the case.

6.2 Riptide Today and Tomorrow

Riptide makes all the contributions presented in this thesis available for use via an open source GitHub repository. Today, Riptide enables push-button training of fully binarized networks that can be easily deployed on ARM CPU based platforms with order of magnitude speed up. Although this enables many IoT applications today it is admittedly somewhat limited in scope. However, Riptide is designed in a way that makes it extremely well suited as a platform for implementing new binarization techniques and easily deploying those techniques to a multitude of backends that can even include exotic targets like FPGAs and ASICs. We believe that this ease of extensibility may allow Riptide to grow into a sizeable community for ultra-low bitwidth researchers and practitioners.

Riptide is composed of a few critical pieces that each enable extensibility. At the highest level, Riptide is a high quality implementation of binary neural network training in TensorFlow, the most popular deep learning framework available [1]. Although this thesis primarily focused on the high speed inference of binary neural networks, it is important to remember that binarization is such a lossy approximation that the pretrained weights of full precision networks are no more useful than random initializations. This means that all binary networks must be trained from scratch. Training any network is a significant undertaking that too often relies on the intuition and experience of the practitioner, however, binary networks are even more finicky. BNNs are riddled with non-differentiable functions that require gradient overrides such as the straight-through estimator and are more sensitive to training hyperparameters than full precision networks. These effects jointly make training binary networks extremely difficult for newcomers in the field, especially considering all the variants of binary network that the community is interested in. Riptide alleviates this issue by providing a wide catalogue of training-ready low bitwidth operators. It is straightforward to combine

these operators into arbitrary or even new binarization techniques without having to worry about writing new gradient functions. Additionally, Riptide has many out-of-the-box training scripts with proper hyperparameter setup for yielding high accuracy networks. Thus, even without Riptide's contributions to deployment, it fills a crucial gap in being able to use binary networks and can serve as a foundation for future binary training projects.

Once a binary neural network is trained using Riptide, we provide all the plumbing to directly convert the trained model into a Relay representation [52], a graph IR that integrates with TVM [6]. Once the network is represented in Relay, it is simple to make use of lowering passes to apply graph level optimizations such as node fusion or heterogenous backend execution. By building out the infrastructure needed to convert models to Relay, Riptide paves the way for further work on exploring high level optimizations of binary neural networks. Additionally, Riptide is well integrated in the TVM infrastructure through rich support of bitserial operators. Although we primarily target ARM CPUs today, TVM is by design able to easily extend to new backends. It would be only a small amount of work, for example, to compile bitserial convolution to CUDA tensor cores. Enabling this mapping would immediately allow full Riptide models to run on cutting edge GPU accelerators. AutoTVM [7] support makes finding optimal schedules require very little engineering effort. We thus believe that Riptide provides all the tooling necessary to both develop new binary algorithms and architectures and deploy those models to emerging hardware.

6.3 CPUs, GPUs, and Custom Hardware

Throughout this work, we've primarily considered the application of binary networks to CPU based platforms. We argue that binarization is most reasonably applied in low cost platforms that require high speed and low memory usage. Today, it is very uncommon for platforms matching these criteria to have any type of processing besides a CPU. However, improving model performance on GPUs would have a considerable impact as well, since much cloud computing and virtually all training is performed on Nvidia GPUs. Until recently, though, GPUs were highly optimized only for floating point arithmetic and didn't even

support the core xnor-popcount operations required for fast binary implementations. With the introduction of tensor cores [39], GPUs made in the last year have experimental low-bit support that is perfectly suited to the techniques presented in this dissertation. Extension of Riptide to support GPU is a prime candidate for future work.

The deep learning boom also kicked off the exploration of custom hardware specifically designed for machine learning workloads. The most well known custom ML chip is Google's TPU [31], however many others are being developed. As hinted at in Chapter 5, binarization is extremely well suited to custom hardware as it requires only several gates per xnor accumulate compared to the thousands used in floating point multiply units. This dissertation may serve as motivation to develop new custom hardware that leverages binary networks more than one can in floating point based CPUs and GPUs. It is worth noting that chips designed for binary networks benefit significantly from the complete removal of floating point operations from a network due to the high cost of including floating point ALUs. The fused glue operation presented in Chapter 3 thus may be essential in such designs. It is important to remember that CPUs are highly optimized for floating point arithmetic. That we've demonstrated such significant speedups in this dissertation bodes extremely well for the future of binarization on more specialized hardware.

6.4 Recommendations for Future Directions

The work in this dissertation has addressed many of the lingering issues of binarization in the current literature and presented a system for deploying high speed binary networks. However, there remain many applications that binarization may be well suited to and questions that could expand the use cases of binary networks. I present recommendations based on these applications and unanswered questions for promising avenues of future work.

6.4.1 Vision Vs Language

The field of binarization has so far been limited to vision models, which universally use convolutional layers as their primary building block. However, there is nothing about bina-

rization that makes it better suited to vision tasks than other domains of machine learning. Instead, the focus on vision occurred due to the expertise of the original authors of binary networks, and has propagated in the same community since then. However, there is nothing about binarization that makes it better suited to convolutions than the dense layers more commonly used in Natural Language Processing (NLP) tasks. In fact, it's quite the opposite; as demonstrated by Ko et al. [33], the high compute intensity of raw matrix multiplies allows binarization to offer much higher speedups than memory bound convolutions. It is our belief that there should be more investigation in applying binarization to NLP domains going forward.

6.4.2 Finetuning

Current binrization techniques, including ours, require that a network be trained from scratch rather than from pretrained full precision weights. This is in contrast to many other optimizations. For example, when quantizing to 8-bits, the quantization can be directly applied to full precision weights and the resulting network often yields high accuracy without any retraining or even fine tuning. When dealing with large datasets like ImageNet, retraining a network from scratch can take hundreds of hours on a GPU, and GPU time is not an inexpensive resource. Indeed, the accuracy results presented in Tables 3.1 and 5.1 required thousands of GPU hours and many months of management to generate. We postulate that by quantizing to 8-bits, then iteratively reducing bitwidth and finetuning, the process of binarizing a network could be greatly expedited.

6.4.3 Combining Optimizations

Although we focus only on improving the state of network binarization, it is important to remember that binarization is one of many approximating optimizations. It remains an open question on how to combine multiple approximating optimizations to yield even faster models. Additionally, there is a concern that combining approximations may be disastrous to accuracy. However, at least one case is particularly promising. Block sparsity [42] improves

a networks performance by setting chunks of weights to have zero value. During inference, these chunks can be skipped over thereby improving runtime and memory usage. We note that HBNNs are essentially bit sparse, and by could be naturally combined with a block sparse implementation to both enable an efficient CPU implementation and offer significant speedups on top of those offered by block sparsity alone.

6.5 In Summary

As the accuracy of neural networks continues to skyrocket, the demand to deploy neural models in meaningful applications expands as well. However, state-of-the-art models have ever greater compute and memory requirements that often invalidate them from being used on inexpensive hardware. Network binarization offers a solution to this dilemma by promising order of magnitude compression and speedup. In this dissertation, we have identified and addressed many problems in existing binarization techniques and introduced novel algorithms that enable the first truly end-to-end binary network. We present Riptide, an open-source solution to training and deploying high speed networks. Additionally, we offer a novel algorithm for creating fractional bitwidth networks that may motivate the development of future low-bit hardware. It is our hopes that the combination of these efforts cements binarization is a viable and widespread tool for deploying neural networks in resource constrained settings.

ACKNOWLEDGEMENTS

I want to first thank my parents who have always done everything in their power to make my life as easy as possible so that I could focus on this Ph.D. Without your ongoing love, support, and advice I wouldn't be the person I am today. I also want to thank my partner Meiling for always pushing me to be the best version of myself. Having you by my side has made this journey the best time of my life.

To my advisor Shwetak; thank you for giving me the freedom to explore my passion. My research has changed dramatically multiple times but you've been immeasurably supportive throughout it all. Thank you for always going far out of your way to find opportunities for me to learn and grow.

To Matthai; thank you for shaping my Ph.D. and teaching me so much along the way. Without your continual patience, mentorship, and ability to keep me on track none of this would be possible. Thank you for seeing my potential and helping me realize it. I sincerely look forward to our continuing collaboration going forward.

To my labmates in the Ubicomp lab; thank you for creating a fun and supportive environment to work and learn in. Mayank, Keyu, Hanchuan, and Elliot. Thank you for taking me under your wing when I had no idea what I was doing and teaching me so much. Your mentorship helped me understand how to perform research and gave me a broad range of experience to build off of. Alex, Edward, Lilian, Ruth, Farshid, Mohit, Xin, Morelle. It's hard to imagine what the lab would be like without all your strong personalities. I've truly enjoyed getting to spend so much time with you and work with you all these years. Eric was there too sometimes.

To the members of the SAMPA group; thank you for being so welcoming and making me feel like a valued member of the team. Luis, Thierry, Tianqi, Meghan, Jared and Eddie.

I've learned so much in getting to know you and appreciate you all putting up with all my questions. Without your support much of the work I did would be impossible to realize in a meaningful way. I look forward to continuing to build great things together.

To my many internship mentors Steve, Ofer, Geremy, Ankit, Lubomir, and Oren; Thank you for giving me the opportunity to explore new fields and learn new things. I'm lucky to have been able to work on such a wide range of projects and will continue to value the broad foundation you all helped build.

BIBLIOGRAPHY

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[2] Hande Alemdar, Vincent Leroy, Adrien Prost-Boucle, and Frédéric Pétrot. Ternary neural networks for resource-efficient ai applications. In *Neural Networks (IJCNN), 2017 International Joint Conference on*, pages 2547–2554. IEEE, 2017.

[3] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[4] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. Deep learning with low precision by half-wave gaussian quantization. *arXiv preprint arXiv:1702.00953*, 2017.

[5] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. An analysis of deep neural network models for practical applications. *arXiv preprint arXiv:1605.07678*, 2016.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: End-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[7] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166*, 2018.

[8] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[10] Jungwook Choi, Pierce I-Jen Chuang, Zhuo Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Bridging the accuracy gap for 2-bit quantized neural networks (qnn). *arXiv preprint arXiv:1807.06964*, 2018.

[11] Intel Corporation. *Intel Math Kernel Library Reference Manual*. 2009.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in Neural Information Processing Systems*, pages 3123–3131, 2015.

[13] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[14] Meghan Cowan, Thierry Moreau, Tianqi Chen, and Luis Ceze. Automating generation of low precision deep learning operators. *arXiv preprint arXiv:1810.11066*, 2018.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[17] Yinpeng Dong, Renkun Ni, Jianguo Li, Yurong Chen, Jun Zhu, and Hang Su. Learning accurate low-bit deep neural networks with stochastic quantization. *arXiv preprint arXiv:1708.01001*, 2017.

[18] Andreas Ehliar. Area efficient floating-point adder and multiplier with ieee-754 compatible semantics. In *Field-Programmable Technology (FPT), 2014 International Conference on*, pages 131–138. IEEE, 2014.

[19] Clément Farabet, Yann LeCun, Koray Kavukcuoglu, Eugenio Culurciello, Berin Martini, Polina Akselrod, and Selcuk Talay. Large-scale fpga-based convolutional networks. *Scaling up Machine Learning: Parallel and Distributed Approaches*, pages 399–419, 2011.

[20] Josh Fromm, Shwetak Patel, and Matthai Philipose. Heterogeneous bitwidth binarization in convolutional neural networks. *arXiv preprint arXiv:1805.10368*, 2018.

[21] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

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

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[25] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[26] Yuwei Hu, Jidong Zhai, Dinghua Li, Yifan Gong, Yuhao Zhu, Wei Liu, Lei Su, and Jiangming Jin. Bitflow: Exploiting vector parallelism for binary neural networks on cpu. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 244–253, 2018.

[27] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

[28] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[29] MKL Intel. Intel math kernel library. 2007.

[30] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[31] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[32] Nal Kalchbrenner, Erich Elsen, Karen Simonyan, Seb Noury, Norman Casagrande, Edward Lockhart, Florian Stimberg, Aaron van den Oord, Sander Dieleman, and Koray Kavukcuoglu. Efficient neural audio synthesis. *arXiv preprint arXiv:1802.08435*, 2018.

[33] Jong Hwan Ko, Josh Fromm, Matthai Philipose, Ivan Tashev, and Shuayb Zarar. Precision scaling of neural networks for efficient audio processing. *arXiv preprint arXiv:1712.01340*, 2017.

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

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[36] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[37] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[38] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[39] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[40] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. A hardware-software blueprint for flexible deep learning specialization. *arXiv preprint arXiv:1807.04188*, 2018.

[41] Wojciech Muła, Nathan Kurz, and Daniel Lemire. Faster population counts using avx2 instructions. *The Computer Journal*, 61(1):111–120, 2017.

[42] Sharan Narang, Eric Undersander, and Gregory Diamos. Block-sparse recurrent neural networks. *arXiv preprint arXiv:1711.02782*, 2017.

[43] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[44] Florent Perronnin, Yan Liu, Jorge Sánchez, and Hervé Poirier. Large-scale image retrieval with compressed fisher vectors. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 3384–3391. IEEE, 2010.

[45] Raspberry Pi. Raspberry pi model b, 2015.

[46] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[47] Aswin Raghavan, Mohamed Amer, and Sek Chai. Bitnet: Bit-regularized deep neural networks. *arXiv preprint arXiv:1708.04788*, 2017.

[48] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[49] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference onG computer vision and pattern recognition*, pages 779–788, 2016.

[50] Oren Rippel and Lubomir Bourdev. Real-time adaptive image compression. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2922–2930. JMLR. org, 2017.

[51] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G Anderson, and Lubomir Bourdev. Learned video compression. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3454–3463, 2019.

[52] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: a new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 58–68. ACM, 2018.

[53] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[54] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?(no, it is not about internal covariate shift). *arXiv preprint arXiv:1805.11604*, 2018.

[55] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[56] Doug Stiles. The hardware security behind azure sphere. *IEEE Micro*, 39(2):20–28, 2019.

[57] Xing Su, Xiangke Liao, and Jingling Xue. Automatic generation of fast blas3-gemm: A portable compiler approach. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 122–133. IEEE, 2017.

[58] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. *Neural processing letters*, 9(3):293–300, 1999.

[59] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[60] Wei Tang, Gang Hua, and Liang Wang. How to train a compact binary neural network with high accuracy? In *AAAI*, pages 2625–2631, 2017.

[61] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

[62] Zhang Xianyi, Wang Qian, and Zaheer Chothia. Openblas. *URL: http://xianyi. github. io/OpenBLAS*, 2014.

[63] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044*, 2017.

[64] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160*, 2016.