Simplification Of Deep Neural Networks For Efficient Inference

JULIAN FARAONE

B.Eng (Hons) & B.Com

Supervisor: Philip H.W. Leong
Associate Supervisor: David Boland

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy

School of Electrical and Information Engineering
Faculty of Engineering
The University of Sydney
Australia

10 August 2021

Abstract

In recent years, Deep Neural Networks (DNNs) have become an area of high interest due to it's ground-breaking results in many fields and applications. In many of these applications however, the model's runtime and memory cost of computing inference is more important than the cost of training the model. Inference is computationally expensive, making them difficult to deploy in constrained hardware environments. This has lead to an increasing interest in recent years for model compression techniques for these models.

In this thesis, model compression techniques are presented for achieving efficient representations of DNNs for hardware acceleration. Firstly, a weight pruning technique to achieve unstructured sparse representations of bitwise DNNs is explored on the MNIST and CIFAR10 datasets. Accompanying this, is a hardware exploration of the resulting representations. Secondly, a hardware-aware filter pruning technique to achieve structured sparse representations of bitwise DNNs is investigated on the ImageNet dataset and hardware performance improvements are evaluated via a Field Programmable Gate Array (FPGA) implementation. Thirdly, a quantization method is introduced for training highly accurate bitwise networks with high computational efficiency on the ImageNet dataset. A hardware architecture is designed for this representation and its performance evaluated via FPGA simulations. Lastly, a custom arithmetic is designed which utilizes FPGA-optimized multipliers. Additionally, a training methodology is presented which is customized for DNN models to be compatible with the multiplier.

Together, this work illustrates the effectiveness of designing DNNs with hardware in mind. Adjunctly, designing customized hardware helps in optimizing accuracy and hardware efficiency. This is very useful for many real-world DNN applications where hardware performance is paramount.

Acknowledgements

First and foremost I'd like to thank my supervisor Professor Philip H.W. Leong. He is an outstanding supervisor. I am so grateful for his guidance throughout this PhD journey. On reflection, I learnt many valuable skills from him which will benefit my career immensely. Specifically, I learnt the ability to define a problem, how to investigate that problem thoroughly and present results such that others will benefit from the knowledge. Professor Leong's passion for the area of machine learning and hardware is infectious and this allowed me to stay effortlessly motivated throughout the PhD. His foresight and ability/willingness to help all his students on a deeply technical level is truly inspiring and something I will strive towards in my own endeavours.

I'd also like to thank my friends and lab mates from the Computer Engineering Lab at The University Of Sydney. Firstly, my co-supervisor Dr. David Boland. Having another senior staff member to consult with was highly beneficial and provided great perspective, ideas and discussions. In addition, other PhD students: Sean Fox, Dr. Stephen Tridgell, SeyedRamin Rasoulinezhad, Dr. Siddartha and Dr. Duncan Moss. These people provided me with great assistance and discussions throughout my PhD. It has been a great environment and culture to work in.

I also had the pleasure of collaborating with many other great researchers from industry along this journey. Internships at Xilinx in Dublin, Ireland, Xilinx in San Jose, USA and Mythic-AI in San Francisco, USA improved my technical ability and gave me new perspective on my research area. This experience was invaluable and helped improve my research significantly. I sincerely thank Dr. Michaela Blott, Dr. Nicholas Fraser, Dr. Giulio Gambardella, Dr. Kees Vissers, Dr. Stephen Neuendorffer and Dr. Mark Beardslee. I learnt a great deal from working with each one of you.

I was also lucky enough to collaborate with other universities outside of The University Of Sydney. I would like to thank Dr. Martin Kumm, Martin Hardieck and Professor Peter Zipf from The University Of Kassel and also Dr. Jiang Su from Imperial College London.

I would also like to thank my friends and family for their love and support throughout this. Being able to relax and enjoy myself with you whilst away from my PhD was as important as doing the work itself. Most notably, my parents. My mother's continued support, especially on life around my PhD was immensely helpful. Also, I'd like to give a huge thank you to my late father, who unfortunately passed away during my final years over in Perth, Western Australia. His mentorship throughout my PhD was highly significant in keeping me on track. We had countless chats regarding my progress and opportunities to improve. He always ensured I was prioritizing this PhD first over other life distractions. I am forever grateful.

Author Statement

All the content of this thesis is a product of my own work. All assistance in the preparation of this thesis have been acknowledged as follows:

- Professor Philip H.W. Leong provided the research direction for all this work
- The hardware cost models and implementation in Chapter 3 was done in collaboration with Dr. Giulio Gambardella and Dr. David Boland.
- The SYQ hardware implementation was done in collaboration with Dr. Nicholas Fraser
- The AddNet idea was conceived in discussion with Dr. David Boland, Dr. Martin Kumm and Martin Hardieck. The design of the multipliers was done by Dr. Martin Kumm and Martin Hardieck.

Julian Faraone - 14/11/2020

Publications

The work in this thesis has been published in journals, conferences and as patents. The publication titles and their corresponding publication destinations are stated as follows:

Journal Publications:

- **Julian Faraone**, Martin Kumm, Martin Hardieck, Peter Zipf, Xueyuan Liu, David Boland, Philip HW Leong - "AddNet: Deep Neural Networks Using FPGA-Optimized Multipliers" - 2019 IEEE Transactions on Very Large Scale Integration Systems (TVLSI)

Conference Publications:

- **Julian Faraone**, Nicholas Fraser, Giulio Gambardella, Michaela Blott, Philip HW Leong - "Compressing low precision deep neural networks using sparsity-induced regularization in ternary networks" - 2017 International Conference on Neural Information Processing (ICONIP)
- **Julian Faraone**, Giulio Gambardella, Nicholas Fraser, Michaela Blott, Philip Leong, David Boland - "Customizing low-precision deep neural networks for FPGAs" - 2018 IEEE International Conference on Field Programmable Logic and Applications (FPL)
- **Julian Faraone**, Nicholas Fraser, Michaela Blott, Philip HW Leong - "Syq: Learning Symmetric Quantization For Efficient Deep Neural Networks" - 2018 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)
- Jiang Su, **Julian Faraone**, Junyi Liu, Yiren Zhao, David B Thomas, Philip HW Leong, Peter YK Cheung - "Redundancy-reduced MobileNet acceleration on reconfigurable logic for ImageNet classification" - 2018 International Symposium on Applied Reconfigurable Computing (ARC)

Patents:

- **Julian Faraone**, Michaela Blott, Nicholas Fraser - “System and Method For Implementing Neural Networks In Integrated Circuits” - June 2018. Filed with the US Patent and Trademark Office

Contents

List of Figures

CHAPTER 1

Introduction

Data has overtaken oil as the most valuable commodity in the world [10]. A majority of the world's most valuable companies have been successful due to their expertise in monetizing information from data [29]. The ability to collect, analyze and make data-driven decisions has now become crucial for governments, companies and research institutions to innovate and increase productivity and efficiency [113]. In the current technological age, the quantity of data has exploded as information becomes increasingly digitized [59]. When records and other information are digitized, it provides an opportunity for applying machine learning algorithms to turn raw data into knowledge [136]

Machine learning is a statistical learning algorithm whereby decision-making computer systems, without explicit programming, can be made. This is particularly useful for applications whereby it is infeasible to explicitly program rules for a computer to act upon. A superior machine learning algorithm for many applications are Deep Neural Networks (DNNs). Having access to large amounts of data can be very effective at training highly accurate DNNs, especially for complex datasets.

Achieving both high accuracy and high hardware performance is ideal for all DNN applications. However, particularly in resource-constrained environments, achieving both of these becomes difficult. For some applications, high accuracy is more important and low hardware performance is sufficient, and vice versa. For example, using DNNs for life-critical decision making, such as diagnosing certain diseases through medical imaging, requires high accuracy whereas extreme speed of the decision is likely to be not as important. On the other hand, using DNNs for targeted advertisements on social media would require fast decisions, however

the most optimal output is not critical. Thus, using different compression techniques is useful to meet the requirements of the application at hand.

Software implementations of DNNs typically use 32 bits of precision to represent floating point numbers on general purpose hardware such as CPUs and GPUs. However, it is now known that less precision can be used for training and inference [122, 88, 94, 118, 36]. This poses advantages for specialized hardware units such as application-specific integrated circuits (ASICs) and field programmable gate arrays (FPGAs), because the arithmetic can be customized for any bitwidth, which reduces computational and memory complexity. For low-precision, such as 1-8 bits, specialized hardware platforms can yield much higher computational performance.

For inference computation, it has been shown that only 1 and 2 bits of precision for weights and activations can be sufficient to achieve floating point accuracy [148]. In these cases, networks are represented by binary and ternary values. These are known as *bitwise* networks because Multiply and Accumulate operations (MACs) are replaced by bit operations. As the number of commercial applications of DNNs has increased in recent years, the ability to design fast, low-power hardware has become very significant. For DNN applications which do not require personalization, once a model is trained it can be deployed to compute inference, without the need for updating the model. This can be useful for implementing DNNs on embedded platforms and mobile platforms where hardware resources are restricted.

When deploying DNNs on embedded platforms and other resource-constrained hardware environments, their size and computational complexity significantly impacts hardware performance. Many state-of-the-art networks consist of millions/billions of operations to compute inference per image and consume large amounts of storage, making them increasingly difficult to deploy on embedded platforms. For example, EfficientNet-B7 [123] achieves state-of-art-results on the ImageNet challenge which requires 66 million parameters for storage and 37 billion operations to compute inference per image.

1.1 Aims and Contributions

The aim of this thesis is to develop DNN training techniques which produce high accuracy and result in network representations which are amenable to custom hardware implementations. Furthermore, this thesis aims to explore different degrees of freedom to improve the accuracy-hardware performance trade-off of DNNs. The goal is to push the envelope in achieving different accuracy-hardware performance trade-offs, particularly on embedded platforms, making these techniques relevant to applications which both place a higher importance on accuracy and/or hardware performance.

For embedded platforms, the highest hardware performance is achievable from quantization to bitwise representations for DNNs. The first contribution of this thesis explores how to utilize sparsity to enhance hardware performance beyond this limitation, enabling implementations on smaller devices. Novel training techniques are introduced which maintain accuracy whilst introducing high sparsity. The drawback with the above method is the requirement for additional hardware, making it only compatible for particular hardware architectures. As such, the next main contribution is to introduce sparsity in a way that doesn't require additional hardware. This improves hardware performance further and makes it compatible with many already existing DNN hardware implementations. However, the absolute accuracy from both these methods is limited. Hence, rather than introduce sparsity to a bitwise network, their large information loss is overcome by introducing additional ordered scaling factors. Ordering the scaling factors allows us to achieve higher accuracy, whilst preserving the hardware simplicity of the representation. This maintains the network's applicability to embedded devices. The final contribution is to take accuracy improvement one step further by increasing the precision of the network with minimal impact on hardware usage. This is achieved by developing a family of novel arithmetic multipliers which utilizes specialized adders which require low resource consumption, and a corresponding method to train networks to use these resources. This provides an improvement in the accuracy-hardware performance tradeoff against conventional methods of increasing precision.

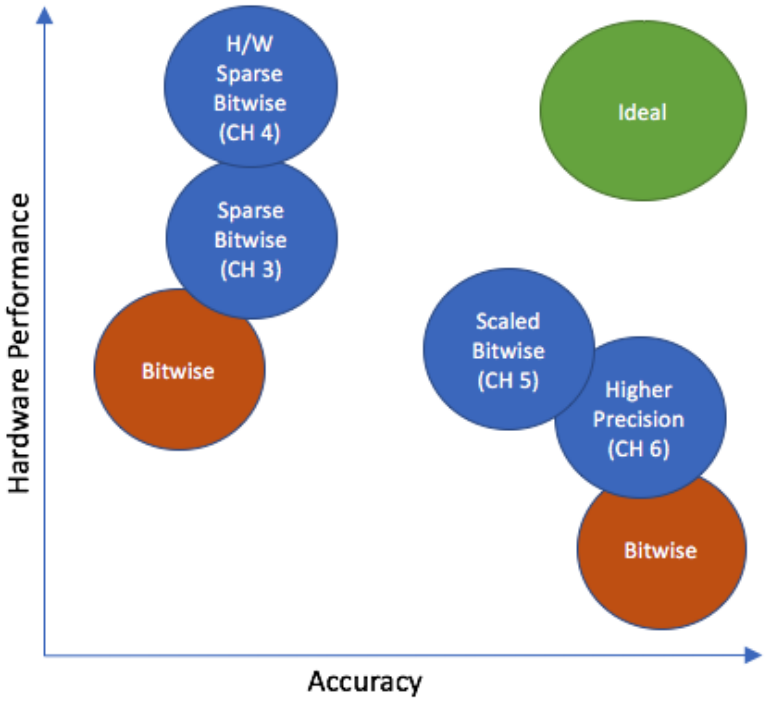


FIGURE 1.1. The accuracy-hardware performance tradeoff of each of the methods explored in this thesis.

Figure 1.1 presents a visualisation of these contributions in terms of accuracy and hardware performance. For a full-precision floating point network, it is known we can achieve the accuracy required for many applications but not quite the hardware performance, particularly on edge devices. On the other end of the spectrum, we have bitwise networks where we quantize weights and/or activations to 1-bit and improve on the performance but suffer accuracy degradation. Ideally, we could get to this green bubble and this is where we have both high accuracy and performance. These types of implementations are not quite achievable yet. However, in this thesis, the aim is to improve on these red bubble benchmarks here and previous state-of-the-art methods and get closer to this ideal spot.

We essentially split this work up into 4 parts: In Chapter 3 & 4 bitwise networks are used which lack accuracy, however can generally achieve higher hardware performance than the networks with higher precision used in Chapters 5 & 6. In all the methods, an improvement in the accuracy-hardware performance trade-off is demonstrated over standard bitwise networks

and also recent state-of-the-art related work. The methods and contributions of each Chapter are now discussed in greater detail:

The technique to maintain accuracy of bitwise networks whilst introducing sparsity is described in Chapter 3. The aim is to use sparsity to improve the performance of hardware implementations of bitwise networks on image classification tasks. This involves pruning weight parameters, meaning many of the operations can be ignored. The challenge with this method is the potential for information loss causing accuracy degradation. Thus, an informative method is required to determine which weights are less important than others. Previous methods sparsify or quantize a network at different stages of training [37, 49]. However, in this method, the quantization function is used to sparsify the network and include a cost function with L2 regularization to enhance sparsity. Additionally, sensitivity information from the solution found by training is used to distinguish how aggressively each layer is pruned. The main contribution is the ability to train bitwise networks with equivalent accuracy but significantly higher sparsity than previous work. Using lossless compression algorithms, this can save memory and lead to an improvement in throughput when a hardware decompressor is instantiated. This is demonstrated by evaluation of a hardware cost model which targets FPGA platforms.

To make a sparse representation which is amenable to DNN hardware implementations, in Chapter 4 groups of weights are set to zero, whereby the groups constitute convolutional kernels. This effectively removes the kernel as it's whole computation can be ignored and hence improves hardware performance significantly for various platforms. With this method, typically less weight parameters can be pruned in total with respect to techniques used in Chapter 3. However, both the amount of sparsity and the target hardware is considered. The main contribution is a kernel sparsity method for customizing bitwise DNNs to common FPGA dataflows. Unlike traditional kernel sparsity methods which follow sensitivity metrics to determine the amount to prune in each layer, layers are chosen based on their hardware costs (i.e. the amount of resources consumed). Which kernels are most important in that layer are then deciphered based on the total quantization error of each kernel. The representations are evaluated via an FPGA implementation. Using this technique, accuracy is maintained

whilst significantly improving throughput. The networks are evaluated on the AlexNet [69] network using the ImageNet dataset [116].

To improve the accuracy of the networks, in Chapter 5, the aim is to develop a quantization function which learns efficiently and preserves the simplicity of the bitwise representations. The issue with bitwise networks is there is large information loss as weights are quantized to 1-2 bits. The main contribution is the introduction of fine-grained learnable scaling factors to recover this loss, which maintain data regularity, making them amenable to hardware implementations. The technique is evaluated across a range of benchmark networks, including AlexNet, ResNet [55] and VGG [120] on the ImageNet dataset, achieving state-of-the-art results for various precision. To demonstrate the area reduction benefits, a hardware design is then presented for custom hardware platforms which can exploit this representation. Lastly, this design is evaluated via a resource exploration on an FPGA which demonstrates very minimal hardware cost implications.

Finally, the aim is to investigate networks with higher precision to achieve higher accuracy, whilst attempting to maintain high hardware performance. Given the same arithmetic method, the issue with increasing precision is that it typically increases the hardware cost. Thus, in Chapter 6, a custom arithmetic is derived which is suited to a particular low-level FPGA architecture which minimizes this cost. Digital multipliers are firstly designed which fit tightly into its slices to reduce area consumption. A customized training methodology is then described targeting these multipliers which are restricted by their input coefficient sets. This uses a distribution matching technique and a single learnable scaling factor. A network with this representation is then trained to make it compatible with the multiplier and evaluate these results on AlexNet and ResNet on the ImageNet dataset. Finally, an FPGA implementation of the work is provided and results are compared for accuracy and resource consumption against methods using conventional digital arithmetic. This benefits of this type of arithmetic are demonstrated over traditional fixed-point arithmetic.

Through the various techniques in all the Chapters, the following contributions are presented:

- The first training technique for sparse DNNs which minimizes hardware costs as part of the objective function [32].
- The first filter pruning training technique tailored to low-precision DNNs [33].
- A novel resource-aware training method for customizing low precision DNNs to underlying FPGA dataflow architectures [33].
- A novel quantization technique for improving the ability of convolutional weights to learn low-precision representations whilst maintaining hardware efficiency. This improved upon previous state-of-the-art networks for Top-1 ImageNet classification for binarized neural networks by 2-8% [34].
- A hardware architecture to exploit the resulting representation of this quantization technique [34].
- An open source tensorflow implementation of this quantization technique [34].
- A novel arithmetic for computing DNN inference which is tailored to the FPGA fabric. This significantly reduces resource requirements over conventional arithmetic [31].

1.2 Thesis Structure

A background of concepts discussed and terminology used throughout this thesis is provided in Chapter 2. The chapter begins with an overview of Deep Learning and the types of Deep Learning models used throughout the thesis. Then detail of how these networks are trained is given, using a simple neural network example. Following this, popular data representations are introduced for digital arithmetic. Quantization is then discussed, including explicit definitions for bitwise networks. After this, common methods of learning such representations are discussed during training. Finally, specialized hardware is introduced and more specifically, FPGAs and why FPGAs are used to evaluate most of this work. Lastly, the applications the methods are evaluated on are described and details are given of the specific benchmark datasets used throughout the thesis.

In Chapter 3, methodologies for inducing sparsity during training by pruning individual weight parameters are explored. Details of the quantization functions, cost function, pruning techniques and compression methods are described. Furthermore, a training algorithm is discussed in detail. This training process is evaluated on several benchmark datasets and networks. Finally, a hardware cost model for these networks is designed and evaluated.

Next, in Chapter 4, derivations for bitwise networks are re-introduced which are related to this Chapter. Background information is then provided regarding common DNN acceleration methods on various hardware platforms such as CPUs, GPUs and FPGAs. Following this, the layer selection and kernel filter selection methods are described. A training algorithm is then provided for achieving the desired representations. This training method is then evaluated on both Image Classification and Object Detection tasks and accuracies are reported for various sparsities. Finally, the hardware performance is evaluated via throughput measurements of an FPGA implementation.

In Chapter 5, the quantization functions are described, which are used for this technique. This includes the learnable scaling factors. The granularity of the scaling factors is then explored and how they are arranged within each layer's weight matrix. Functions for initializing the network and also quantizing the activations are presented. Following this, an algorithm to train for the desired representations is shown. The method is evaluated on an Image Classification task for various benchmark networks and for various precision. The computational complexity of this method is explored against previous state-of-the-art methods. Finally, the hardware design is illustrated and the hardware performance is evaluated via simulations targeting an FPGA platform.

Furthermore, in Chapter 6, the required background for small softcore multipliers is firstly introduced. Next, reconfigurable constant coefficient multipliers (RCCMs) are explained using examples and illustrations. How each multiplier fits tightly into the FPGA architecture is then discussed, before displaying the exact RCCMs used in this work. A training algorithm is described with details of the quantization function for weights and activations. A hardware accelerator system design is then presented with a detailed description of the experimental

setup. The hardware is then evaluated for resource usage, power consumption and frequency. Accuracy of the networks are also evaluated on Image Classification.

In the final chapter, the contributions of the thesis are summarized and possible avenues for future research is discussed.

In this thesis, hardware performance is used as a general term for many hardware metrics such as memory, latency, throughput, resource usage/area, power etc. The idea is to show improvements for at least one of these metrics in each of the methods described throughout the thesis. These metrics are measured via physical hardware implementations or via hardware cost models which predict values based on some assumptions.

Mathematical notation is introduced in the background chapter of this thesis. In the subsequent chapters, some of the equations are reused and symbols are redefined. The symbols relating to one chapter are only meaningful to that particular chapter.

CHAPTER 2

Background

In this chapter, we introduce background ideas and terminology spanning across Deep Learning, numerical precision and computer architecture which are relevant to this thesis. Firstly, we describe what DL is, how their models are designed, trained and their computational requirements. We then describe some common numerical data representations found in computers. Following this, we discuss compression methods such as quantization and pruning for DNNs and how these are used to reduce their computational load in hardware. We then outline the application areas targeted throughout this thesis. Finally, we provide a background on hardware implementations of DNNs and in particular for FPGAs.

2.1 Deep Neural Networks

Deep Learning is the sub-field of Machine Learning concerned with the application of biologically-inspired models to perform an AI function such as detecting/classifying objects and recognizing/translating speech [80, 153]. These models are typically known as Deep Neural Networks (DNNs). DNNs are typically composed of many intermediate computations, known as *layers*, whereby higher level features are progressively extracted from the raw input [122]. The length of the chain of layers gives us the depth of the model. The term "deep" in DNNs, is generally concerned with networks consisting of many layers. DNNs can perform tasks effectively, particularly in learning complex mappings from large amounts of exemplar data. DNNs currently are the best approach for many previously intractable problems such as image classification, object detection, and machine translation. In this thesis,

we are concerned with making implementations of DL models which can be executed with improved speed, power and accuracy.

A DNN can perform classification of an input $x \in \mathbb{R}^n$ to category $y \in \mathbb{R}^z$, where n and z are the dimensionality of the input and output arrays/vectors, respectively. A DNN finds some mathematical function $y = f(x; \theta)$, where θ is the set of learnt parameters found during the training phase. These parameters are typically known as weight and bias parameters. The goal of the DNN is to find the function which produces predictions y which most accurately match the target outputs $\hat{y}$. In supervised learning (which is what we are concerned with in this thesis), each input x is accompanied by a target output $\hat{y}$. During the training phase, we update the learnt parameters to minimize an error function, which is a function of both y and $\hat{y}$. Feedforward networks compute y by passing the input x through layers, which are used to form the structure of f.

In the testing phase, unseen data is fed into the model and the performance is evaluated based on its ability produce a desired output based on input data [8]. The difference between the error on the training data (the training error) versus the error on the test data is referred to as the generalization error. Having two solutions which fit the training data equally, the representation which reduces the generalization error will be preferred as the final solution. This is because it will perform better on unseen test data. This is the idea of regularization. Regularization is the idea of adding a preference for certain representations of functions over others, without changing the hypothesis space size of potential functions of the model [106]. Generally, regularization methods are used to reduce the generalization error but not necessarily the training error.

In this thesis, we focus on the implementation of feedforward networks, mainly in the form of Convolutional Neural Networks (CNNs). However, many of our techniques could also be applied to recurrent neural networks. Additionally, we use compression methods which add preferences for certain representations, similar to regularization. However, we use these methods to choose representations which optimize an accuracy-hardware tradeoff.

2.1.1 Inference

The structure of a DNN approach is a repeated application of a simple nonlinear function $g(x)$ at each layer l. This is typically a sigmoid function, relu function, hyperbolic tangent function, etc. Each layer of a DNN computes dot products between weight parameters and its input values. The input layer represents the input vector of information $\boldsymbol{x}$ and the output layer is the last layer which produces an output vector $\boldsymbol{o}$. All layers in between are called hidden layers because the training data does not show the desired output for these layers. The desired output is only shown at the output layer [44]. By adding layers or units per layer, we can represent functions of increasing complexity.

The most classic form of DNNs are multi-layer perceptrons [135]. These consist of full-connected layers whereby each neuron in a hidden unit is connected to every input. Suppose a network has L layers with 1 and L being the input and output layers, respectively. Also, layer l for $l = 1, 2, 3, ..., L$ has $\hat{n}_l$ neurons, so that $\hat{n}_1$ is the dimension of the input data and $\hat{n}_L$ is the dimension of the output data. Then, from the input layer to the output layer, the network maps from $\mathbb{R}^{\hat{n}_1} \rightarrow \mathbb{R}^{\hat{n}_L}$. The weight matrix at layer l follows $\boldsymbol{W}^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$. Each weight parameter is then denoted as $w_{ij}^{[l]}$, which is the weight at neuron i that layer l applies to the output from neuron j at layer $l-1$. Each bias parameter $b_i^{[l]}$ used by neuron i at layer l are described by the vector of biases $\boldsymbol{b}^{[l]} \in \mathbb{R}^{n_l}$. Each neuron i in the hidden layer l, $h_i^{[l]}$, can be described by output vector of the hidden layer as:

$$\boldsymbol{h}^{[l]} = g(\boldsymbol{W}^{[l]}\boldsymbol{h}^{[l-1]} + \boldsymbol{b}^{[l]}) \in \mathbb{R}^{n_l}, \quad for \quad l = 1, 2, 3, ..., L \tag{2.1}$$

where g is an element-wise nonlinear activation function. The input to the network is $\boldsymbol{h}^{[1]} = \boldsymbol{x}^{[1]} \in \mathbb{R}^{n_1}$ and the output of the network is $\boldsymbol{o} = \boldsymbol{h}^{[L]} \in \mathbb{R}^{n_L}$. This computation is repeated throughout the network, therefore overall model complexity is dependant on its structure.

The number of columns in $\boldsymbol{W}$ equals the size of the input vector $\boldsymbol{x}$ and the number of rows equals the size of vectors $\boldsymbol{b}$ and $\boldsymbol{h}$. Thus, the output of each neuron i in the hidden layer l, is

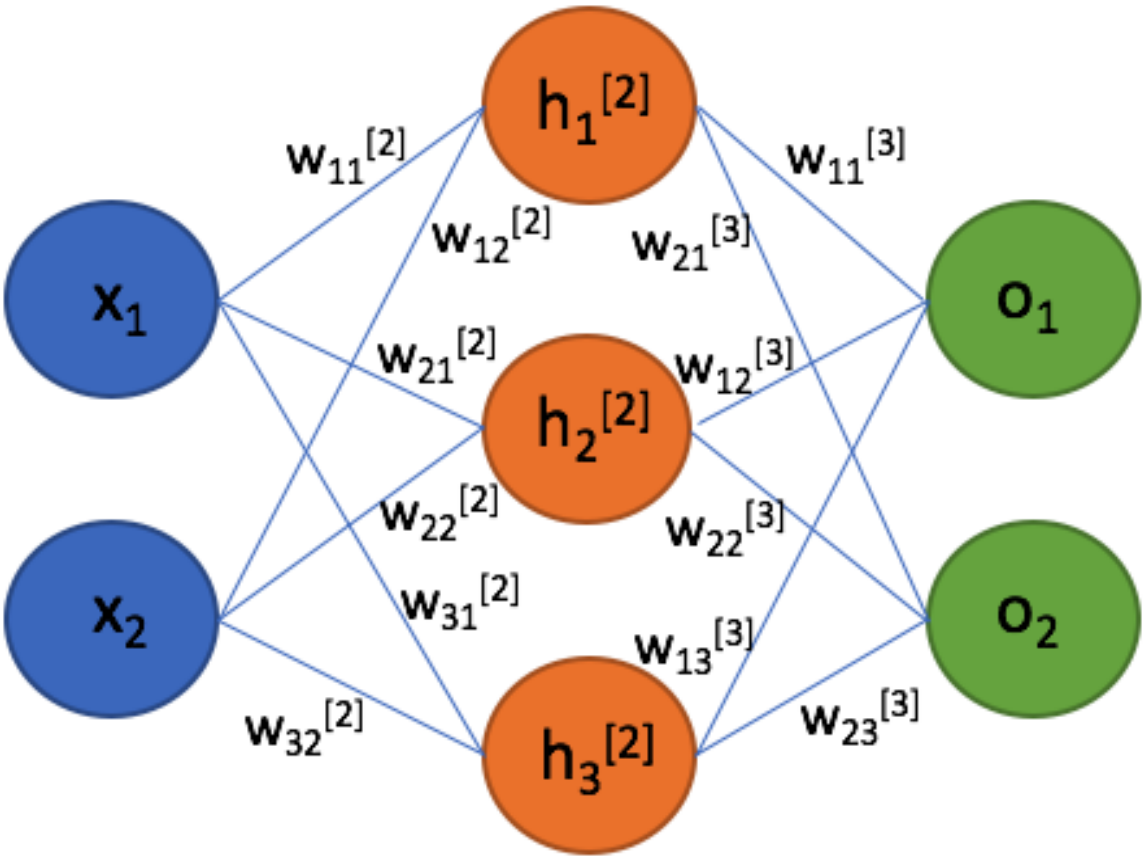


FIGURE 2.1. An example of a basic Neural Network architecture, consisting of an input layer, one hidden layer and an output layer.

computed as:

$$h_i^{[l]} = g(\sum_j w_{ij}^{[l]} h_j^{[l-1]} + b_i^{[l]}), \tag{2.2}$$

where the sum runs over all values in x [61]. For example, consider a simple Neural Network with two input components, one layer of three hidden neurons and two output components as described in Figure 2.1 (Note: biases are not shown in this Figure). The output from the second layer, the hidden layer, has the form:

$$\boldsymbol{h}^{[2]} = g(\boldsymbol{W}^{[2]}\boldsymbol{x} + \boldsymbol{b}^{[2]}) \in \mathbb{R}^2 \tag{2.3}$$

where $\boldsymbol{W}^{[2]} \in \mathbb{R}^{3\times 2}$, $\boldsymbol{x} \in \mathbb{R}^2$ and $\boldsymbol{b}^{[2]} \in \mathbb{R}^2$. The third layer, the output layer, then has the form:

$$\boldsymbol{o} = \boldsymbol{h}^{[3]} = g(\boldsymbol{W}^{[3]} g(\boldsymbol{W}^{[2]}\boldsymbol{x} + \boldsymbol{b}^{[2]}) + \boldsymbol{b}^{[3]}) \in \mathbb{R}^2 \tag{2.4}$$

This expression in Equation 2.4 defines a function $\boldsymbol{o} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ in terms of all its weight and bias parameters for all layers. This completes the forward pass for the inference phase. Based on our resulting outputs we can infer something based on the input. When we want to train the network by learning optimal weight parameters, we then include the backward pass as discussed in the following section.

2.1.2 Training

The non-linearity from the activation functions in DNNs causes the error function to become non-convex [44]. Thus, training DNNs is a non-convex optimization problem. It is an iterative process whereby gradient-based optimizers are used to minimize the error function $E(x)$. We refer to $E(x)$ as the objective function, cost function, loss function or error function interchangeably throughout this thesis. In training, the aim is to compute the partial derivatives of the error function with respect to $w_{ij}^{[l]}$ & $b_i^{[l]}$. To derive these, it is worthwhile to introduce further variables:

$$\boldsymbol{z}^{[l]} = \boldsymbol{W}^{[l]}\boldsymbol{h}^{[l-1]} + \boldsymbol{b}^{[l]} \in \mathbb{R}^n \tag{2.5}$$

Given we have training data consisting of m training batches of examples, we can describe the error function E as:

$$E(\boldsymbol{W}, \boldsymbol{b}) = \frac{1}{m}\sum_{i=1}^{m} e_i(\boldsymbol{W}, \boldsymbol{b}) \tag{2.6}$$

where e is an error function such as mean-squared error, cross-entropy, etc. The iterative method of training a DNN is most commonly done via the backpropagation algorithm and stochastic gradient descent (SGD) [115]. As discussed in Section 2.1.1, training involves a forward path, which feeds in an input to the network and infers the output. There's also a backward path to calculate gradients and update its parameters. SGD is the iterative algorithm which updates the parameters of the model via a gradient of the error function with respect to the parameters. This intends to descend the error function. Thus, at each training iteration, the parameter update aims to move the network function towards a more optimal solution. For this iteration the SGD will typically select a random batch from the training set, corresponding to a function $\hat{e} \in \{e_1, e_2, ..., e_m\}$. For example, weight parameters $\boldsymbol{W}_{t+1}$ are updated with the

following:

$$\boldsymbol{W}_{t+1} = \boldsymbol{W}_t - \eta \frac{\partial \hat{e}}{\partial \boldsymbol{W}_t} \tag{2.7}$$

where η represents the learning rate. Computing the forward path, followed by backpropagation and a weight update constitutes one training iteration. The term $\frac{\partial \hat{e}}{\partial \boldsymbol{W}_t}$ is the partial derivative of $\hat{e}$ with respect to $\boldsymbol{W}_t$ or alternatively it is known as the weight gradient. It tells us how much a change in $\boldsymbol{W}_t$ affects the total error, E. This is calculated using the chain rule for partial derivatives. Similar update equations can be made for the biases. The back propagated loss at each layer is the activation gradient $\boldsymbol{\delta}^{[l]}$. Thus to calcualate the activation gradient and the weight updates, we use:

$$\boldsymbol{\delta}^{[l-1]} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{h}^{[l-1]}} \frac{\partial g}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{\delta}^{[l]}}{\partial g} \tag{2.8}$$

$$\frac{\partial \hat{e}}{\partial \boldsymbol{W}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} \frac{\partial g}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{\delta}^{[l]}}{\partial g} \tag{2.9}$$

We recall from Equation 2.4 that the network output is calculated by computing each layer in order during the forward pass. After doing this, we can compute the error function and calculate the activation and weight gradients by finding the partial derivatives via backpropagation.

2.1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a type of DNN that use a convolution operation in at least one of their layers [79]. The convolution is a linear operation that process input tensors (multidimensional arrays) of (a, p, I) dimensions in a translationally invariant manner [81]. Typically, a and p are spatial dimensions (width and height) and I is the number of channels, also known as Input Feature Maps (IFMs). Processing operations such as convolution, pooling and activation functions are applied in a series of layers in the forward path, each of which transforms the input tensors from dimension $(a, p, I) \rightarrow (a', p', N)$. A convolutional layer produces N output channels, also known as Output Feature Maps (OFMs). Each of the OFMs is formed through a convolution of the input tensor with a $K^{[l]} \times K^{[l]}$ convolutional kernel filter, where typically $K^{[l]}, K^{[l]} \ll a, p$ so it operates on local input regions. For a given convolutional layer, there are $K^{[l]} \times K^{[l]} \times I^{[l]} \times N^{[l]}$ weights.

The output of convolutional layer l, takes as input $S^{[l]}$ images of spatial dimensions $a^{[l]}$ and $p^{[l]}$ and $I^{[l]}$ is the number of neurons. The pixel $h^{[l]}_{I,a,p}$ at location (a, p) for the Ith neuron is calculated as

$$h^{[l]}_{a',p',N} = g(\sum_{s=0}^{S^{[l]}} \sum_{j=0}^{K^{[l]}} \sum_{k=0}^{K^{[l]}} w^{[l]}_{N,s,j,k} \cdot h^{[l-1]}_{I,a+j,p+k}) \,, \tag{2.10}$$

Thus, kernel filters are applied to all pixels of the IFMs, with the result passed into g to compute OFMs. The OFMs then become the IFMs for the next layer. The IFMs to the first layer are from the raw input data, such as the RGB components of an image. For each fully-connected layer, all pixels of the IFMs are multiplied by weights to generate each pixel of the OFM. The element-wise activation function is then once again applied to produce the OFMs.

A 2-dimensional (2D) convolutional layer can be described as matrix multiplication, followed by the elementwise activation function. Convolutional layers form the bottleneck for CNN implementations and this tensor form allows efficient matrix-multiplication libraries to be applied.

Pooling layers are downsamplers of 2D images. Max pooling layers provide a spatial maximum function which divides an input image into small sub-tiles of a given window size and then replaces these with the maximum value in the sub-tile. An average pooling layer is similar, however it finds the average in the sub-tile rather than the maximum.

In Chapters 3, 4, 5 & 6, we use CNNs for image classification and object detection to evaluate our training methods. We also explore various data representations of these models and analyze their respective hardware performances.

2.2 Data Representations

On digital computers, performing operations on different data representations requires different arithmetic. In this section, we describe the two most common types of numerical data representations on digital computers, fixed-point (FX) and floating-point (FP).

2.2.1 Floating-point

The most common form of digital arithmetic in modern day CPUs and GPUs is floating point (FP) [43]. Most commonly these are of the form 32-bits for single-precision and 64-bits for double-precision. The IEEE-754 binary floating point format [65] represents real numbers x by a subset in normal form as:

$$\hat{x} = (-1)^{s_x}(1 + m_x)2^{e_x} \tag{2.11}$$

where $s_x \in \{0, 1\}$ is the sign bit, e_x is an integer representing the exponent of $\hat{x}$ and m_x is the mantissa of $\hat{x}$. Thus a floating point number format can be described as a (s_x, e_x, m_x) tuple. In binary form the representation is $(b^s, b_1^e, b_2^e, ..., b_{B_{e_x}}^e, b_1^m, b_2^m ..., b_{B_{m_x}}^m) \in \{0, 1\}^B$, with B_{e_x} and B_{m_x} being the number of exponent and mantissa bits, respectively. The infinite set of real numbers $\mathbb{R}$ is represented in a computer with $B = 1 + B_{e_x} + B_{m_x}$ bits. The real numbers representable in FP format are known as *exact values*, $\mathbb{F} \subset \mathbb{R}$. Real numbers which aren't representable, known as inexact values I, are approximated in floating point by their nearest exact value.

The main advantage of FP over fixed point is that it has a higher dynamic range. The main drawback however, is that it is less precise and its hardware implementation consumes more area than a fixed point representation with an equivalent bitwidth.

2.2.2 Fixed-point

Exploiting reduced numerical precision for data representation has been very promising for high performance DL implementations. In particular, low-precision neural networks reduce both memory and computational requirements whilst achieving accuracies comparable to floating point [47]. Low-precision representations typically are of the FX format as it provides hardware advantages over FP.

In computer arithmetic, a FX number data representation is able to represent fractional numbers with a fixed number of digits. A set of integers is represented by a fixed number of digits. In contrast, FX is the extension of this which represents a set of rational real numbers

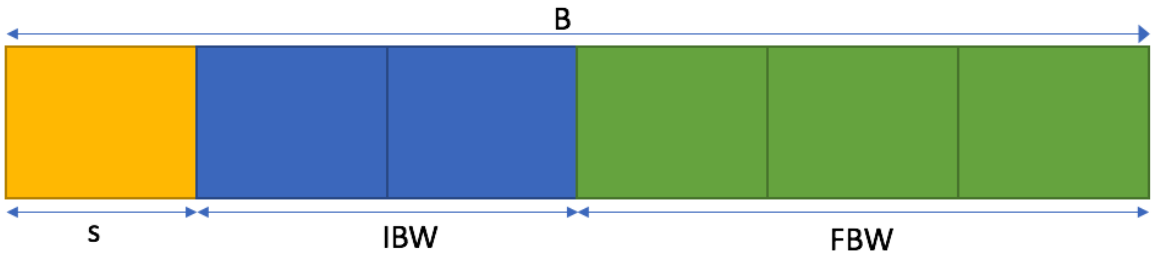


FIGURE 2.2. A two's complement fixed-point number representation

by a fixed number of digits [60]. FX representations can either be signed or unsigned. There are three components to a FX representation: The signed component, the integer component and the fractional component. As shown in Figure 2.2, the total number of bits, B for a signed FX representation is $B = 1 + B_{IBW} + B_{FBW}$ where IWL is the integer bitwidth and FBW is the fractional bitwidth. With this representation, the range of the number is $[-2^{IBW}, 2^{IBW})$, with a step size of 2^{FBW}. Such number formats can be described as a (B, IBW, FBW) whereby the signed-bit is implicit. For example, $(4, 0, 3)$ represents one sign-bit, no integer bits and 3 fractional bits.

The range and step size of an FX number can be increased or decreased via a scaling factor. Thus, for an FX number with scaling factor α, the range becomes $[\alpha \times -2^{IBW}, \alpha \times 2^{IBW})$ and the step size becomes $\alpha \times 2^{FBW}$. The scaling factor is usually a power of 10 for human convenience or a power of 2 for computational efficiency (as then it can be implemented via bit shifts). To add or subtract two FX numbers with scaling factors α and R, if α and R are the same, then it is sufficient to add/subtract the FX numbers and then keep the common scaling factor. If the scaling factors are different however, one of the FX numbers must be scaled so that the scaling factors match before addition/subtraction is executed. Thus, to make the scaling factor R match α, we multiply the FX integer by $\frac{\alpha}{R}$. For multiplication, we can multiply the scaling factors and FX numbers separately. Thus, having common scaling factors amongst numbers can improve computational efficiency as it lowers the number of required multiplications. This is the premise for the representation designed in Chapter 5.

2.3 Quantization Network Training

In this section, we introduce the notion of quantization of neural networks (QNNs). We then discuss bitwise neural networks which are able to replace operations such as multiplies and additions with simple additions/subtractions. These include both binary and ternary representations for weights and/or activations. Following this, we discuss common training methods for learning such networks effectively.

2.3.1 Quantization

Quantization is the process of mapping input values from a large set of possible values to output values in a smaller set [45]. This is typically done via truncation and rounding. We define the possible values for the output as the codebook. If these values are evenly spaced apart, then the quantization is regarded as *uniform quantization*. If they are uneven, the quantization is regarded as *non-uniform*. For example, rounding a real number $x \in \mathbb{R}$ to an integer value is a form of uniform quantization. This is done via a quantization function, $quantize$, as per the following in Equation 2.12:

$$quantize(x) = \gamma \times \left\lfloor \frac{x}{\gamma} + \frac{1}{2} \right\rfloor \tag{2.12}$$

where γ is the step size or distance between between two neighbouring values in the desired codebook. $\lfloor \rfloor$ is the floor function. In this thesis, we explore various forms of quantization for DNNs. In Chapters 3 & 5 we use uniform quantization and in Chapter 6 we use non-uniform quantization.

The quantization function can either be stochastic or deterministic. Stochastic quantization uses probabilistic calculations by sampling a random variable to determine the quantization state [20]. In deterministic quantization however, the quantization state is fully determined from the value being quantized. For example, Equation 2.12 is an example of deterministic quantization. We can quantize x to an integer stochastically using a random variable β via the

following:

$$quantize(x) = \gamma \times \left\lfloor \frac{x}{\gamma} + \beta \right\rfloor \tag{2.13}$$

In this case, $\beta = 0, 1$ and its value can be determined via sampling from a uniform distribution, i.e. $\beta = round(U\ (0, 1))$. In this thesis, we focus on deterministic forms of quantization due to the hardware simplicity over stochastic forms of quantization.

2.3.2 Network Quantization

For QNNs, quantization can be performed as a post-training step [104, 150, 103, 15, 152]. Alternatively, it can be performed during each training iteration [140, 154, 157]. In this instance, the resulting quantized representation after training is used for hardware implementation. In this thesis, the latter methodology which uses quantization during training is the focus. All the networks in this thesis use some form of quantized representation for hardware evaluation.

For quantizing DNNs into lower precisions, the distribution of full precision weights for each layer $\boldsymbol{W}^{[l]}$ are approximated by a function $quantize$, resulting in a quantized weight matrix $\boldsymbol{Q}_l$:

$$\boldsymbol{Q}^{[l]} = quantize(\boldsymbol{W}^{[l]}) \tag{2.14}$$

for $w_{ij}^{[l]} \in \mathbb{R}$ and $q_{ij}^{[l]} \in \mathbb{C}$. The codebook $\mathbb{C} = \left\{c_1, c_2, ..., c_r\right\}$ is a set of all possible values for $q_{ij}^{[l]}$ where $c_i \in \mathbb{R}$ and $i \in \mathbb{R}^+$ represent each codebook value and index respectively. $q_{ij}^{[l]}$ represents each element in the quantized weight matrix $\boldsymbol{Q}^{[l]}$. The codebook values and size is dependant on the data representation used and the bitwidth, respectively. Such data representations can be FX, FP or binary/ternary. In this thesis, we refer to network weight representations with FX or binary/ternary as *low-precision networks*. However, we also refer to binary/ternary networks explicitly as *bitwise networks*. Typically, the smaller the codebook, the harder the network will be to train as the information loss from quantization is larger. Hence we must alter standard training algorithms to ensure efficient learning. In many instances, to enable further hardware advantages, the output of the activations $\boldsymbol{h}^{[l]}$ are also

quantized.

$$\boldsymbol{G}^{[l]} = A(\boldsymbol{h}^{[l]}) \tag{2.15}$$

In this way, the inputs to the next layer are also quantized values, represented by $\boldsymbol{G}^{[l]} \in \mathbb{C}_{\mathbb{A}}$. A represents the quantization function used for the activations. Both quantization of weights and activations are explored throughout this thesis.

2.3.3 Bitwise Networks

Optimizations via compression, quantization and neural network layer explorations have been utilized to reduce complexity and boost performance, e.g. [49, 62]. In particular, quantizing inference networks to very low precision, such as constraining weight representations to binary or ternary values, both reduces memory requirements and enables multiplications to be replaced with the bitwise exclusive NOR operation [18, 127]. This translates to massive reductions in storage requirements and spatial complexity in hardware [126]. Additionally, large power savings and speed gains are achieved when networks can fit in on-chip memory.

2.3.3.1 Binarized Neural Networks

Binarized Neural Networks (BNNs) restrict their weight spaces to have a codebook $\mathbb{C} = \{-1, +1\}$ [20]. To binarize a weight matrix deterministically, we use the element-wise sign function as follows:

$$\boldsymbol{Q}^{[l]} = sign(\boldsymbol{W}^{[l]}) \tag{2.16}$$

with,

$$sign(x) = \begin{cases} 1 & \text{if} \quad |x| \geq 0 \\ \text{-1} & \text{if} \quad x < 0 \end{cases} \tag{2.17}$$

This is the most extreme form of quantization as their are only two possible values, both of which can be represented via 1-bit in hardware. The addition of scaling factors can be useful in recovering the dynamic range of the network without imposing hardware costs. In the most

simplest form, one scaling factor α can be included per layer. In this case, the codebook for a particular layer becomes $\mathbb{C} = \{-\alpha, +\alpha\}$. Another, way to add representational capacity to the network is to ternarize the network by adding 0 to $\mathbb{C}$, however this requires a higher computational cost.

2.3.3.2 Ternary Neural Networks

Ternary Neural Networks (TNNs) restrict their weight spaces to be $\mathbb{C} = \{-1, 0, +1\}$ [82]. To ternarize a each element in a weight matrix deterministically, we use:

$$q_{ij}^{[l]} = \begin{cases} 1 & \text{if} \quad w_{ij}^{[l]} > \eta \\ 0 & \text{if} \quad -\eta \leq w_{ij}^{[l]} \leq \eta \\ -1 & \text{if} \quad w_{ij}^{[l]} < -\eta \end{cases} \tag{2.18}$$

where the quantization threshold, η is a hyperparameter, which can be tuned by the user. By including zero into the codebook, sparsity is naturally introduced into the representation. As similarly discussed in 2.3.3.1, the addition of a scaling factor per layer for TNNs will result in a layer's codebook to be $\mathbb{C} = \{-\alpha, 0, +\alpha\}$.

2.3.4 Straight Through Estimator Learning

As mention in Section 2.1.2, DNN training is an iterative process which has a feedforward path to compute the output and a backpropagation path for learning, which involves calculating gradients and update the network weights. Training of low-precision networks typically involves maintaining a set of single precision floating point weights $\boldsymbol{W}^{[l]}$ which are quantized to a representation $\boldsymbol{Q}^{[l]}$ prior to inference, e.g. [18]. The issue is that a large reduction in precision, leads to large information loss which incurs significant accuracy degradation, especially for complex datasets such as ImageNet [116]. Ideally, we can train networks which have both high prediction capabilities and minimal computational complexity.

During training, heavily quantized functions commonly get trapped in poor local minima, causing accuracy degradation. This becomes problematic especially for BNNs where this effect is greatest. As the quantization functions employed are piecewise and constant, the

FIGURE 2.3. A simple one layer CNN training flow diagram. This shows the difference of how different network layers are related to each other for (a) conventional learning and (b) STE learning with quantized weights and activations

gradients of quantized weights are calculated and applied to update their corresponding full-precision weights [9]. In this way, information from small updates won't be totally lost from the discretization. The quantization neural network training used in this thesis uses the straight through estimator (STE) approach as described in [9]. The STE has been successful as a surrogate for the derivative of a non-differentiable function [64]. This approach allows the non-differentiable function defined in the forward path to use a non-zero surrogate derivative function in the backward path gradient calculations. Thus, an error function E used

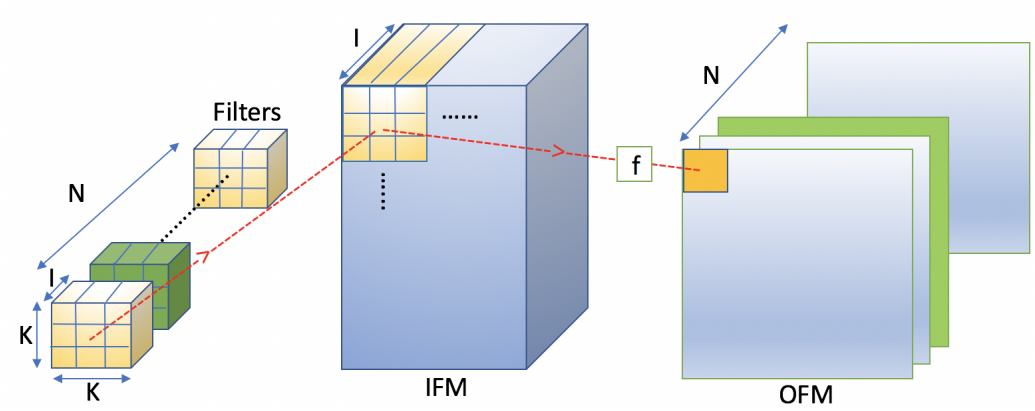


FIGURE 2.4. Convolutional Layer Connectivity and Pruning

to calculate the loss during training (also known as *training loss*), we then allow:

$$\frac{\partial E}{\partial \boldsymbol{Q}^{[l]}} = \frac{\partial E}{\partial \boldsymbol{W}^{[l]}} \tag{2.19}$$

The quantized weights $\boldsymbol{Q}^{[l]}$ are used for inference in the forward path and the floating point weights $\boldsymbol{W}^{[l]}$ are updated in the backward path. This is shown in Figure 5.2, where we illustrate how different network operations are related to each other for STE training in contrast to conventional full-precision training. In Figure 5.2 (a), we see that the real-valued weights and activation outputs are used with the loss to update the real-valued weights. However in Figure 5.2 (b), the quantized weights and activations are used with the loss to update the real-valued weights.

2.4 Pruning

While quantization methods reduce the number of bits for weights and arithmetic operations, pruning methods [5] reduce the total number of weights that must be stored and arithmetic operations that must be performed. For some CNNs, weight pruning allows tensors, which previously required off-chip memories for storage to be eliminated. For FPGAs and ASIC implementations, this enables significant improvements in speed and power.

There are two broad categories of CNN pruning methods: fine-grained pruning and filter pruning. Fine-grained pruning methods modify a trained network by setting the least important set of weights to zero. Weights are considered important if their removal results in accuracy degradation. The issue with this simple approach is that it introduces a sparse matrix representation which incurs overheads due to irregular data access patterns. Filter pruning overcomes this problem by choosing the least important filter/s to keep and pruning out both those filters and their corresponding OFM. This is demonstrated in green in Figure 3.1 and allows the network to maintain the dense matrix format. However, while filter pruning reduces the total number of operations, on hardware designs, it is typically the feature map memory or available resources for processing elements that limits performance. In this work, choosing to prune filters according to these limitations is focused on. It has also been shown that network pruning and quantization are orthogonal and can be applied iteratively together for further redundancies [52], [49].

This has lead to weight pruning methods for low precision networks for applications including compressing binarized neural networks [5], as well as techniques to utilizing sparsity information obtained during training to compress ternary neural networks [32]. Subsequent research has compared varying pruning granularities, highlighting that coarser-grained methods can offer higher memory savings for a given accuracy at the same time as avoiding the costs associated with random sparse matrix representations whereby the nonzero values are randomly situated [90].

2.4.1 Quantization and Pruning for Efficient Hardware Designs

Various research studies have explored how to take advantage of these quantization methods in order to create FPGA-based CNN accelerators with high throughput and low power [100], [151], [108], [129]. A Framework For Fast, Scalable Binarized Neural Network (FINN) [128], demonstrated the advantages of fitting models in on-chip memory. The methods introduced in this work could be applied to improve the performances of many of these implementations for minimal or no accuracy loss. Similarly, there have been efforts to take advantage of pruning

for hardware implementations. This includes pruning to improve performance and efficiency of FPGA architectures for LSTM-based Recurrent Neural Networks [51].

Finally, there has also been some research into tailoring the pruning methodology to hardware. Pruning weights which contribute to the highest data movements and hence power consumption were prioritized in [141]. This demonstrated improvements in energy consumption over traditional techniques. However, the energy estimates in this work were based upon models estimating energy consumption as a function of the number of multiply-accumulate operations and the number of memory accesses required, as opposed to real hardware. Customizing different pruning methodologies to micro-controllers, CPUs and GPUs was proposed in Scalpel [145]. Optimal filter pruning for highly parallel GPU hardware is achieved by removing the largest number of filters. In this work, a different pruning strategy is implemented which focuses on customizing the pruning process for FPGA architectures. FPGAs pose unique considerations over other hardware platforms as the amount of layer unrolling is set by the designer and resources can be arbitrarily allocated. In addition, since the highest performance implementations store intermediate FMs in on-chip BRAM, only considering the model-size is misleading to the overall hardware savings from pruning.

2.5 Applications

In this section, we provide details of the target applications on which our proposed methodologies are evaluated on. We introduce the datasets used for experimentation in this thesis, which are all freely available online. Computer vision continues to be a rapidly growing research area due to its widespread applicability. Its market size in 2019 was 10.6 billion USD and is projected to grow at a compound annual growth rate of 7.6% from 2020-2027 [16]. Within computer vision tasks, we target Image Classification and Object Detection. Image Classification is a foundational component for solving many computer vision based machine learning problems. It can be used on its own for certain applications such as medical imaging analysis. Additionally, many other tasks such as object detection and semantic segmentation initially use models pre-trained on Image Classification tasks and hence efficient models

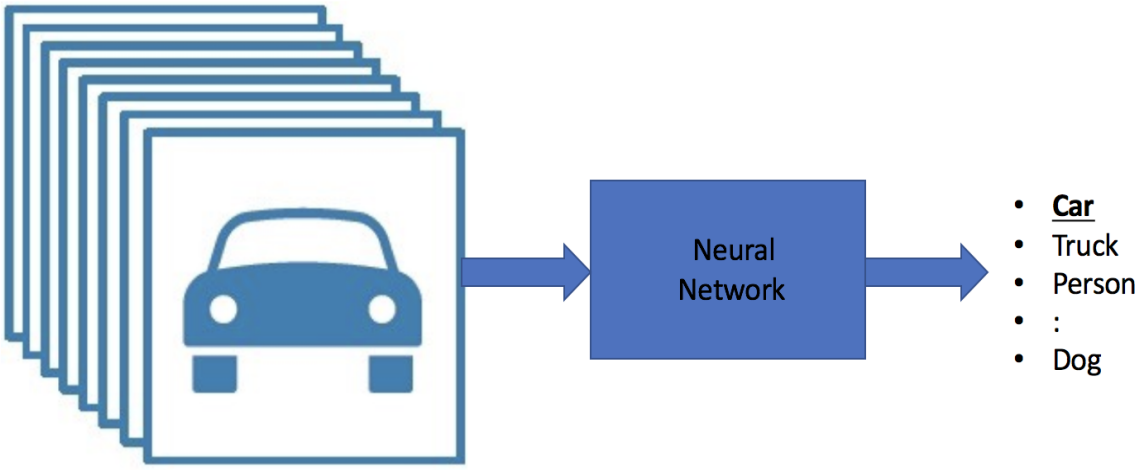


FIGURE 2.5. Image Classification Using Neural Networks

in this realm are critical. We also use object detection because it is becoming increasingly adopted in many applications such as self-driving cars, virtual reality and security camera systems.

2.5.1 Image Classification

Image classification is the task of classifying a digital image or video into a category/class. For a given task, there are n potential classes that an image can be classified to. When wanting to classify thousands or millions of images, it becomes a very tedious task for a human to complete. Thus, with the aid of DNNs, we can rely on computers to complete the task much faster. An illustration of image classification using DNNs is presented in Figure 2.5. In this example, we feed an image into our DNN algorithm and determine that it belongs to the car category. Some of the common image classification benchmark datasets used throughout computer vision research are discussed below.

MNIST: The MNIST dataset [78] is a large dataset consisting of handwritten digits that is commonly used for training various machine learning systems. It consists of 60,000 training images and 10,000 testing images. The images are 28×28 resolution with grey-scale colour. There are 10 different categories which consist of the numbers 0-9.

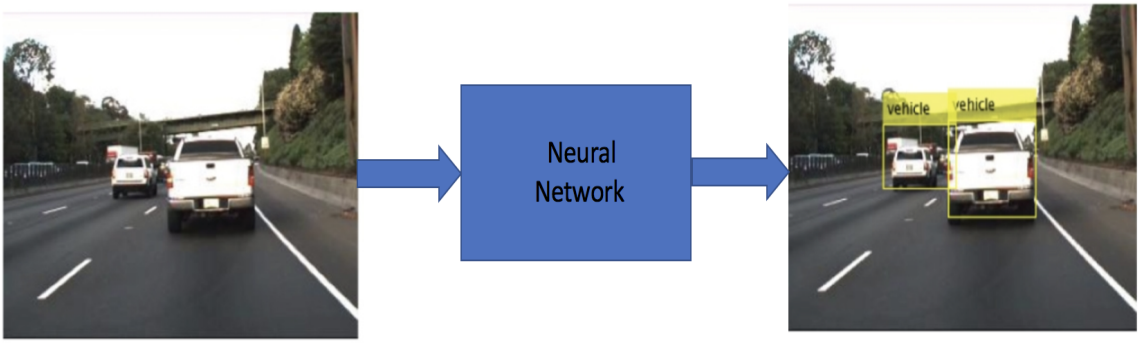


FIGURE 2.6. Object Detection to detect vehicles in an image using DNNs

CIFAR10: The CIFAR10 dataset [67] is benchmark dataset consisting of 50,000 training and 10,000 test images. The images are square 32×32 in Red-Green-Blue (RGB) colour format with 10 different categories. These categories are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. This dataset is widely used to benchmark image classification techniques.

ImageNet: The ImageNet dataset [70] is another benchmark dataset used in computer vision machine learning applications. It consists of over 15 million labelled high resolution images and 22,000 categories. ILSVRC uses a subset of ImageNet which has roughly 1000 images for 1000 categories. It consists of 1.2 million training images, 50,000 validation images and 150,000 testing images. ILSVRC is used through this thesis with images cropped/resized to a 224×224 resolution.

2.5.2 Object Detection

Object Detection is another application area for DNNs in computer vision. Object detection is the task of detecting instances of semantic objects of certain classes/categories in digital images/videos. Other detection tasks such as face detection, pedestrian detection, car detection, etc are all sub-tasks of object detection or can be defined as single class object detection task. Typically boxes are drawn around the objects, known as 'bounding boxes'. An example of detecting vehicles in an image via object detection is illustrated in Figure 2.6.

PASCAL VOC: A common dataset used to evaluate object detection algorithms is the Pattern Analysis Statistical Modelling and Computational Learning (PASCAL VOC) dataset [30] provides standardised image data sets for object detection, semantic segmentation and image classification. In this thesis, it is used for object detection in Chapter 3. It consists of 20 different annotated object classes. The total training/validation data has 11,530 images containing 27,450 annotated objects and 6,929 segmentations.

2.6 Hardware Implementations

In this section, we discuss why we focus on optimizing DNNs for specialized hardware over conventional computing platforms.

2.6.1 Specialized Hardware For DNNs

Specialized hardware, such as FPGAs and ASICs, have the advantage of higher hardware performance and the disadvantage of difficult programmability over conventional computing platforms. For DNN applications whereby personalization is unimportant, the network is not altered regularly. Therefore, the burden of difficult programmability diminishes and the benefits of higher hardware performance are more easily reaped. This makes specialized hardware units are a much higher performant over conventional computing platforms for these types of applications.

FPGA and ASIC implementations have demonstrated improved latency and power efficiency for DNN applications compared with central processing unit (CPU) and graphics processing unit (GPU) technologies, e.g. [146, 127]. This is because conventional computing platforms such as CPUs/GPUs are byte-addressable, making it difficult to take advantage of networks with very low-precision. In this thesis, we use the term *low-precision* to refer to a representation of 1-8 bits. For custom hardware, these limitations do not hold as arithmetic components can be designed for the exact precisions. Therefore, the advantages of low-precision networks can be most highly leveraged on custom hardware, such as FPGAs [107]. In contrast to CPU/GPU technologies, specialized hardware allow customized data paths, enabling higher

degrees of parallelism and less data movement. This design flexibility poses an opportunity to optimize system performance through custom hardware tailored to the application.

2.6.2 FPGAs

To evaluate the representations designed in this thesis, we use FPGA platforms. FPGAs play a vital role in various industries such as Aerospace & Defence, ASIC prototyping, Audio, Automotive, Data Center, Medical Equipment, Wireless Communications and many more [114]. They also offer the option of tightly integrating machine learning to low-level data acquisition hardware such camera sensors and networking hardware for low-latency implementations [119, 99]. FPGAs are a semiconductor computing platform consisting of a number of logic cells that can be interconnected to other logic and input/output (I/O) cells each connected via programmable interconnects [102]. FPGAs allow users to configure these logic cells and programmable interconnects via bit-level programming data, which is stored in memory cells in the FPGA. The complete design is described via a configuration bitstream which specifies the logic and I/O cell functionality, and their interconnection. The main difference between FPGAs and ASICs is that the logic cells on FPGAs can be reconfigured/reprogrammed after manufacturing whereas ASICs have a fixed processing path [74].

A block diagram illustrating a generic fine-grained island-style FPGA is given in Figure 2.7 A logic cell consists of userprogrammable combinatorial elements, with an optional register at the output. They are often implemented as lookup tables (LUTs), flip flops and multiplexors [117] with a small number of inputs. Multiple logic cells are grouped together to form a single unit, known as a *slice*. Multiple slices are also grouped together to form a configurable logic block (CLB). The number of slices in a CLB is dependant on the FPGA family. Modern FPGAs typically consist of configurable logic blocks (CLBs) in a "sea" of programmable interconnects. This is shown in Figure 2.7. Other main components found on an FPGA are Digital Signal Processors (DSPs) and Block Random Access Memories (BRAMs). DSPs are hardened blocks which operate at much higher frequencies and consume less resources than the equivalent circuit on programmable logic. They are particularly useful for higher

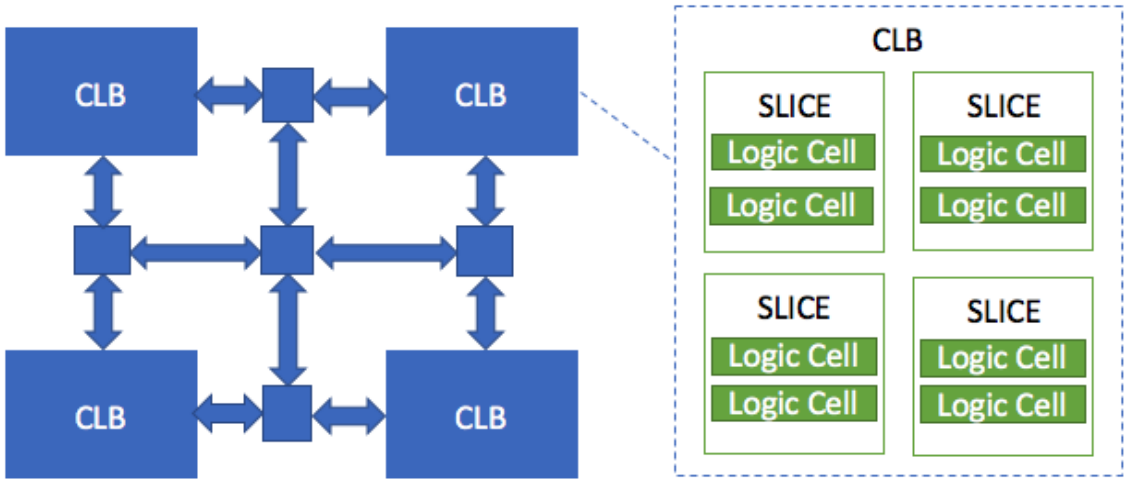


FIGURE 2.7. An array of CLBs whereby each CLB contains four slices.

precision computations. They consist of a multiplier and additionally other components such as pre-adders, adders and accumulators. BRAMs are situated through the FPGA and used as buffers for data storage. They vary in size and consist of one or two ports (known as dual port BRAMs). All these components mentioned consume the majority of the area on an FPGA. Hence, throughout this thesis we evaluate resource usage results in terms of Flip-Flops, LUTs, DSPs and BRAMs.

2.6.3 FPGAs For DNN implementations

Although quantizing networks reduces its representational capacity, for a wide-variety of networks, it has been shown that low-precisions (1-8 bits) for weights and/or activations can be achieved with negligible or no accuracy loss. For bitwise networks, Multiply and Accumulate operations (MACs) are replaced by bit operations. For example, Figure 2.8 shows average resource usage on FPGA hardware to implement a MAC operation under different precisions, which scales quadratically with the multiplier size at $\mathcal{O}(k^2)$ where k is the number of bits[1].

As shown, no high precision multipliers (known as Digital Signal Processors (DSPs) on an FPGA) are required for precision less than or equal to ternary weights and 8-bit activations.

[1]Results are obtained from instantiating MAC modules using Vivado

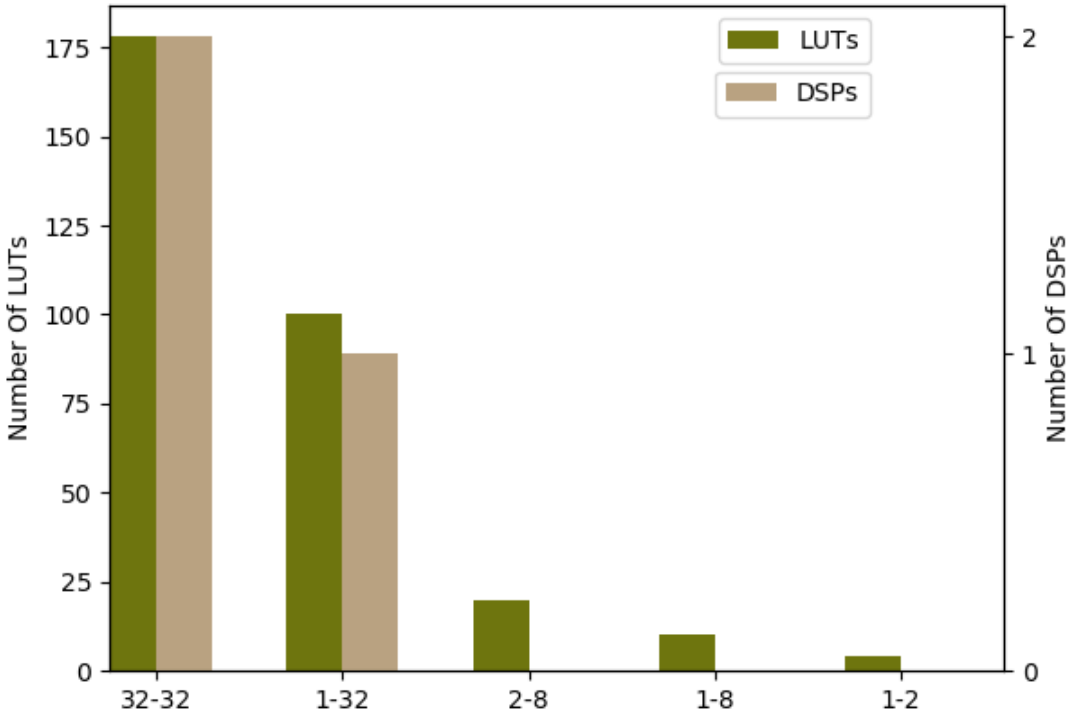


FIGURE 2.8. The average cost per MAC operation on an FPGA device for different bitwidths (weight-activation)

Furthermore, the logic element (known as Look-Up Tables (LUTs) on an FPGA) requirement reduces proportionally with both weight and activation precisions. Additionally, the storage requirements for both weights and activations is reduced by $8 - 32\times$. This significantly improves the network's ability to fit in on-chip memory and constrained hardware environments, and broadens the applicability of DNNs.

2.6.4 Hardware Exploration Summary

The key contributions of this thesis surround the simplification of algorithmic representations of DNN algorithms. To validate our results, we use hardware explorations as proofs of concept to demonstrate the potential benefits and guide the reader on the practicality of this work.

In Chapter 3 we discuss memory savings achieved by calculating the storage requirements of the network. Additionally, in Chapter 3, we report throughput in terms of operations/second via a hardware cost model based on a design in High Level Synthesis (HLS) [91]. In Chapter 4, we report throughput using a physical FPGA implementation based on HLS in terms of Frames Per Second (FPS) and FPS/Area. In Chapter 5, the FPGA resource consumption is presented to demonstrate area impacts from the proposed representation. The design was also done in HLS. Finally, in Chapter 6, we present FPGA resource consumption results based on Place and Route results to deomonstrate area savings. We also present numbers for estimated power. These numbers are all based on a hardware design produced via the HDL language, VHDL.

The area savings in all Chapters is considered to be an improvement in hardware cost. This reduction in resource requirements will likely translate into implementations with improved latency, throughput and/or power consumption. This is because, if we can fit more of the DNN computation on a smaller portion of area. The implementation can then be scaled by either improving the parallelism of the computation, which leads to improved throughput, or improve the frequency of the implementation, leading to a latency reduction. It also potentially allows certain designs to fit on smaller devices, which can lead to reductions in power.

In Chapters 3, 4 & 5, the reduction in computation from pruning and quantization will either reduce silicon area in hardware or reduce the time to compute a layer/network. This will generalize to most hardware platforms. For Chapter 6, the particular multipliers are not guaranteed for other platforms as they are tailored to a specific FPGA family. However similar types of multipliers have been designed for other FPGAs as seen in [72]. These multipliers require different coefficient sets, which would not require any changes to the training methodology.

Due to the research purpose of this work, some of these results are not explicitly presented with physical FPGA implementations. This is because the hardware explorations are used as a proof of concept. This may suggest that these improvements are not guaranteed, however, substantial hardware improvements are demonstrated in most cases which gives the reader confidence that improvements for physical implementations can be obtained.

Physical FPGA implementations, involving a bit stream being downloaded to the board were not completed for Chapters 3, 5 & 6 because physical the FPGA was not available at the time. Additionally, the need to source one was not considered as it was beyond the scope of the research. The measurements from the Vivado tools were sufficient from this research perspective. In the case of using these designs in real-world products, physical implementations would need to be explored.

2.6.5 FPGA Platforms

For Chapters 3, 4 & 6, the Xilinx Kintex Ultrascale 115 (KU115) was used to evaluate the results. This is a large FPGA which consists of ample resources (663K LUTS, 5.5k DSPs and 4.3k BRAMS). This allowed for detailed resource explorations for different scales of parallelism for networks and their proposed representations. In addition to this, in Chapter 5, the Xilinx Zynq Ultrascale+ MPSoc (ZU3) was used. This board targets embedded applications. This is significantly smaller to the KU115 (70.6k LUTs, 360 DSPs and 432 BRAMS). We used this smaller board as we only wanted to explore resources for a single network layer and demonstrate its suitability in the embedded application space.

2.7 Summary

In this chapter, we introduced the theory for DNNs. Many of the key concepts and terminologies were defined for common DNN models. The methodology for training DNNs using conventional methods was explained and equations were introduced. Additionally, we examine common data representations for DNNs, such as fixed-point, floating-point and bitwise neural networks. These representations are used throughout Chapters 3, 4, 5, 6. We also discuss the STE learning method used in training for these representations in these Chapters. Also, FPGA hardware is defined and discussed. Finally, we present the applications and datasets used for evaluating our methods throughout the thesis.

In the following Chapters, we present different methods which explore different combinations of computation type, precision size and arithmetic type as presented in Table 2.1. We

	CH 3	CH 4	CH 5	CH 6
Computation	Sparse	Dense	Dense	Dense
Precision	Bitwise	Bitwise	Low-precision	Low-precision
Arithmetic	Uniform	Uniform	Uniform	Non-Uniform

TABLE 2.1. Categorizing each technique in terms of computation, precision and arithmetic type

design representations requiring sparse matrix computation in Chapter 3 and dense matrix computation in all other chapters. In Chapters 3 & 4, we use bitwise networks and in Chapters 5 & 6 we use higher precision FX representations. Notably, in Chapter 5, we describe how the network can be computed as a bitwise network due to the ordered scaling factors. However, the resulting codebook still requires a higher precision. In Chapter 6, we explore non-uniform quantization, whilst all other techniques use uniform quantization.

CHAPTER 3

Using Sparsity To Enhance Bitwise Network Performance

A sparse representation of data has only few non-zero elements and a majority strictly zero elements. Sparse neural network representations can be created by making zero-valued weight parameters and/or activations. This methodology is known as *Pruning*. As discussed in Section 2.4, pruning has been demonstrated as an effective regularization method for improving network generalization of DNNs [52, 48] and also for compression [49]. The latter can translate to reductions in memory and computational complexity in hardware. In this section, a training methodology for inducing sparsity is discussed. The technique *Compressing Ternary Deep Neural Networks Using Sparsity-Induced Regularization* is introduced. Using bitwise networks is the focus. This is because at this precision, significantly higher performance custom hardware implementations are possible as most/all operations are bitwise.

Specifically, the method involves producing sparse, ternary neural networks. It incorporates hardware implementation costs during training to achieve significant model compression for inference. Training involves three stages: network training using L2 regularization and a quantization threshold regularizer, quantization pruning, and finally retraining. Resulting networks achieve improved accuracy, reduced memory footprint and reduced computational complexity compared with conventional methods, on MNIST and CIFAR10 datasets. The networks are up to 98% sparse and 2 & 11 times smaller than equivalent bitwise models, translating to significant resource and speed benefits for hardware implementations. In this method, individual weights are pruned out based on their magnitudes, meaning the resulting sparsity patterns are random.

3.1 Compressing Bitwise Using Sparsity-Induced Regularization

As discussed in Section 2.3.2 networks can constrain either weights alone or weights and activations via quantization, leading to extremely efficient hardware implementations. In this methodology, the inherent sparsity of TNNs is utilized whilst maintaining the advantages of multiplierless computations. Similar Convolutional Neural Networks (CNNs) to [133] are used for CIFAR10 classification and achieve similar accuracies, although their network has a full precision 1st layer compared to the ternary weights used in our networks. Regularization techniques and reduced precision weight representations have been extensively studied for compression, acceleration and power minimization. Many efforts have concentrated on building efficient computational structures from floating point networks through sparse weight representations and quantization [49, 52, 139, 147, 121, 7, 57, 89, 85, 149, 143, 77]. However, such networks still require fixed-point multiply-accumulate operations which limits power savings and speed.

Instead of considering sparsity and reduced precision separately, sparse TNNs explore both. Pruning the fully connected layers of bitwise networks was proposed in [5] to reduce the number of model parameters for efficient hardware implementations. In our work, all layers are pruned and the focus is on inference acceleration. Additionally all networks are in low-precision (1-8 bit representations) for weight and/or activations. With recent breakthroughs in low-precision deep learning, specialized hardware solutions have been increasingly investigated. FINN implements scalable BNN accelerators on FPGAs [128] and we use this framework to explore hardware performance advantages of sparse TNNs.

We propose a three-stage training approach for TNNs which is able to reduce hardware costs for inference. Firstly, the network is trained using L2 regularization and a quantization threshold regularizer, secondly we use quantization pruning whereby the sparsity pruning threshold is the same as the quantization threshold and thirdly the network is retrained. During training, the network learns in a sparse environment. This has significant benefits as the

sensitivity of the weights to sparsity regularizers can be determined after the model is initially trained. The contributions of this method are thus as follows:

- The first reported low-precision training method which minimizes hardware costs as part of the objective function. This uses a quantization threshold regularizer and L2 regularization to encourage sparsity during training.
- A layer-based quantization pruning technique which utilizes sparsity information obtained during training.
- A quantitative comparison of the proposed sparse TNN with state-of-the-art multiplierless networks in terms of accuracy, memory footprint, computational requirements and hardware implementation costs.

Together these techniques achieve between 2 and 11x compression. For memory-bound hardware architectures, this would directly translate into speed-up.

3.2 Sparse TNN Training

The key idea in this work is to introduce sparsity in TNN weight representations. TNN training consists of real-valued weight parameters, w_r, which are quantized deterministically to w_q using a quantization threshold hyperparameter, η. Thus, duplicating equation 2.18, the ternary weights in this Chapter are calculated as:

$$w_q = \begin{cases} 1 & \text{if} \quad w_r > \eta \\ 0 & \text{if} \quad -\eta \leq w_r \leq \eta \\ -1 & \text{if} \quad w_r < -\eta \end{cases} \tag{3.1}$$

For the forward path, w_q is computed and used for inference. For the backward path, the gradients are computed with w_q and parameter updates are then applied to w_r. In training DNNs, generally many values for w_r can achieve the same training loss. Regularization techniques incorporate a preference for certain weight representations with the aim of improving generalization. In this method, representations are chosen that minimize the number of

nonzero parameters. This is done by inducing sparsity and with the aim of improving hardware performance. The scheme considers the hardware costs not only during the fine-tuning stage, but also during the initial training process.

3.2.1 Quantization Threshold

The deterministic quantization described in Equation 3.1 partitions the w_r weight space via a threshold η. Typically, different values for η are used based on different assumptions made on w_r. For example, to uniformly partition the weight space, $\eta = 0.33$ [5] or to minimize quantization error, $\eta = 0.5$. Generally, a higher η will lead to more sparsity as more of the w_r weight space is quantized to 0. As such, in this methodology, η is increased to make 0's consume a large portion of the weight space (up to 95%) which induces a similar sparsity effect to L1 regularization. However, L1 regularization has a continous shrinkage effect which induces sparsity amongst all w_r but not necessarily w_q. Increasing the threshold on the other hand, induces sparsity explicitly on w_q. Parameter updates for w_r are either penalized or rewarded based purely on the gradients. An example of the effect of this scheme is shown in Figure 3.1. By partitioning the weight space such that $\eta = 0.9$, the network is initialized with high sparsity. This takes much longer to converge than training with a uniformly distributed weight space. Initially the validation error is much larger for $\eta = 0.9$, however over time, the network converges to achieve the same validation error as $\eta = 0.33$.

3.2.2 L2 Regularization

In conventional TNN training, the cost function C can be represented as the average loss L_i over all training examples n:

$$C(w_q) = \frac{1}{n} \sum_{i=1}^{n} L_i(w_q) \tag{3.2}$$

L2 regularization adds the squared magnitude of a coefficient as a penalty term to the loss function [17]. It has the property of penalizing large $\mid w_r \mid$ more and small $\mid w_r \mid$ less. This tends to generate a more diffused set of weights which have a smaller range. L2 regularization

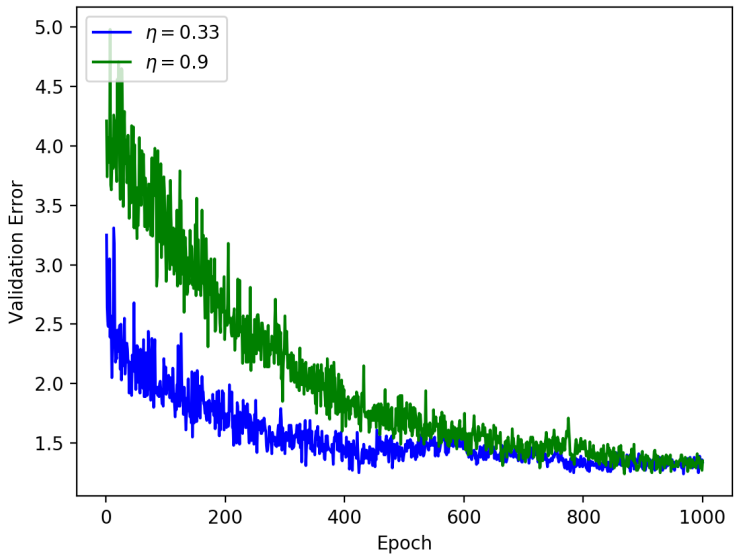


FIGURE 3.1. Validation Error Convergence on MNIST. Comparing training with $\eta = 0.33$ and $\eta = 0.9$

is added as a function of the quantized weights, directly into the cost function to penalize nonzeros and induce sparsity:

$$C(w_q) = \underbrace{\frac{1}{n}\sum_{i=1}^{n} L_i(w_q)}_{\text{data loss}} + \underbrace{\lambda R(w_q)}_{\text{regularisation loss}} \tag{3.3}$$

where the regularization term is the quadratic penalty over all parameters, w_q is the quantized/ternary weights,

$$R(w_q) = \frac{1}{2} w_q^2 \tag{3.4}$$

and the gradient contribution from the regularization term becomes:

$$\frac{dC(w_q)}{dR(w_q)} = \lambda w_q \tag{3.5}$$

where λ is the regularization strength hyperparameter. With L2 regularization, each epoch becomes a greedy search to reduce hardware costs as only the corresponding w_r for each nonzero in w_q is penalized by λ. From (5) and (1) it is evident the regularization term will only affect the corresponding parameter updates on w_r for nonzero w_q. This is desirable when

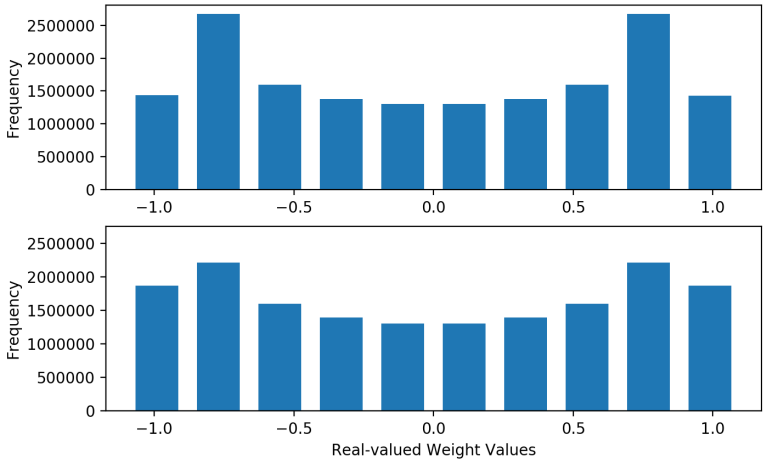


FIGURE 3.2. Weight distribution for w_r for MLP Layer in MNIST training. With L2 regularization (top) and without (bottom). Both are for TNNs with $\eta = 0.9$.

used in conjunction with a large η as peaky weights (weights with values close to -1 and 1 in this case) are more likely to be pulled below the threshold for a given regularization strength. Also, it avoids L2 regularization from continually penalizing weights, making them stuck at low values. This allows for weight values which are penalized in earlier training epochs, to then be more easily recovered through parameter updates if required later in training. As seen in Figure 3.2, under L2 regularization the frequency shrinks for weight values closer to -1 and 1. Thus, more values of w_r are quantized to zero under the ternary scheme. It is also evident that many weights clump around values closer to the threshold of 0.9.

3.2.3 Quantization Pruning

Training with L2 regularization and the quantization threshold achieves a certain sparsity before accuracy starts to degrade. This is addressed via our re-training method *quantization pruning*. In quantization pruning, the subset in w_r which have been quantized to 0 after the initial training phase are all fixed to 0 for the re-training phase. The network is then retrained,

whereby a masking vector w_m is applied to fixate this subset of w_r to zero:

$$w_m = \begin{cases} 1 & \text{if} \quad w_r < -\sigma \\ 0 & \text{if} \quad -\sigma \leq w_r \leq \sigma \\ 1 & \text{if} \quad w_r > \sigma \end{cases} \tag{3.6}$$

In the re-training phase, there is also opportunity for more w_r to be pruned also. In doing so, quantization pruning utilizes weight sensitivity information from the initial training phase to determine which weights to prune for each layer. This is different to sensitivity pruning [52]. In sensitivity pruning, the sparsity hyperparameter σ is chosen by setting different sparsities for different layers. Depending on the type and order of layer, they have a different sensitivity to pruning. In our method, by forcing sparsity through regularization during training, the gradient-based optimization converges to a solution which determines the inherent sensitivity of each layer to sparsity. The ratio of zeros in each layer is utilized from the first training phase by pruning only w_r below or equal to the quantization threshold.

$$\sigma \leq \eta \tag{3.7}$$

Allowing w_{r_1} to be the resulting weights from the initial training phase, weight initialization for retraining then becomes the elementwise multiplication w_{r_2}.

$$w_{r_2} = w_{r_1} \odot w_m \tag{3.8}$$

For retraining, w_{r_2} is updated but the pruned weights are fixed at zero. Also, the threshold is set to the same value as in the initial training phase.

3.2.4 Weight Representations

In practical implementations of TNNs, real numbers cannot be represented, so the quantized representation, w_q, must be used. Due to the high data regularity of zero-valued weights, storing all the ternary weights as 2-bits is not necessary. Instead, two compression methods are used. The first is Run Length Encoding (RLE), which stores only the index differences between each nonzero and also a sign bit which defines the type of operation. In the second method, we use Huffman Coding (HC) on the index differences to assign variable length

codewords whereby the most frequently occuring indexes are represented with shorter length codes and vice versa. HC has higher complexity for its decoder implementation and a higher compression rate than RLE.

3.2.5 Algorithm

Algorithm 1 describes the compression process and consists of four parts. Part 1) represents typical TNN training and additionally requires hyperparameters λ and η to be set. Algorithm 2 represents the training process experienced in Parts 1) and 3). In Part 2) we use quantization pruning to calculate the masking vector and this is used for retraining in Part 3). After the network is trained, the real-valued weights are discarded and the quantized weights are encoded for Part 4). Outputs and inputs for each layer are represented by y and x respectively; b is the bias term (if applicable); L is the learning rate; and CGU is compute gradient updates.

Algorithm 1

1. Train
Set λ and η for sparsity requirements and implement Algorithm 2
2. Quantization Pruning
Compute w_m **with** $\sigma = \eta$
3. Retrain
Keep λ and η the same
Repeat Step 1. **with** $w_{r_2} = w_{r_1} \odot w_m$ and λ, η
4. Encode
Apply HC or RLE on resuting w_q

Algorithm 2

-Forward Pass:
for each weight layer p **do**
  $w_{q_p} = Q(w_{r_{1_p}})$ **with threshold** η
end for
for each layer i in range(1,N) **do**
  Compute y_i **with** w_q, x_i
end for
-Backward Pass:
Compute cost: $C(w_q)$ **with** y_N, λ
for each weight layer j **do**
  CGU: $g_1 = \frac{dC(w_{q_j})}{dw_{q_j}} + \lambda w_{q_j}$
  CGU: $g_2 = \frac{dC(w_{q_j})}{db_j}$
  Updates: $w_{r_{1_j}} = w_{r_{1_j}} - Lg_1$
  $b_j = b_j - Lg_2$
end for

3.3 Sparsity and Networks

The training methods are evaluated on two image classification benchmarks, MNIST and CIFAR10. We compare against other bitwise network results from BinaryConnect [21] and BinaryNet [111]. BinaryConnect uses floating point ReLu activation functions and BinaryNet uses binary activation functions. Their results are shown in Tables 3.2 & 3.3 as 'model-a-b' where a is the weight bitwidth and b is the activation bitwidth (bitwidth = 32 is for floating point, bitwidth = 1 is for binary and bitwidth = 2 is for ternary equivalents of these architectures with a uniformally distributed weight space). The results are reported as TNN with resulting sizes represented as x/y which represents the sizes after encoding in RLE/HC respectively. For all results, the number of weight parameters (Params) in millions is reported, percentage of zero-valued parameters, the error-rate and size of the network in megabytes (MB). In all the models, we use only one pruning iteration except for the MLP with floating point activations for which two iterations were used.

3.3.1 MNIST

The networks used for classification consist of 3 hidden layers of 4096 neurons for the network with binary activations and 1024 neurons for the network with floating point activations. The network is trained for 1000 epochs and the network which produces the best validation error rate is chosen. The effect of quantization pruning is first analyzed on the MNIST dataset for different threshold settings. No L2 regularization is used in these numbers in order to focus on the effect of different pruning thresholds. Setting a higher threshold allows for more aggressive pruning at the threshold.

The results are displayed in Table 3.1 whereby the error-rate is the percentage of incorrect classifications on the test set. Although other threshold settings achieve similar error-rates, $\eta = \sigma = 0.9$ achieves significantly improved sparsity. At this setting, 80% of weights can be pruned. As $\eta = \sigma$, only the subset of w_{r_1} which are quantized to 0 is pruned away. When $\sigma > \eta$, some of the nonzero w_{r_1} are also pruned away. It is evident that pruning nonzeros impinges on the network accuracy and hence the quantization threshold is an

η	σ	Pruned	Error-rate	Nonzeros
0.9	0.95	91%	1.08	1,220,468
0.9	**0.9**	**80%**	**0.92**	**1,863,521**
0.9	0.65	50%	0.96	3,826,912
0.7	0.8	76%	0.98	3,595,898
0.7	0.7	64%	0.91	5,243, 764
0.7	0.58	50%	0.92	6,863,798
0.5	0.9	89%	1.14	2,448,073
0.5	0.75	74%	1.04	5,416,539
0.5	0.5	74%	0.98	10,396,476

TABLE 3.1. Quantization Pruning for TNN (Binary Activations) on MNIST, without L2 regularization.

Model	Params	Zeros	Error-rate	Size (MB)
MLP-2-1	36.4	54%	0.92	9.12
MLP-1-1	36.4	0%	0.96	4.56
TNN	36.4	97.6%	0.93	0.83/1.59
MLP-2-32	2.91	34%	1.23	0.72
MLP-1-32	2.91	0%	1.29	0.36
TNN	2.91	92.8%	1.22	0.07/0.11

TABLE 3.2. Classification accuracies for Sparse TNNs for MLPs on MNIST with L2 regularization and pruning (Right)

effective indicator for which weights can be pruned. Pruning at a lower sparsity threshold maintains accuracy benefits, although results in more nonzeros. To make the network more sparse, it would require several pruning iterations. This could take days/weeks as each training iteration takes days itself. The results are displayed in Table 3.2. Using the network with binary activations produces up to 97.6% sparsity and over $5\times$ compression over its binarized network (BNet) with better accuracy and approximately $11\times$ its ternary network with the same accuracy. The network with a floating point activation function, achieves 92.8% sparsity and $3.5\times$ compression over its binarized equivalent network. For these networks, we use $\eta = 0.9$

Model	Params	Zeros	Error-rate	Size (MB)
VGG-2-1	14.02	65%	11.2	3.52
VGG-1-1	14.02	0%	11.4	1.76
TNN	14.02	92.3%	10.8	0.88/1.05
VGG-2-32	14.02	35%	9.2	3.52
VGG-1-32	14.02	0%	9.9	1.76
TNN	14.02	90.1%	9.6	0.96/1.22

TABLE 3.3. Classification accuracies for Sparse TNNs for CNNs on CIFAR10 with L2 regularization and pruning.

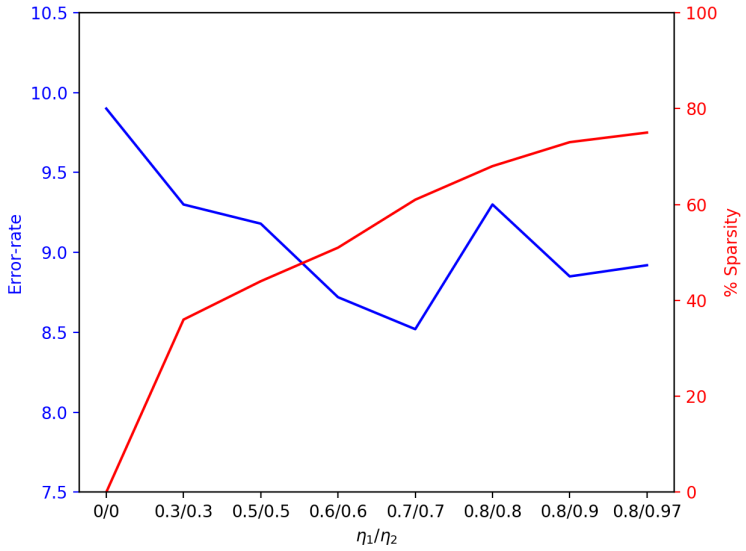


FIGURE 3.3. Accuracy vs Sparsity for CIFAR10. We vary the quantization threshold regularizer for convolutional layers, η_1, and fully-connected layers, η_2. No quntization pruning is implemented.

3.3.2 CIFAR10

We use a VGG-derivative architecture inspired by BinaryConnect [21]. From Table 3.3, it is evident that there is an improvement in accuracy and/or compression for both networks in contrast to their binarized and ternary equivalents. The convolutional layers are less robust to higher values of the quantization threshold regularizer η in Equation 3.1. Hence, a lower value for the convolutional layers $\eta_1 = 0.8$ is set and a higher value for the fully connected layer $\eta_2 = 0.9$. The accuracy and sparsity relationship is shown in Figure 3.3. For varying thresholds, it is shown that threshold regularization improves accuracy. The leftmost point

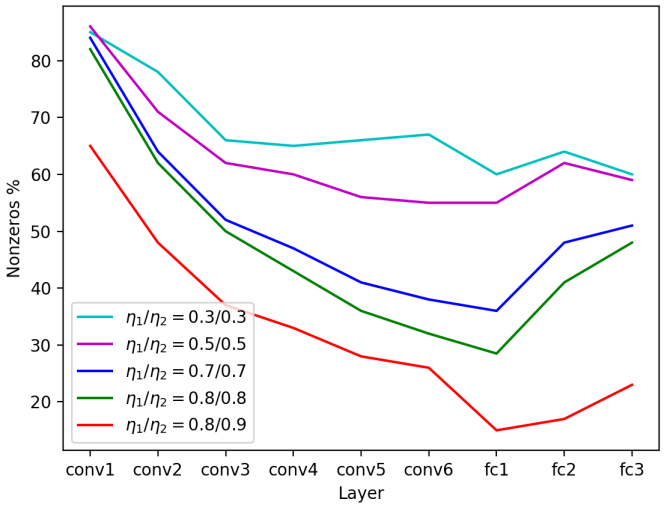


FIGURE 3.4. Per Layer Sparsity for different quantization threshold regularization on CIFAR10. No L2 regularization or pruning is implemented

is the fully dense binarized network where $\eta = 0$ and as the threshold regularization is introduced, the error-rate drops by up to 1.4% as the sparsity is increased. In Figure 3.4, the percentage of nonzeros is plotted for each layer in the CNN for varying values of the threshold regularizer. By increasing the threshold, the robustness of each layer under sparsity becomes more prominent. For most of the networks, the first two convolutional layers are the most sensitive to sparsity and consist of around 80% nonzeros and the last convolutional and first fully connected layers are the least sensitive. These are similar conclusions to [52] who pruned each layer independently to determine their sparsity sensitivity. In this case, the network learns these sensitivities by training in sparse environments. This is advantageous as efficient sparsity parameters are determined for any layer type or order and don't require a hyperparameter search. Varying the threshold provides sensitivity information for the sparsity of each layer and quantization pruning takes advantage of this by pruning each layer according to the threshold and hence these ratios.

3.4 Hardware Implications of Sparse TNNs

In this section, the hardware implications of implementing TNNs with unstructured sparse data representations are explored. Storing the weights in a compressed format requires a decompressor which incurs some overheads. A fully parallel architecture would require a decompressor for every weight in the convolution or fully connected layer and decompressors in this case would consume significant amounts of resources. For sparse TNNs, data reuse patterns within convolution layers and fully connected layers (when batching is applied) can be utilized to increase the ratio of processing elements (PEs) to decoders. When the sparsity of these networks is taken into consideration, the number of *effective operations* (discussed later in this Section) increases the potential accuracy of TNNs to values well beyond those of BNNs. For conventional computing platforms (e.g., CPUs and GPUs), the main benefit of sparsity and compression is the increase in operational intensity that is achieved for a particular layer. Sequential processors, (such as CPUs) will also be able to benefit from the reduction in required operations per layer, as a result of the high sparsity of TNNs. For parallel processors, (such as GPUs, FPGAs and ASICs) it is a lot more difficult to take advantage of this benefit due to the irregular data access patterns. A hardware decompressor described, along with a corresponding parallel architecture suitable for FPGAs, such as the Xilinx Kintex Ultrascale board used, and the potential performance of that architecture in terms of effective operations per second.

3.4.1 Hardware Decompressor

The proposed hardware decompressor iterates through a list of weights, stored in a sign–magnitude form in on-chip memory. In each cycle, the hardware decompressor outputs the complement of the sign bit to represent the weight value and adds the magnitude value to an internal counter, which is used to generate the address of the value to be accessed from the input vector. The RLE decoder consists of a counter which controls the address of the input to feed into the PE for computation. The resource and performance estimates given by Vivado

HLS [105] v2017.1 of the resultant hardware description are that the design can produce an address and a weight every cycle at 250 MHz while using 112 LUT resources on the FPGA.

3.4.2 A Sparse TNN Accelerator

Two types of bitwise networks are described in this work: 1) networks with binary activations (VGG/MLP-1/2-1); and 2) networks with floating point activations (VGG/MLP-1/2-32). For all networks, the predominant calculations for inference are multiply-accumulate operations (MACs). For type 1) networks, this corresponds to XNOR-popcount operations [111], where a popcount is the number of set bits in a word. For type 2) networks, this corresponds to an XNOR operation on the sign bit of a floating point value, followed by a floating point accumulate.

3.4.3 Accelerator Architecture

The proposed accelerator architecture is based on that generated by FINN [128]. In particular, a design is proposed which has processing engines with a similar datapath to FINN. Similar PEs are discussed in more detail in Section 4.6.2 with Figure 4.5. To compute the input-weight matrix in specialized hardware implementations, typically a series or array of PEs are used to receive input data and a weight value to perform the multiply accumulate operations, as required for the datatype. For the compression format described in Section 3.2.4, these implementations require a decompressor between the weight matrix and the PE as represented in Figure 3.5. For type 1), with binary activations, resource usage is estimated on the roofline given by [128], which is reported to have an average cost of 5 LUTs for both an XNOR and popcount operation.[1] For type 2), with floating point activations, resource usage is estimated by instantiating a Xilinx Floating Point 7.1 IP core addition module. The peak throughput numbers are what can be achieved if 70% of the LUTs or 100% of the DSPs are used on the target device, a Xilinx KU115 running at 250 MHz. These are 46.4 TOPs for type 1) and 1.3 TOPs for type 2). The total KU115 resources are 663k LUTs and 5,520 DSPs.

[1] FINN quotes 2.5 LUTs per operation, which is multiplied by 2 to get LUTs / per MAC.

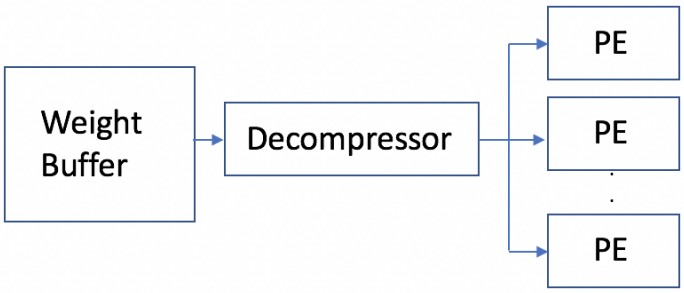


FIGURE 3.5. Diagram of Decompressor Feeding Multiple Processing Elements with Data Reuse

3.4.4 Exploiting Sparsity Through Data Reuse

Convolutional layers require many operations on different input pixels to the same weight value. Hence, data re-use optimizations can be utilized [39] to instantiate a decompressor for a specific weight and calculate several MAC operations on different input pixels. This greatly reduces the average resource usage of the decompressor per operation. Similar optimizations can be utilized for the fully connected layers, whereby batching can be applied to allow a single weight to calculate several MAC operations across multiple input vectors.

Let us introduce a data re-use factor, R, which denotes the total amount of data re-use available in a particular layer. For fully connected layers, $R = B$, where B is the batch size. For convolutional layers, $R = B \times P$, where P is the number of output pixels in the output image. Furthermore, the RLE decoder allows us to easily avoid calculating any zero valued weights. In comparison to the benchmark BNNs, which have strictly dense weights, only the non-zero weight computations need to be calculated. The sparsity factor, then becomes a multiplier which significantly reduces the cost per operations and hence the regularization techniques discussed in this work directly minimize hardware costs during training. To this end, an *effective operation* cost is introduced, given by: $C_e = \gamma * (C_{op} + C_d/R)$, where γ is ratio of non-zero weight values to total weights in the layer, C_{op} is the proportion of the KU115 which is utilised by a single operation and C_d is the proportion of the KU115 which is utilised by the decoder. For these calculations, we are assuming 70% of the LUTs and

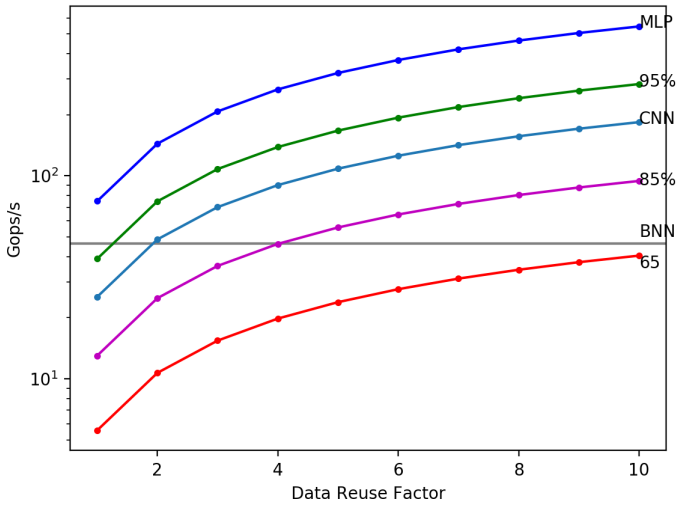


FIGURE 3.6. Effective throughput: BNNs vs TNNs (type 3) while varying γ and R.

100% of the DSPs can be utilised for compute. Some examples of these types of utilization numbers can be found in [27, 35, 159]. An *effective throughput* can then be calculated as: $T_e = 1/C_e * 250\text{MHz}$. Figures 3.6 & 3.7 show the effective throughput of type 1) & 2) networks respectively, while varying γ and R. The horizontal lines represent the benchmark BNN networks, MLP and VGG (VGG is labelled as CNN in figures) from the results in Tables 3.2 & 3.3 and the other percentages represent networks of the same type with varying sparsities. Note that these are theoretical peak values and further overheads are likely for all datapoints when they are implemented in a real system. For type 2), a lower sparsity factor is required to improve on the benchmark throughput as these operations are more expensive and hence every zero weight has a greater hardware benefit than for the type 1). We make note that a physical implementation of these designs was not completed and is considered beyond the scope of this research topic. The hardware cost model provided is intended to make an accurate estimate of this.

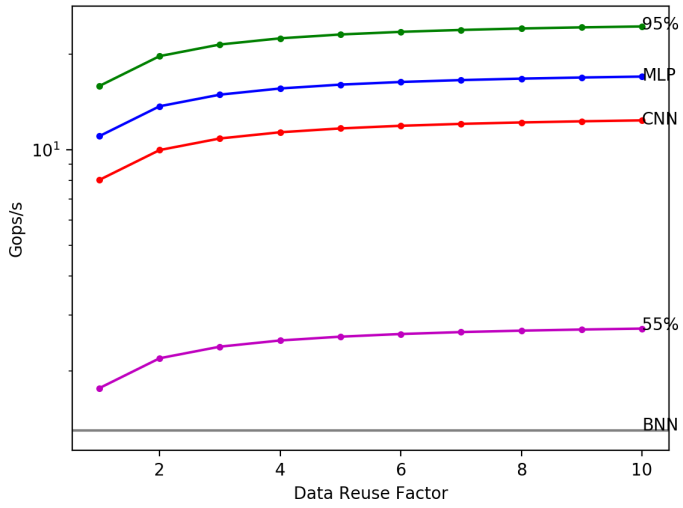


FIGURE 3.7. Effective throughput: BNNs vs TNNs (type 1) while varying γ and R. Note: VGG is labelled as CNN.

3.5 Summary

In this Chapter, a training methodology was presented which leverages sparse representations of bitwise DNNs to improve hardware performance. This contributes to the applicability of DNNs on embedded devices and specialized hardware. A sparse TNN training method is introduced which uses a quantization threshold hyperparameter, complemented by L2 regularization and quantization pruning to substantially reduce the memory requirements and computational complexity. This was shown using different network topologies on the MNIST and CIFAR10 benchmarks. Additionally, a hardware cost model was derived to evaluate the proposed representations for FPGA hardware.

CHAPTER 4

Customizing Bitwise Networks For Hardware Platforms

In the previous Chapter, we introduced a technique for pruning individual weight to achieve high network sparsity and demonstrated how this could benefit hardware performance. To achieve this we discussed storing the weights in a compressed format and implementing a decompressor between the weight storage buffers and the PEs. However, given a typical DNN implementation on specialized hardware, which rarely support decompression, performance improvement from this type of sparsity is not possible.

In this section, we present the technique *Customizing Bitwise Deep Neural Networks For FPGAs*, a hardware-software co-design methodology for optimizing FPGA performance. A training methodology to prune out weight groups in a computationally structured manner is presented. We do this by removing full sets of weights from a convolutional kernel and their corresponding output feature maps, so that we can completely ignore the computation. This type of pruning is known as *filter pruning*. This avoids the need to implement decompressor hardware to handle the sparse computations as in the previous chapter. The choice of layers in which we choose to prune is guided by a hardware cost model of the underlying FPGA implementation. The choice of filters to prune within that layer is guided by a novel metric which we derive.

In our method, it is argued that instead of solely focusing on developing efficient architectures to accelerate well-known networks, modifying these networks to suit the underlying hardware implementation should be explored. Thus, with this technique, we don't only consider the amount of sparsity, but also the underlying target hardware. A fully automatic toolflow is developed which efficiently utilizes FPGA hardware whilst satisfying a predefined accuracy threshold. Although fewer weights are removed in comparison to traditional pruning

techniques designed for software implementations, the overall model complexity and feature map storage is greatly reduced. As in the previous Chapter, the focus is on bitwise networks, however the resulting representation requires dense matrix computation. The AlexNet [69] and TinyYolo [112] networks are implemented on the large-scale ImageNet classification and PascalVOC object detection datasets, to demonstrate up to roughly $2\times$ speedup in frames per second and $2\times$ reduction in resource requirements over the original network, with equal or improved accuracy.

4.1 Background

Pruning seeks to remove parameters from the network, reducing the on-chip memory, memory bandwidth and computational requirements, whilst minimizing any loss in accuracy. In some cases, pruning can even improve accuracy.

Pruning methods range from fine-grained weight connection pruning as seen in Chapter 3, to coarse-grained approaches such as filter pruning which is used in this Chapter. The former removes individual weights and results in high overall sparsity at the cost of poor data locality and irregular data access patterns [84]. The latter results in a lower total number of pruned weight connections, but a structured sparse representation that maintains the regular data access patterns of the original network. This regular structure can make use of existing optimized hardware implementations, such as those described in [145].

One of the limitations of existing filter pruning methods is that they do not take into account the underlying FPGA accelerator architecture. While reducing the total number of network weights, or filters, reduces the number of operations and memory use, these are not the only factors limiting performance. Due to high data re-use, modern FPGA-based low precision CNN accelerators attempt to store feature maps (FMs) in on-chip memory in addition to weights, and divide the computation between processing elements (PEs). This ensures the performance is bounded by the available computation as opposed to the available off-chip memory bandwidth. It follows that pruning methods should be designed to ensure PEs are used efficiently. In addition, filters in modern benchmark CNNs are typically designed with

high precision for the first and final layers, as well as differing FM sizes. This potentially creates large resource imbalances between layers for FPGA designs, which can translate to sub-optimal performance.

4.2 Contributions

Instead of focusing on using pruning methods to maintain or improve accuracy, in this work, a method is presented which prioritizes resource reduction and efficient use of the underlying architecture for a given accuracy threshold. There's also a discussion of how different pruning strategies are required depending on the device constraints and the accelerator architecture. In particular, dataflow architectures are studied which are restricted by the throughput of the slowest layer and available on-chip memory. For a predefined accuracy threshold, it is demonstrated how the pruning method improves the performance over existing methods. Altogether, this work makes the following contributions:

- The first study on bitwise network filter pruning. A novel quantization error pruning heuristic is proposed which minimizes the error in the weights rather than in the output feature maps and show that it outperforms conventional methods. This method can be applied to applications on all hardware platforms.
- A resource-aware method is proposed for customizing bitwise CNNs to underlying FPGA dataflow architectures. This is able to significantly reduce resource requirements and improve efficiency, making FPGAs a higher performant implementation alternative for CNN computation.
- Results are compared to traditional pruning which prioritise sensitivity analysis of each layer rather than the underlying hardware and show significant reductions in complexity. This improves the scalability/applicability of bitwise CNNs on FPGAs.
- The highest reported frames per second (FPS) is achieved, FPS/kLUT and FPS/BRAM on the popular AlexNet network for the ImageNet dataset [69]. This improves the amenability of high throughput implementations of large networks/datasets on constrained FPGA environments.

4.3 Network Quantization Setup

For this Section 4, let us represent the full precision weight matrices for any layer l by $\boldsymbol{W}^{[l]}$. For QNNs, $\boldsymbol{W}^{[l]}$ is approximated with a quantized weight matrix $\boldsymbol{Q}^{[l]} \in \mathbb{C}^{[l]}$ where $\mathbb{C}^{[l]}$ represents the possible quantized weight values in layer l. The quantized weight matrix for BNNs is described by (4.1):

$$\boldsymbol{Q}^{[l]} = sign(\boldsymbol{W}^{[l]}) \tag{4.1}$$

Alternatively, TNNs are described by (4.2), where η is the quantization threshold hyperparameter.

$$\boldsymbol{Q}^{[l]} = \begin{cases} 1 & \text{if} \quad \boldsymbol{W}^{[l]} > \eta \\ 0 & \text{if} \quad -\eta \leq \boldsymbol{W}^{[l]} \leq \eta \\ -1 & \text{if} \quad \boldsymbol{W}^{[l]} < -\eta \end{cases} \tag{4.2}$$

These equations are very similar to what is described in Section 2.3.3. To more accurately approximate the dynamic range of full precision weights, $\boldsymbol{Q}$ is multiplied by a scaling factor coefficient α [156]. For example the fully-connected layer, with biases ignored, is computed by (4.3), where g is the activation function, $\boldsymbol{h}^{[l]}$ and $\boldsymbol{h}^{[l-1]}$ are the output and input vectors of layer l.

$$\boldsymbol{h}^{[l]} = g(\alpha^{[l]}\boldsymbol{Q}^{[l]} \cdot \boldsymbol{h}^{[l-1]}) \tag{4.3}$$

For the networks discussed in this work, the activations are quantized using signed two's complement fixed point representations. This is accomplished by quantizing a real number $x \in [0, M]$ to a k-bit number:

$$G(x) = \frac{1}{2^f} floor((2^f)x + \frac{1}{2}) \tag{4.4}$$

where M is the upper bound. M itself is bounded by its arbitrary unsigned two's complement fixed point representation where f is the number of fractional bits and $M = 2^{k-f} - 2^{-f}$. Similar activations have been proposed in [156] by fixing $M = 1$. During training both the full precision and quantized weight values are needed; for inference only the quantized weights are required.

4.4 CNN Acceleration

In this section, basic approaches are discussed for accelerating CNNs on GPUs, CPUs and FPGAs. Throughout this section, we use the following notation to describe the CNNs. For layer l in a CNN, it is assumed there are $I^{[l]}$ IFMs of dimensions $F^{[l]} \times F^{[l]}$ and $N^{[l]}$ filters of dimensions $K^{[l]} \times K^{[l]}$ (assume feature maps and filters of squared dimensions). The OFMs from layer l are the IFMs to layer $l+1$, and can be represented as $I^{[l+1]}$.

4.4.1 CNN acceleration on CPUs/GPUs

On a CPU or GPU, the typical approach to implement convolutional layers is so called lowering to matrix multiplication; this enables the use of optimized matrix multiplication kernels and described in detail in [14]. The basic concept is that convolutional layers involve multiplying a $K^{[l]} \times K^{[l]}$ window of data for every IFM by one or more filters of size $K^{[l]} \times K^{[l]}$ to produce each element of the OFMs. To achieve this using matrix multiplication, one image-matrix is constructed of dimensions $(F^{[l+1]} \times F^{[l+1]}) \times (K^{[l]} \times K^{[l]} \times I^{[l]})$ representing every window of data from the IFMs and multiply this with a second filter-matrix of dimensions $(K^{[l]} \times K^{[l]} \times I^{[l]}) \times N^{[l]}$ representing all of the filters. For efficiency, optimized GPU routines avoid performing this conversion in off-chip memory, and construct this in on-chip RAM [14]. Since the matrix multiplication kernels have previously been optimized, including binarized matrix multiplication kernels [22], the aim of pruning for GPUs and CPUs is simply to reduce the total number of operations by reducing $I^{[l]}$ and $N^{[l]}$.

4.4.2 FPGA-based CNN acceleration of Dataflow Architectures

One problem with the approach of lowering to matrix multiplication is that if the kernel windows overlap, the $(F^{[l+1]} \times F^{[l+1]}) \times (K^{[l]} \times K^{[l]} \times I^{[l]})$ matrix contains large amounts of redundant information. This translates to significant extra data transfer overhead. An alternative approach, potentially better suited to an FPGA, is to use a sliding-window unit (SWU) [128], which generates the image matrix from an incoming IFM. While this is a common approach for basic image or video processing kernels, because CNNs have multiple

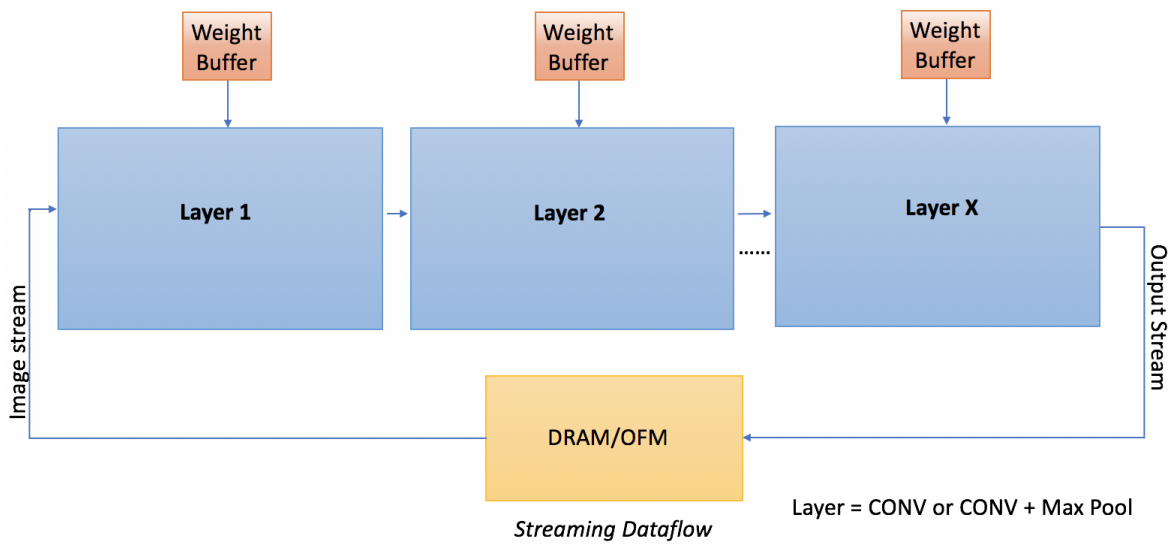


FIGURE 4.1. Generic system diagram for CONV layer computations

IFMs, it is desirable to use multiple SWUs to construct windows from each IFMs in parallel. In this case, a total of $I^{[l]}N^{[l]}$ filters operate in parallel per layer. Unfortunately, the I/O requirements to stream the data would typically be too large.

Instead, the fastest FPGA architectures attempt to fit the entire network on the FPGA, as in Figure 4.1. This is possible for smaller CNNs, in particular those that have been binarised [128]. In this case, I/O requirements are greatly reduced, as the input and output layers of CNNs are typically much smaller (for example, the input is typically an image, read one pixel at a time into the SWU, with the output being a classification, whereas middle layers tend to have many more IFMs). Unfortunately, it is not typically possible to create fully unrolled CNNs because the resource requirements are too large.

4.5 Hardware-Aware Pruning

In this section, an effective pruning method is described for dataflow architectures.

4.5.1 Layer Selection

To implement modern DNNs on low-end FPGAs, it is usually necessary to re-use the PEs. To enable this, data buffers are added to hold IFMs and OFMs between layers, as shown in Figure 4.2. The total number of operations for layer l is given by (4.5). Assuming that a layer has $PE^{[l]}$ processing elements that can be re-used to perform these operations. The total number of times that these PEs are reused is given by (4.6). Note that for maximum efficiency throughput must be matched between layers, to do this, the reuse should be the same for all layers. In a fully pipelined implementation $Reuse = 1$, in this case there is no need to use any memory: OFMs from one layer can go directly to the SWUs of the next layer. If $Reuse = 2$, then half of the time, OFMs must be stored, half of the time they can go directly to the SWUs of the next layer. As the reuse is increased, the PE requirements are reduced, but the RAM requirements are increased for intermediate buffers. The cost of BRAMs for OFMS per layer can be modelled using (4.7), where B_w denotes the number of words stored in a BRAM respectively.

$$OPS^{[l]} = K^{[l]} \times K^{[l]} \times I^{[l]} \times N^{[l]} \times F^{[l+1]} \times F^{[l+1]} \tag{4.5}$$

$$Reuse = \lceil \frac{OPS^{[l]}}{PE^{[l]}} \rceil \tag{4.6}$$

$$BRAM_{IFM_{l+1}} = \lceil \frac{I^{[l]} \times F_{in} \times F_{in} \times (Reuse - 1)}{Reuse \times B_w} \rceil \tag{4.7}$$

To help us create a working hardware design these equations can be utilized, along with a model of the resources cost per PE, the resource cost per SWU and the BRAM cost to store all network weights. A basic heuristic is followed for hardware-optimized pruning, given in Algorithm 5. This heuristic helps us to determine which layer to prune. A fully pipelined design ($Reuse = 1$) is begun with. If this does not fit on the FPGA, the design is constrained by the resources for PEs. There's the option of pruning the layer utilizing the most resources for PEs, or increasing $Reuse$. After this, if the design is constrained by the resources for PEs, there are the same options; if the design is limited by BRAMs, the layer utilizing the most BRAMs (FM + Weight memory) can be pruned. Once a design fits and satisfies the accuracy criteria, some optimizations described in Section 4.5.2 can be applied.

Algorithm 3 Neural Network Pruning Process For FPGA implementation

1. Initialize:
Choose PE size such that reuse = 1.
2. Iteratively prune filters:
while accuracy change $\leq \eta$ **do**
  Prune filters by 10% using (4.8)
  Re-train
  Save Model
end while
3. Reuse:
while design exceeds available FPGA resources **do**
  Increase hardware reuse
end while
4. Model Finetuning:
Add/Remove filters and ensure they are a multiple of the number of PEs
Re-train
5. Check:
if Accuracy satisfies η **then**
  continue
else
  Go to Step 3 using saved model from previous iteration.
end if
6. Deploy

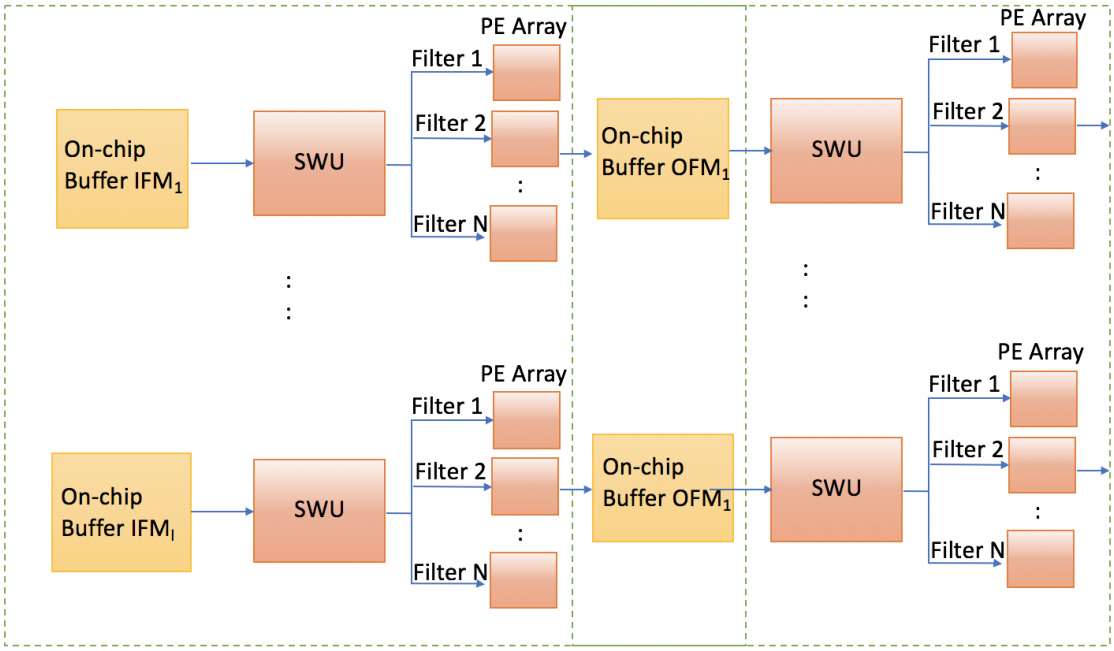


FIGURE 4.2. Advantage of reading from on-chip BRAMs

4.5.2 Model-finetuning

Pruning layers involves the removal of feature maps. To ensure efficient use of the underlying hardware, layers are pruned so that the remaining number of feature maps in a layer is a multiple of the number of PEs assigned to that layer, i.e. $N\%PE = 0$. In addition, it is also attempted to *increase* the number of feature maps in other resource-inexpensive layers such that $N\%PE = 0$. This is only the case if it doesn't impinge on the desired performance of the design. Once again, this is to ensure maximum efficiency. This can translate into a significant increase in the accuracy of the design, or recover some of the accuracy that is lost by pruning the most expensive layer. But importantly, comes with a minimal increased cost in terms of resources: there is only a slight increase in weight memory/resources. There is also a slight increase in power requirements as more of the circuit will be active. Note that on other architectures such as GPUs or CPUs, this decision is unlikely to be taken as it would increase the workload; instead the ideal course of action is to simply prune the overall network to reduce the total number of operations.

4.5.3 Quantization Error Pruning

Once the layers to prune have been chosen using the method in Section 4.5.1, which filters must be chosen from a given layer to prune. Pruning approaches attempt to remove weights or filters with minimal reduction in accuracy. With traditional floating point network pruning, error in the OFMs is what impinges overall accuracy. It follows that a typical pruning technique is to select weights or filters which have the lowest magnitudes and hence contribute to the smallest activation outputs. Unfortunately, this technique does not easily extend to bitwise DNNs. For example, in the case of BNNs, real-valued weights are quantized and hence regardless of their magnitude, if two weights have the same sign, they have the same quantized value and hence equal contributions to the OFMs. As such, a different approach is taken. It is hypothesized that for highly quantized representations, such as bitwise networks, it is the quantization error in the quantized weight values that impinges overall accuracy. As such, the importance of filters is ranked based on the highest accumulated mean-squared

quantization error (MSQE) for each filter, as described by (4.8), where $n = K^{[l]} \times K^{[l]} \times I^{[l]}$.

$$MSQE_N = \frac{1}{n}\sum_{z=1}^{n}(q_z - w_z)^2 \tag{4.8}$$

where q_z and w_z represent each quantized and real-valued weight in each filter, respectively. Filter ranking by quantization error pruning will also benefit CPU and GPU implementations in obtaining greater accuracy for bitwise networks. However, once again, it is highlighted that such networks can be implemented most efficiently using FPGAs.

4.5.4 Filter Ranking

As filter pruning for low precision networks is yet to be explored, it is demonstrated empirically, the effectiveness of pruning 10% of filters from the first three layers iteratively, based on three metrics: randomly selecting filters (Random), ranking filters based on the highest MSQE (Qerror) and the highest magnitude of real-valued weights (Real). While the results only investigate BNN accelerators, for this analysis, both binary (BNN) and ternary (TNN) AlexNet experiments are observed in Figure 4.3. It is clear that low pruning rates improve accuracy showing the regularization effect of filter pruning. Importantly, it is evident that Qerror pruning improves accuracy by up to 0.8%. As the pruning rate increases, both Qerror and Real are relatively similar but substantially better than randomly pruning the weights. After calculating these, each filter is ranked to determine the amount to prune for each iteration. Figure 4.4 shows each layer for the BNN trained on CIFAR10 and shows that roughly 5-10% of filters in each layer have significantly higher quantization error than all other layers.

4.5.5 Data Fine-tuning

To avoid excessive training times, and ensure minimal accuracy loss, after filters are selected to prune, they are removed to reconstruct the network with its new customized configuration by initializing the weights with values from the saved filters. Retraining is then implemented using a quarter of the initial training epochs to fine-tune weight values and recover the

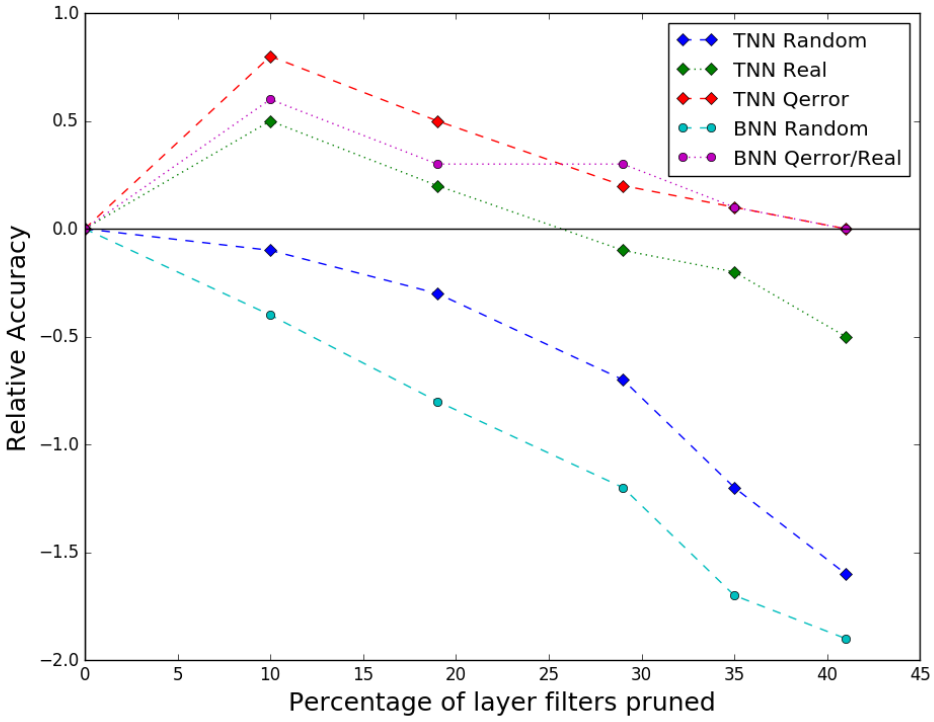


FIGURE 4.3. Measuring the relative accuracy of AlexNet against the pruning percentage for different filter importance ranking metrics

accuracy for the new CNN configuration. As such, the pruning process is done iteratively for some predefined percentage of filters in each iteration. In this work, $\approx 10\%$ of filters are pruned per iteration. In the last pruning iteration it is ensured $N\%PE = 0$, as discussed in Section 4.5.2.

4.6 Experimental Setup

In this section, the device and architecture used to demonstrate the effectiveness of the pruning strategy is described.

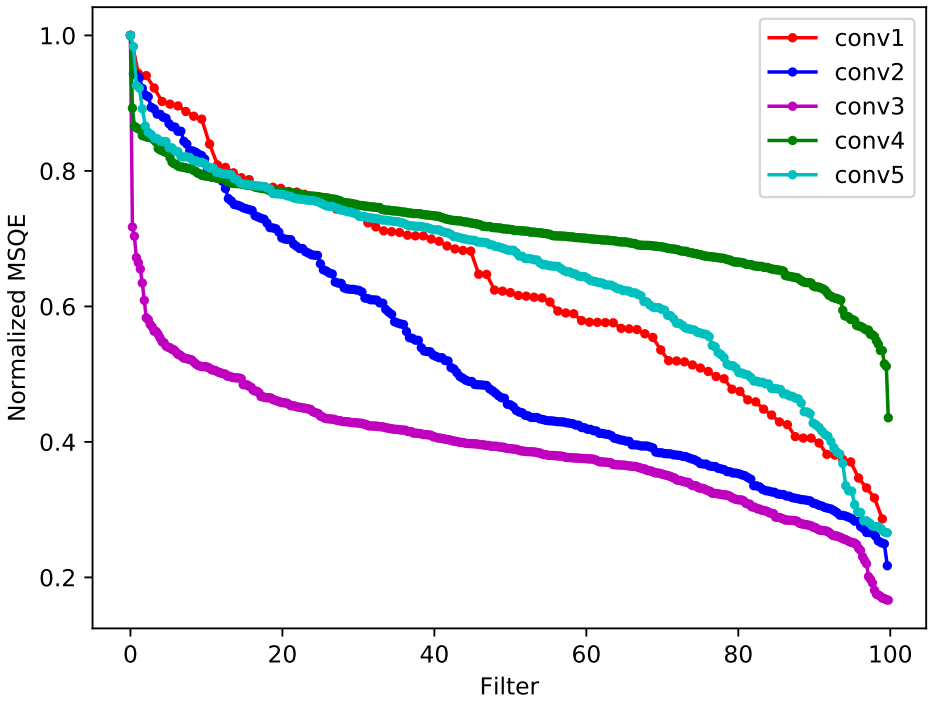


FIGURE 4.4. The normalized MSQE of each filter from highest to lowest for each layer of binarized AlexNet

4.6.1 Networks

The methods were evaluated on 2 networks trained on the ImageNet dataset. We used these networks as opposed to the ones in Chapter 3 because they are more suitable for real-world applications and our target hardware supported their computation. Firstly, we use an AlexNet-variant [69] inspired by DoReFa-Net [156] which has 1-bit weights and 2-bit activations. This is used for classifying the ImageNet dataset [24] which has 224×224 input image sizes. Secondly, a fully-convolutional network is considered, TinyYolo [112], whereby we binarize the network weights and use 3-bits for activations. This network is trained on the Pascal VOC dataset for Object Detection which has a 418×418 input image size. The TinyYolo model is much more compact in terms of weight memory footprint than AlexNet, although has a larger total number of operations and input image size, as represented in Tables 4.1 & 4.2. In these

TABLE 4.1. DoReFa-Net Network Configuration

Layer	K	N	F	% Ops	%Mem
Conv1	12	96	54	6.24	22.8
Conv2	5	128*	54	46.24	23.0
Conv3	3	384	27	33.30	36.9
Conv4	3	192*	14	6.71	10.4
Conv5	3	256	8	4.48	6.9
FC1	1	4096	1	1.95	-
FC2	1	4096	1	0.87	-
FC3	1	1000	1	0.21	-

tables, the network configurations are displayed, the percentage of operations per layer and also the percentage memory in terms of FMs and weights.

For both networks, the first and last layers are quantized to 8-bit representations and the activations bit widths for all layers is the same. Mixed layer precisions are utilized because accurate low-precision neural networks on large scale complex datasets using conventional quantization functions, typically require the first and last layers to have higher precisions than all other layers [12]. Finally, the inputs are also quantized to 8-bit values, for no accuracy loss.

TABLE 4.2. TinyYolo Network Configuration

Layer	K	N	F	% Ops	%Mem
Conv1	3	16	418	2.14	8
Conv2	3	32	210	5.72	12
Conv3	3	64	106	5.72	6
Conv4	3	128	54	5.72	4
Conv5	3	256	28	5.72	3
Conv6	3	512	15	5.72	6
Conv7	3	1024	15	22.88	20
Conv8	3	1024	15	45.76	40
Conv9	1	125	13	0.62	-

4.6.2 Computing Core

We investigate the effectiveness of this pruning method using the FINN [128] hardware library in Vivado HLS. This can be used to create architectures similar to those described in

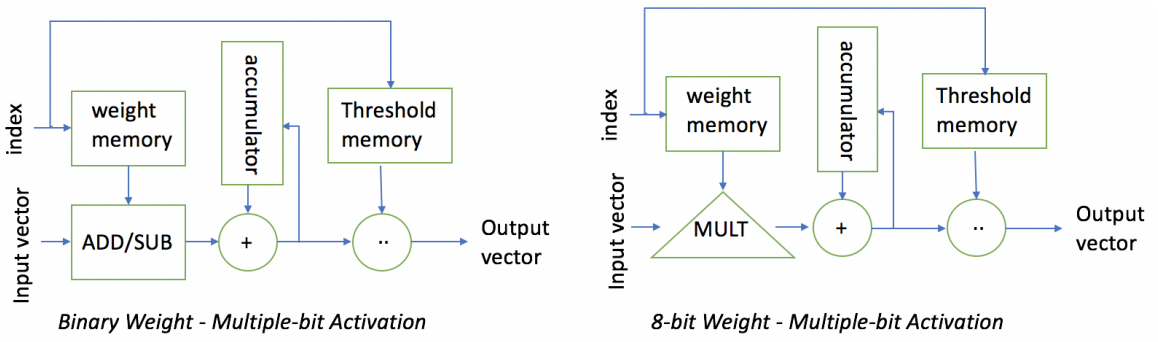


FIGURE 4.5. PEs used for MAC computations

Figure 4.1. The FINN architecture uses SWUs to feed so-called Matrix-Vector Threshold Units (MVTU) for CONV or FC layers. SWUs can also feed Pooling Units (PU). There are three key parameters for FINN: PEs, SIMD lanes and Matrix-Multiple Vector (MMV) length. PEs refer to the number of OFMs evaluated in parallel, SIMD refers to the number of parallel IFMs processed in each PE, MMV controls to the amount of output pixels evaluated in parallel.

We extend the PEs in the MVTU to arbitrary precision activations for the middle layers, which replaces the XNOR-popcount from the PEs in FINN by either an addition or subtraction (ADD/SUB) depending on the sign of the weight. The datapath for this PE is shown in Figure 4.5. For the 8-bit input vector and 8-bit weights in the first layer, DSPs are used for the MAC before being fed into the thresholding unit. The overall system designs for the architectures discussed in this work use the MVTU extensions to compute the CONV layers.

4.7 Results

In this section, the experimental design is described and results from the implementations presented.

4.7.1 Streaming Dataflow

The case of a streaming dataflow architecture corresponding to Figure 4.1 is investigated. Here, there is an option to either add more hardware or prune feature maps to improve throughput.

4.7.1.1 System Design

For dataflow architectures, FINN allows the selection of parallelism ($P = SIMD \times PE \times MMV$) for each layer depending on the number of operations. The overall throughput of the network is dependent on the layer which requires the largest number of cycles to compute, thus the best Throughput/Resources ratio is achieved when all layers have same compute latency. The neural network is analyzed in a per-layer fashion, the compute requirements evaluated and the parallelism determined. For the AlexNet implementation, a large proportion of total operations is done in the 2nd and 3rd layers. Additionally, the first layer requires more resources per MAC operation, as it is computed with 8-bit inputs and weights, hence requiring DSPs for implementation and more memory resources for weight storage. Hence, the first three layers are chosen to be pruned equally for resource and throughput improvements. This is done iteratively for a total of 40% of filters. For the first layer, the number of IFMs is 3, meaning $SIMD \leq 3$. The number of PEs and MMV is restricted by the number of OFMs and OFM dimension respectively, hence $MMV \leq 54$ and $PE \leq 60$ (as the OFMs are reduced to 60 after pruning and model fine-tuning). Also $SIMD = 3, PE = 60$ & $MMV = 18$ are set. In order to avoid resource inefficiencies, the MMV is restricted to 18 in this case as the throughput mustn't exceed the latency of the SWU to construct the image matrix. To achieve load balancing for all succeeding layers, the resources are allocated such that the estimated cycles matches this first layer. The CONV layers consist of 97% of the total operations and hence the throughput of the device is largely constrained by these layers. In this system design, the last CONV layer writes to the host memory and FC layers are computed on the host CPU. This is due to their large BRAM and low operations requirement. A large proportion of the operations and weight + FM memory in the original TinyYolo network is in the last two quantized layers. Hence, these layers are pruned to improve the load-balancing and

TABLE 4.3. AlexNet and TinyYolo Implementation Resource Usage

AlexNet	LUTs	DSPs	BRAMs	Freq.	FPS	Acc.
Original	375,037 57%	2,693 49%	1,527 35%	159	3,265	50.1
Pruned-30%	228,104 34%	1,938 35%	1,057 25%	172	3,530	50.3
Pruned-40%	188,924 25%	1,698 25%	955 20%	185	3,797	50.1
TinyYolo						
Original	179,265 27%	500 9%	2,731 63%	233	1,189	47.8
Pruned-50%	234,883 35%	189 3%	1,930 45%	240	1,226	48.5
Available	663,360	5,520	4,320			

also reduce the number of BRAMs. Also, the number of second layer filters is doubled, as increasing the number of PEs in that layer from 32 to 64 imposes minimal resource impact and helps preserve accuracy during re-training.

4.7.1.2 Performance

The performance of both pruned and non-pruned versions of AlexNet are measured by implementing the dataflow architecture on the Xilinx KU115 board. For the pruned topologies, there are two alternatives for improving the hardware; 1) By fixing the model's parallelism, concurrency can be reduced and computational resources saved. This translates to a greater ability to fit the model on low-cost FPGAs and achieve higher frequencies. 2) Increasing the parallelism of the original architecture, which reduces latency and improves the FPS. Following 1), the same parallelism is maintained for pruning AlexNet at 30% & 40% as displayed in Table 4.3. This allows us to increase the frequency of the design up to 185Mhz. The frame rate also improves whilst resources are significantly reduced: 40% pruning translates to a reduction in LUTs by roughly 50% and BRAMs and DSPs by roughly 40%. For TinyYolo, there is a slight increase in FPS whilst resources are again reduced: namely a 30% savings in BRAMs for on-chip parameter storage. Note also that the pruning strategy improves the accuracy of TinyYolo by 0.7mAP and AlexNet-30% by 0.2% and remains the same for

TABLE 4.4. Scaling Up Parallelism For Pruned AlexNet

	Original	Pruned-30%	Pruned-40%
Freq (MHz)	159	130	150
LUTs	375,037	410,073	302,325
BRAMs	1,527	1,333	1,156
DSPs	2,693	3,772	2,693
FPS	3,265	5,359	6,172
FPS/kLUT	8.70	13.14	20.44
FPS/BRAM	2.14	4.02	5.39

AlexNet-40% as the accuracy threshold $\eta = 0$. For 2) the parallelism of the implementation is scaled up. A similar frequency to the original AlexNet network is achieved. Also, the throughput is 6,172 FPS which is roughly a $2\times$ speedup over the original network. This is displayed in Table 4.4, along with a significant reduction in resources for the same accuracy. By significantly decreasing resources in the first layer through pruning, the parallelism is able to be scaled up in all other layers to improve FPS. Scaling for the original network was unable to achieve the FPS reported for the pruned topologies due to device resource constraints.

4.7.2 Comparison To Previous Work

Thus far, the results discussed have been compared to an original implementation of AlexNet and TinyYolo. A comparison to previous implementations is now presented in the literature for the popular AlexNet benchmark in Table 4.5. The W/Act Prec) metric is for the weight and activation precision. To calculate the FPS/BRAM against previous implementations, a direct conversion of M20k to BRAM(18k) is assumed. Also, to approximate the FPS per reconfigurable logic units (ALMs/LUTs), for simplicity 8-input ALMs and 6-input LUT units are assumed to be comparable [4]. It is evident the design achieves a highly superior $3.8\times$ improvement in FPS, $4.9\times$ in FPS/kLUT and $7.5\times$ in FPS/BRAM over previous state-of-the-art implementations. This demonstrates the largely efficient use of resources in the design as the pruning strategy is able to remove redundant filters which don't contribute much to the overall accuracy and only impinge on the potential hardware performance of the design. The hardware design here is used as a proof of concept to the pruned network representations

TABLE 4.5. Comparison to previous AlexNet implementations

	Li16 [83]	Aydonat17 [6]	Moss17 [100]	**Ours**
Device	VC709	Aria 10	Aria 10	KU115
Freq	156	303	312.5	150
LUTs	273,805	246k (ALM)	427.2k (ALM)	302,325
BRAMs	1,913	2487 (M20k)	2000 (M20k)	1,156
DSPs	2,144	1,476	1518	2,693
W/Act.	16/16-bit	16/16-bit	1/1-bit	1/2-bit
Acc.	-	56.0	44.2[1]	**50.1**
FPS	391	1,020	1,610	**6,172**
FPS/kLUT	1.43	4.14	3.77	**20.44**
FPS/BRAM	0.20	0.37	0.72	**5.39**

presented. This is not a heavily optimised design, its possible a hardware designer could improve on these results if tailoring it to a specific FPGA/ASIC architecture.

4.7.2.1 Comparison To Other Pruning Techniques

The method is compared against the original network and two other filter pruning strategies: 1) the naive method of pruning each layer evenly and 2) traditional machine learning methods which prune each layer according to their sensitivities. For the analysis, the sensitivities of each layer are empirically determined by pruning each layer once at a time and analyzing accuracy degradation for a given pruning rate. All methods prune to their pre-specified tolerable losses. From Figure 4.7 it is evident that a significant improvement in the required BRAMs and number of operations for all strategies. The resource-aware strategy outperforms the other strategies as it only prunes layers which directly improve the desired hardware performance of the design. The other pruning methods aim to eliminate a high total number by filters whilst preserving accuracy. This leads to filters being removed from all other layers, meaning the resource-heavy layers cannot be pruned as much. This suggests that for FPGA implementations, the method of tailoring the pruning to the underlying architecture by prioritizing resource-heavy layers is much more effective.

[1] Accuracy obtained from 1/1-bit AlexNet in [93]

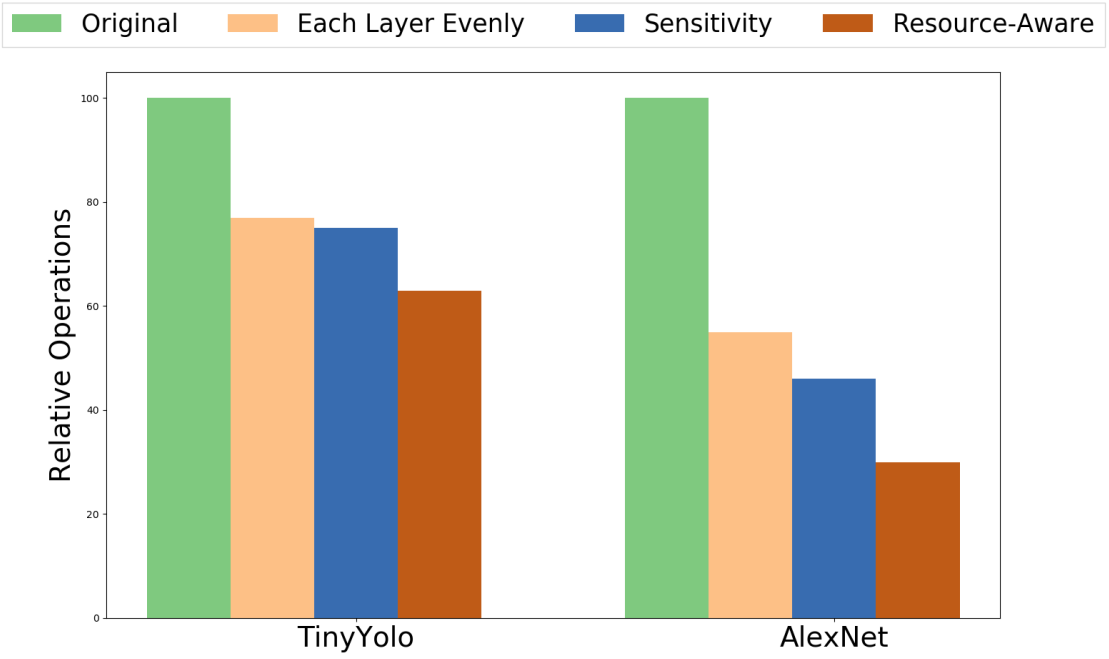


FIGURE 4.6. Total operations of networks for different pruning methods

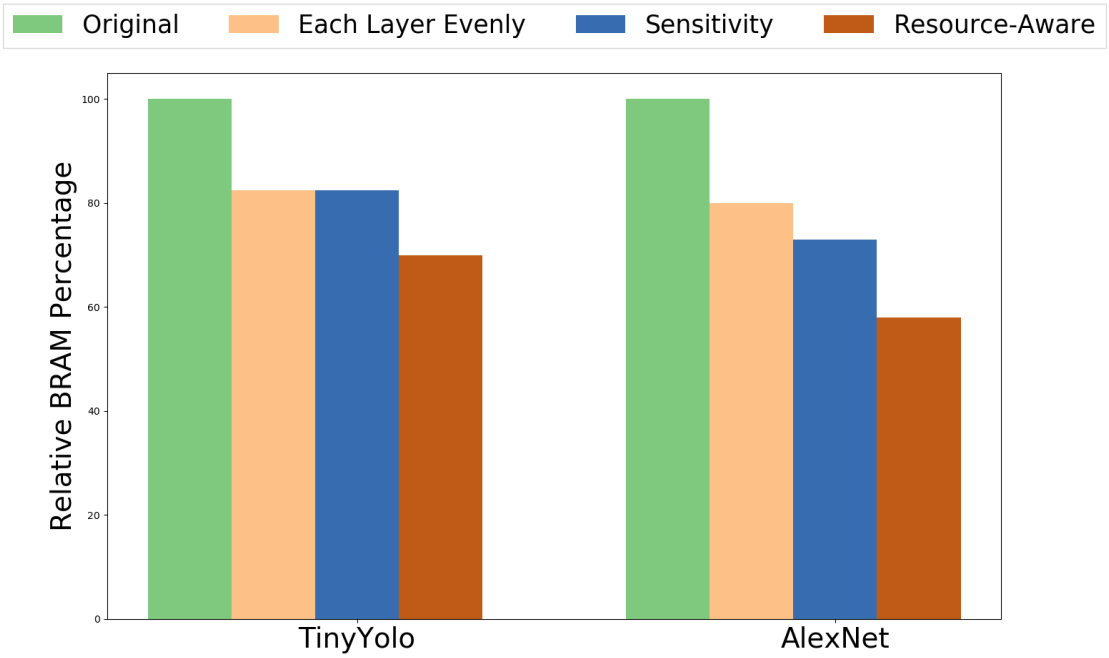


FIGURE 4.7. Relative BRAM requirement for different pruning methods

4.8 Summary

This Chapter presents a filter pruning method for customizing bitwise DNNs to FPGA hardware. A novel quantization pruning heuristic is investigated which minimizes the error in the weight values rather than the output feature maps. This is used to rank filters. The amount of filters to be pruned is then determined via a resource-aware evaluation of the underlying FPGA architecture. We then compare using this method against previous work which only considers the sensitivity of the weights rather than the underlying hardware. Following this, we describe target FPGA implementation dataflows for which we use our training methodology to customize our CNNs to. We then evaluate these methods and report resource usage of the implementation for the pruned methods and the unpruned methods. Lastly, we compare to state-of-the-art FPGA implementations to demonstrate state-of-the-art performance in terms FPS, FPS/kLUT and FPS/BRAM.

CHAPTER 5

Improving Quantization Of Bitwise Networks

Chapters 3 & 4 focused on training methods to reduce the number of required operations for inference computation via network pruning. Both these Chapters utilized bitwise networks to enable high hardware performance implementations as they reduce the cost of arithmetic operations. However, these techniques lack accuracy, particularly on complex datasets such as ImageNet. Thus, in this Chapter, we explore how to improve the accuracy of bitwise networks whilst attempting to maintain the hardware performance. To do this effectively, we must improve learning capabilities and understand data movement in the underlying hardware.

This chapter discusses a novel training methodology for quantizing networks such that the simplicity of the computation is preserved. As discussed in Section 2.3.4, for very low-precisions such as bitwise networks with 1-8-bit activations, the information loss from quantization leads to significant accuracy degradation. Our methodology aims to reduce this loss by learning a symmetric codebook for particular weight subgroups. The subgroups are determined based on their locality in the weight matrix, such that the hardware simplicity of the low-precision representations is preserved. Empirically, we show that our methodology can substantially improve accuracy for networks with extremely low-precision weights and activations. Additionally, we demonstrate that this representation imposes minimal or no hardware implications to more coarse-grained approaches.

Due to the ordering of the scaling factors maintaining data regularity, we show how the network representations in this chapter can be computed similarly to a bitwise network with some minor additional computational complexity. However, due to the addition of multiple scaling factors in each layer, the codebook of weight values does not necessarily represent a bitwise network as in Chapters 3 & 4 but rather a uniformly quantized FX network.

5.1 SYQ: Learning Symmetric Quantization For Efficient Bitwise Networks

As discussed in Section 2.3.4, quantized neural network training involves low-precision networks having a set of full-precision weights which are quantized before inference. As the quantization functions are piecewise and constant, the gradients of quantized weights are calculated and applied to update their corresponding full-precision weights. Similarly, derivatives of quantized activations are calculated by using a non-constant differentiable approximation function. This type of training was first proposed as the Straight Through Estimator (STE) [9] as discussed in Section 2.3.4. The problem is that without an accurate estimator for weights and activations, there exists a significant gradient mismatch which impinges on learning. Seemingly, as discussed in [95], activations are less sensitive to quantization than weights for image classification problems due to weight reuse in Convolutional (CONV) layers affecting multiple operations. To overcome this, methods such as increasing the weight codebook by applying a scaling factor to all weights in a layer, provides better approximations for weight distributions and greater model capacity [82]. This is computationally inexpensive and can be represented as multiplying each weight layer's matrix by a diagonal scalar matrix which only requires storage of one value. Applying fine-grained scaling factors has also been shown to improve accuracy by increasing model capacity [92], [110]. The problem with all of these fine-grained approaches is either large storage requirements for the scaling factors or high computational complexity due to irregular codebook indices. We present Learning Symmetric Quantization (SYQ), a method to design binary/ternary networks with fine-grained scaling factors which preserve these complexities. We do this by learning a symmetric weight codebook via gradient-based optimizations which enables a minimally-sized square diagonal scalar matrix representation. To reduce the large information loss from CONV layer quantization, we use a more fine-grained pixel/row-wise scaling approach, rather than layer-wise scaling in Fully-Connected (FC) layers. In the process, we significantly close the accuracy gap for low-precision networks to their floating point counterpart, whilst preserving their efficient computational structures. Our work makes the following contributions:

- Our approach significantly improves the ability of convolutional weights to learn low-precision representations. This is useful as most layers in modern network architectures consist of convolutions which are typically the least redundant layers.
- The proposed method reduces the computational complexity of traditional fine-grained low-precision scaling and imposes minimal hardware costs to layer-wise scaling.
- On state-of-the-art networks such as AlexNet, ResNet and VGG, our method is empirically shown to improve accuracy for 1-2 bit weights and 2-8 bit activations.

5.2 Related Work

Bitwise Networks: Most methods for training low-precision DNNs maintain a set of full precision weights that are deterministically or stochastically quantized during forward or backward propagation. Gradient updates computed with the quantized weights are then applied to the full precision weights [20], [63], [87]. To produce state-of-the-art results on larger models, [110] proposed scaling the quantized weights by the expectation of real-valued weights to recover the dynamic range of each layer. Reference [82] also implemented a similar technique for ternary networks and optimised a non-zero quantization threshold as a function of the weight expectation. Other gradient-based optimization methods for the scaling factor have been introduced [158].

Low-precision DNNs: Other methods of quantization have also been implemented, i.e. re-training networks using incremental weight subgrouping to produce no accuracy loss for 5 bit weights [155]. Multiple binarizations and a scaling layer were described in [124] to improve accuracy and binarize the last layer. Logarithmic data representations were used to approximate the non-uniform distribution of the weights, activations and gradients down to 3-bits with negligible accuracy loss [95]. Activations quantization has also been investigated with frameworks created for varying activation bitwidths [156] and both weights and activations [109]. Improving the network learnability under low-precision weights and activations was analysed in [12]. More fine-grained approaches of quantization have

effectively clustered weights or grouped filters together and quantize differently based on their statistical distributions [28], [92] . Increasing model capacity by applying scaling factors to positive and negative values separately was proposed in [158].

Low-precision DNN Hardware: Also, many low-precision DNN hardware implementations have been published [131], [50]. For example, FINN [38], [126] demonstrated the hardware performance gains of being able to store all network weights in on-chip memory by implementing binarized neural networks on FPGAs.

5.2.1 Low-precision Networks

Similarly to as discussed in Section 2.3.2, for low-precision DNNs, the distribution of full precision weight matrices for each layer $\boldsymbol{W}_l$ are approximated by a function f, resulting in a quantized weight matrix $\boldsymbol{Q}_l$:

$$Q_{l_{i,j}} = f(W_l)_{i,j} \tag{5.1}$$

for $W_{l_{i,j}} \in \mathbb{R}$ and $Q_{l_{i,j}} \in \mathbb{C}$. The codebook $\mathbb{C} = \left\{c_1, c_2, ..., c_r\right\}$ is a set of all possible values for $Q_{l_{i,j}}$ where $c_i \in \mathbb{R}$ and $i \in \mathbb{R}^+$ represent each codebook value and index respectively. For example, binary and ternary weight spaces have $\mathbb{C} = \left\{-1, +1\right\}$ and $\mathbb{C} = \left\{-1, 0, +1\right\}$ respectively. As similarly discussed in Section 2.3.3, efficient functions for binarizing and ternarizing weight parameters have been proposed as piecewise constant functions in [82], such that:

$$\boldsymbol{Q}_l = sign(\boldsymbol{W}_l) \odot \boldsymbol{M}_l \tag{5.2}$$

with,

$$M_{l_{i,j}} = \begin{cases} 1 & \text{if} \quad \left|W_{l_{i,j}}\right| \geq \eta_l \\ 0 & \text{if} \quad -\eta_l < W_{l_{i,j}} < \eta_l \end{cases} \tag{5.3}$$

where $\boldsymbol{M}$ represents a masking matrix, η is the quantization threshold hyperparameter. $\eta = 0$ for binary networks and in our work we set $\eta = 0.05 \times max(|W_l|)$ for ternary networks as in

[158]. The issue with discretization of the weights, is that it leads to the vanishing gradients problem [9]. To overcome this, an STE is defined to replace the zero derivatives from the piecewise constant function in (5.2), by a non-zero surrogate derivative [63]. During training $\boldsymbol{Q}_l$ is used for inference and backpropagation, and the corresponding elements in $\boldsymbol{W}_l$ are updated based on these gradients. Hence the STE is defined as:

$$\frac{\partial \hat{E}}{\partial W_{l_{i,j}}} = \frac{\partial \hat{E}}{\partial Q_{l_{i,j}}} \tag{5.4}$$

where $\hat{E}$ is the error function for a network without scaling factors. After training, the full precision weights are discarded and we require only the quantized weights for deployment. Whilst these methods greatly reduce computational complexity by eliminating floating point MACs, they increase the difficulty of learning.

5.3 Ordered Scaling Factor Representations

In this section we discuss the fundamentals of applying scaling factors to quantized weight parameters and how they can be ordered for good hardware efficiency.

As discussed in Section 2.1.3 a CONV layer, all weights are typically represented as a tensor $\mathbf{W}_l \in \mathbb{R}^{K \times K \times I \times N}$ where K is the filter size, I is the number of input feature maps and N, the number of output feature maps. In low-precision networks, each weight layer l can typically be represented by a diagonal scalar matrix $\boldsymbol{\alpha_l}$ multiplied by quantized weight matrix $\boldsymbol{Q}_l$ and ideally $\boldsymbol{W}_l \approx \boldsymbol{\alpha_l Q}_l$. Also, the activation function g can be approximated using a piecewise constant activation function G. In our proposed method, we observe that by ensuring quantization levels for $\boldsymbol{W}$ are symmetric around zero, we can construct efficient square diagonal matrix representations of $\boldsymbol{\alpha_l}$, which enable fine-grained quantization whilst having minimal memory requirements (of size K or K^2). This translates to a reduction in overall model complexity and high prediction capabilities. Although, we restrict ourselves by structured matrices and low-precision weights and activations, the network efficiently captures information through our gradient-based symmetric quantizer which learns the diagonal elements of $\boldsymbol{\alpha_l}$ during training.

5.3.1 Reducing Information Loss Through Scaling Factors

The introduction of scaling factors improves learning capabilities by providing greater model capacity and compensating for the large information loss due to binary/ternary quantization. Scaling discrete weight representations requires multiplying all $\boldsymbol{Q}_{l_{i,j}}$ by positive scaling factors $\alpha \in \mathbb{R}^+$. We want to find optimal scaling factors for each layer, α_l, which minimize our error function:

$$\alpha_l^* =_\alpha E(\alpha, \boldsymbol{Q}) \quad s.t. \quad \alpha \geq 0,\ \boldsymbol{Q}_{l_{i,j}} \in \mathbb{C} \tag{5.5}$$

with E representing the error function with scaling factors. Finding the optimal α_l is vital to reducing gradient mismatches in the forward and backward functions. It was proposed in [156] as the mean of absolute weight values for each layer:

$$\alpha_l = \frac{\|W_l\|_1}{Z_l} \tag{5.6}$$

where Z_l is the total number of layer weights. The codebook for each layer after scaling in (5.6) is symmetric: $\hat{\mathbb{C}}_l = \big\{-\alpha_l, +\alpha_l\big\}$ and the scalars become per-layer learning rate multipliers. Additionally, the STE in (5.7) reduces the gradient mismatch from (5.4) by including information from the full precision weights:

$$\frac{\partial E}{\partial W_{l_{i,j}}} = \frac{\partial E}{\partial \boldsymbol{Q}_{l_{i,j}}} = \alpha_l \frac{\partial \hat{E}}{\partial \boldsymbol{Q}_{l_{i,j}}} \tag{5.7}$$

Gradient-based optimizations for scaling factors were also introduced in [158] which applied different scaling factors for positive and negative $\boldsymbol{Q}_{l_{i,j}}$ to improve model capacity and accuracies. These are updated during backpropagation using gradients:

$$\frac{\partial E}{\partial \alpha_l^p} = \sum_{i,j \in S_l^p} \frac{\partial E}{\partial W_{l_{i,j}}},\ \frac{\partial E}{\partial \alpha_l^n} = \sum_{i,j \in S_l^n} \frac{\partial E}{\partial W_{l_{i,j}}} \tag{5.8}$$

where initially $\alpha_{l_0}^p, \alpha_{l_0}^n = 1$ and S_l is the codebook indices for each layer, i.e. $S_l^p = \big\{i, j | W_{l_{i,j}} \geq \eta\big\}$ and $S_l^n = \big\{i, j | W_{l_{i,j}} \leq -\eta\big\}$. This allows each layer's codebook values to be asymmetric around zero, such that $\hat{\mathbb{C}}_l = \big\{-\alpha_l^n, +\alpha_l^p\big\}$. The codebook indices are then highly irregular which increases computational complexity as the matrices cannot be easily decomposed. For many existing DNN hardware implementations, we'd have to

check the sign of every element before computation, leading to extra branching instructions for conventional computing platforms such as CPUs/GPUs and additional logic for custom hardware. The difficulty of designing low-precision networks which have both high learning capabilities and computational efficiency can be solved by learning a symmetric codebook during training and exploiting structured matrix representations.

5.4 SYQ Structural Representations

We now propose matrix representations of SYQ by partitioning the quantization into weight subgroups. Diagonal matrix representations consist of mainly zeros and have non-zero entries along the main diagonal. For a matrix $\boldsymbol{D}$ to be diagonal, $\boldsymbol{D} = 0$ if $D_{i,j} = 0 \; \forall \; i \neq j$, and square if $\boldsymbol{D} \in \mathbb{R}^{m \times m}$. A square diagonal matrix consisting of all equal main diagonal entries is a scalar matrix. A diagonal matrix $\boldsymbol{\alpha_l}$ is defined by the vector $\boldsymbol{\alpha_l} = \left[\alpha_l^1, ..., \alpha_l^m\right]$:

$$\boldsymbol{\alpha} = diag(\boldsymbol{\alpha}) := \begin{bmatrix} \alpha^1 & 0 & .. & 0 & 0 \\ 0 & \alpha^2 & .. & : & 0 \\ : & : & .. & \alpha^{m-1} & : \\ 0 & 0 & .. & 0 & \alpha^m \end{bmatrix}$$

Diagonal matrix multiplication is very computationally efficient as it can be easily decomposed and only the scalar vector requires storage.

5.4.1 Layers

CONV and FC layers have differing computational requirements and sensitivities to network redundancies. CONV weights are reused many times across the input feature map whereas FC weights are used only once per image. Hence, the quantization error of each weight in a CONV layer impacts the dot products across the entire input feature map volume rather than just once for FC weights. Thus, a fine-grained approach to CONV layers is effective at compensating for this error. Quantized CONV weights are represented as a tensor $\mathbf{Q}_l \in \mathbb{R}^Z$ with $Z = K \times K \times I \times N$. As typically $I, N \gg K$, it is optimal to have a diagonal scalar of

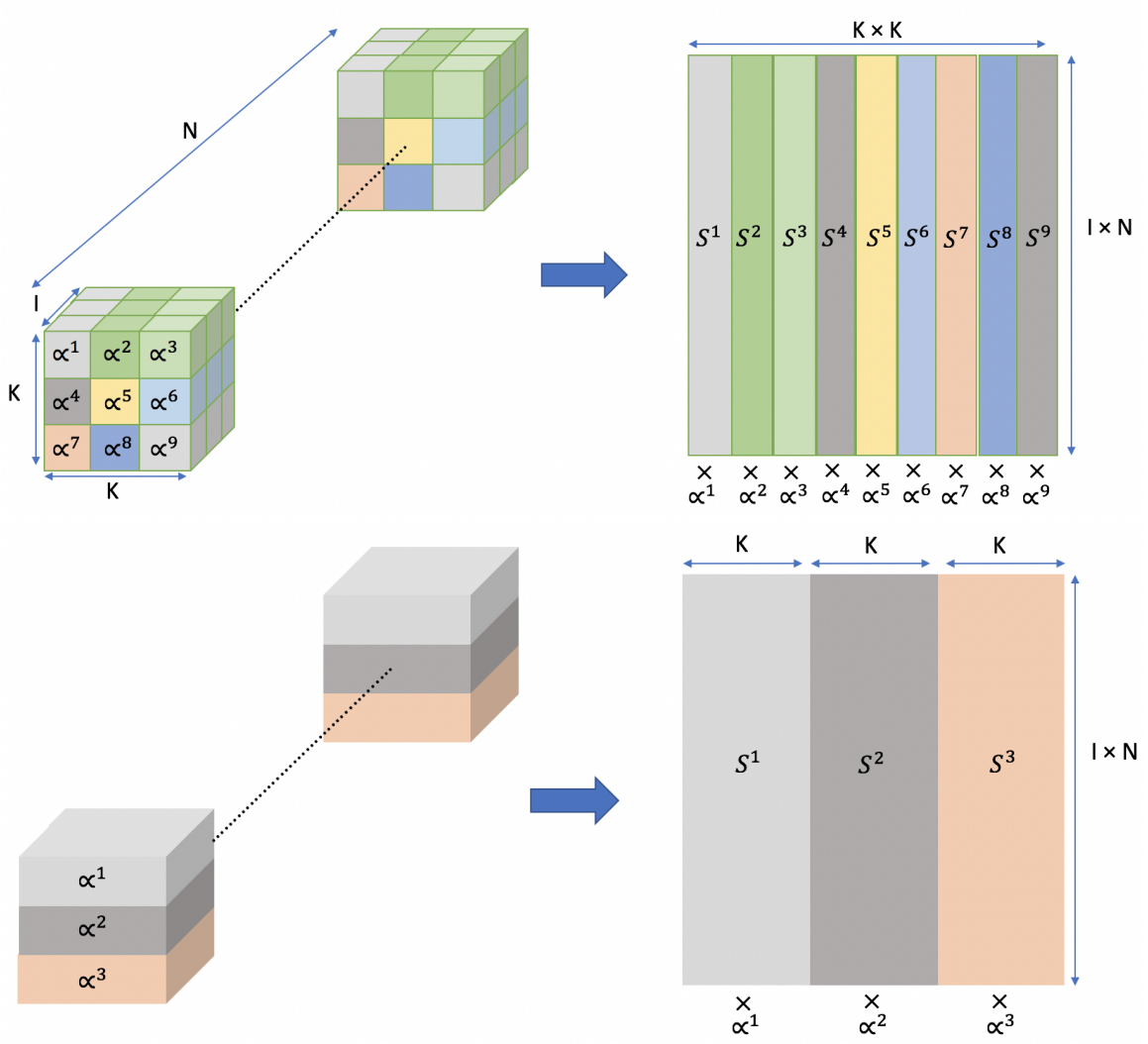


FIGURE 5.1. Computational structure of pixel-wise (Left) and row-wise (Right) subgrouping of a CONV layer (K, I = 3). The tensors represent the weight layer structure during training and the matrices represent the matrix decomposition for deployment.

size $K \times K$ or even $K^2 \times K^2$ as only small scalar vectors are required for storage. By reshaping the tensor $\mathbf{Q}_l$, we form a matrix $\boldsymbol{Q}_l \in \mathbb{R}^{\hat{Z}}$ where $\hat{Z} = K^2 \times (IN)$ or $\hat{Z} = K \times (INK)$ and represent our scalar matrix multiplication as $diag(\boldsymbol{\alpha_l})\boldsymbol{Q}_l^T$ with the square diagonal matrix, $diag(\boldsymbol{\alpha_l}) \in \mathbb{R}^{K^2 \times K^2}$ or $diag(\boldsymbol{\alpha_l}) \in \mathbb{R}^{K \times K}$ respectively. FC layers are represented as a matrix $\boldsymbol{Q}_l \in \mathbb{R}^{L \times H}$ where H is the number of hidden nodes and L the activation neurons. As FC layers are more robust to quantization, one learnable scaling factor (layer-wise) for the FC layer can sufficiently approximate the distribution and also can be represented with scalar

matrix computation. All elements in $\boldsymbol{\alpha_l}$ are then equal and we only require storage of one value.

5.4.2 Subgroups

More fine-grained quantization can improve approximations of the statistical distributions of weights. We implement pixel-wise scaling for CONV layers which involves grouping all spatially equivalent pixels along the $I \times N$ dimension. This results in different values for all the main diagonal elements in $diag(\boldsymbol{\alpha}) \in \mathbb{R}^{K^2 \times K^2}$. With this representation, we can still decompose the matrix computation along each pixel dimension and exploit the parallel nature of convolutions as shown in Figure 5.1. We do this by creating subgroups $1 \leq i \leq K^2$ with codebook indices $S_l^i = \left\{j | W_{l_{i,j}}\right\}$. Other granularities such as row-wise scaling involve grouping all pixels along a row or column ($I \times N \times K$), resulting in $S_l^j = S_l^i \cup S_l^{i+1} ... \cup S_l^K$ where $1 \leq j \leq K$ (as illustrated in Figure 5.1) and also layer-wise scaling: $S_l = S_l^i \cup S_l^{i+1} ... \cup S_l^{K^2}$. Different granularities affect both accuracy and computation as further explored in Sections 6 & 7.

5.5 SYQ Training

In this section, we now describe the methodology to efficiently train SYQ networks.

5.5.1 Symmetric Quantizer

When training low-precision inference networks, the aim is to have the smallest possible codebook. Typically, as the codebook size increases, a network will approach full-precision accuracy but increase hardware cost. However, there are certain codebook representations which are significantly more hardware friendly than others and won't necessarily impose any hardware costs. Given a codebook C, and the nonzero codebooks $C^p = \left\{c_i | c_i > 0\right\}$ and

$C^n = \{c_j | c_j < 0\}$, a quantizer is denoted as symmetric if:

$$\forall c_i \in C^p, \quad \exists\, |c_j| \in C^n \quad \text{where} \quad c_i = |c_j| \tag{5.9}$$

Learning this type of codebook requires updating one scaling factor during training for two bi-polar codebook values. The gradient of each scaling factor for each subgroup becomes:

$$\frac{\partial E}{\partial \alpha_l^i} = \sum_{j \in S_l^i} \frac{\partial E}{\partial W_{l_{i,j}}} \tag{5.10}$$

When computing binary/ternary weight representations followed by a scale, it is ideal to have a codebook which is symmetric around zero, as the codebook storage requirements are almost halved. This is because only the absolute value of the two symmetric values needs to be stored. Additionally, codebook indices become highly regular and ordered for the scalar multiply which greatly reduces computational complexity. The nature of symmetric quantization enables the opportunity to implement fine-grained quantization (pixel/row-wise) whilst maintaining the scalar matrix multiplication structure used in layer-wise scaling. This is also advantageous as the scaling factors become fine-grained adaptive learning rate multipliers for each pixel/row in a CONV layer, i.e. the STE becomes:

$$\frac{\partial E}{\partial W_{l_{i,j}}} = \frac{\partial E}{\partial Q_{l_{i,j}}} = \alpha_l^i \frac{\partial \hat{E}}{\partial Q_{l_{i,j}}} \tag{5.11}$$

As the use of scaling factors can more accurately approximate subgroups and are gradient-based, the gradient mismatch is significantly reduced for weight quantization which enhances network learning.

5.5.2 Initialization

The solution to non-convex functions using gradient descent optimizations depend heavily on parameter initialization to avoid vanishing or exploding activations/gradients and ensure network convergence [42]. For low-precision networks, excessive gradient mismatches between the forward and backward functions must be minimized, otherwise the gradients will not propagate well. To deal with this concern, the scaling factors coefficients are initialized as the mean of full precision weights in it's corresponding subgroup. For example, the scaling

factor in pixel-wise scaling is:

$$\alpha_{l_0}^i = \frac{\sum_{j \in S_l^i} |W_{l_{i,j}}|}{I \times N} \tag{5.12}$$

Layer-wise scaling in FC layers has α_{l_0} as the mean of all layer weights. By incorporating information from the full precision weights, we aim to reduce the mismatch initially and the scaling factors are then optimized during backpropagation.

5.5.3 Activations Quantization

Our forward path approximation to g in (**??**) uniformly quantizes a real number $x \in [0, M]$ to a k-bit number:

$$G(x) = \frac{1}{2^f} floor((2^f)x + \frac{1}{2}) \tag{5.13}$$

where $floor$ represents the round down operation and M is the upper bound. M itself is bounded by its arbitrary unsigned two's complement fixed point representation where f is the number of fractional bits and $M = 2^{k-f} - 2^{-f}$. Uniform quantization translates to a reduction in hardware implementation complexity. To achieve this, we use the following STE for the activations:

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial G} \tag{5.14}$$

Differences in the forward and backward activation functions create a gradient mismatch which can result in unstable and inefficient learning. To minimize this issue, we adjust M as a hyperparameter. The overall SYQ training process is summarized in Algorithm 5.

5.6 Experiments

To demonstrate the versatility of SYQ, we applied it to several state-of-the-art benchmark models, all with different network topologies. We use binary/ternary weights and varying activation bitwidths for classification of the large-scale ImageNet dataset. The ILSVRC-2012 ImageNet is a natural high resolution visual classification dataset consisting of 1000 classes,

Algorithm 4 SYQ Training Summary For DNNs.

Initialize: Set subgrouping granularity for S_l^i and set $\alpha_{l_0}^i$.
Inputs: Minibatch of inputs & targets (I, Y), Error function $E(Y, \hat{Y})$, current weights $\boldsymbol{W_t}$ and learning rate, γ_t
Outputs: Updated $\boldsymbol{W_{t+1}}$, $\boldsymbol{\alpha_{t+1}}$ and γ_{t+1}

SYQ Forward:
for l=1 to L **do**
  $\boldsymbol{Q}_l = sign(\boldsymbol{W}_l) \odot \boldsymbol{M}_l$ **with** η, using (5.2) & (5.3)
  for ith subgroup in lth layer **do**
    Apply α_l^i to S_l^i
  end for
end for
$\hat{Y}$ = **SYQForward** $(I, Y, \boldsymbol{Q}_l, \boldsymbol{\alpha}_l)$ using (6.5)

SYQ Backward:
$\frac{\partial \hat{E}}{\partial \boldsymbol{Q}_l}$ = **WeightBackward**$(\boldsymbol{Q}_l, \boldsymbol{\alpha}_l, \frac{\partial \hat{E}}{\partial \hat{Y}})$ using (5.11) & (5.14)
$\frac{\partial \hat{E}}{\partial \boldsymbol{\alpha}_l}$ = **ScalarBackward**$(\frac{\partial \hat{E}}{\partial \boldsymbol{Q}_l}, \boldsymbol{\alpha}_l, \frac{\partial \hat{E}}{\partial \hat{Y}})$ using (5.10)
$\boldsymbol{W_{t+1}}$ = **UpdateWeights**$(\boldsymbol{W_t}, \frac{\partial \hat{E}}{\partial \boldsymbol{Q}_l}, \gamma)$
$\boldsymbol{\alpha_{t+1}}$ = **UpdateScalars**$(\boldsymbol{\alpha_t}, \frac{\partial \hat{E}}{\partial \boldsymbol{\alpha}_l}, \gamma)$
γ_{t+1} = **UpdateLearningRate**(γ_t, t)

1.28 million training images and 50K validation images. Inputs are resized to 256×256 before being randomly cropped to 224×224. We report our single-crop evaluation results using Top-1 and Top-5 accuracy.

5.6.1 Networks

We compare our results to the full precision baseline and benchmark reference model accuracies in Table 6.9[1], showing that SYQ training achieves similar accuracy to floating point. This suggests the noise induced from replacing floating point weight layers with SYQ versions, provides effective regularization during training. An AlexNet [68] variant is implemented which eliminates dropout and includes batch normalization [66]. A mini batch size of 64 is used, L2 weight decay of 5e-6, and our learning rate is initially 1e-4 with step decays of scale factor 0.2. For ResNet [55], we test on the 18, 34 and 50 layer variations. Our batch size is

[1]Our ResNet and AlexNet reference results are obtained from https://github.com/facebook/fb.resnet.torch and https://github.com/BVLC/caffe, respectively

TABLE 5.1. Summary of Results for 8-bit activations and binary (1-8) and ternary (2-8) weights

Model		1-8	2-8	Baseline	Reference
AlexNet	Top-1	**56.6**	**58.1**	56.6	57.1
	Top-5	**79.4**	**80.8**	80.2	80.2
VGG	Top-1	**66.2**	**68.7**	69.4	-
	Top-5	**87.0**	**88.5**	89.1	-
ResNet-18	Top-1	**62.9**	**67.7**	69.1	69.6
	Top-5	**84.6**	**87.8**	89.0	89.2
ResNet-34	Top-1	**67.0**	**70.8**	71.3	73.3
	Top-5	**87.6**	**89.8**	89.1	91.3
ResNet-50	Top-1	**70.6**	**72.3**	76.0	76.0
	Top-5	**89.6**	**90.9**	93.0	93.0

128, learning rate is initially 1e-3 with step decay of factor 0.2. We also test on a variant of VGG-16 [120], using model-A in [56] with the spp layer replaced by a max pool and only 3 CONV layers rather than 5 for input size blocks of 56, 28 and 14, as in [12]. Batch sizes are set to 32 and our learning rate is initially 1e-4 with a step decay of factor 0.2. The VGG and ResNet models were initialized from floating point baseline weights. Full-precision weights are used for the first and last layer. All other CONV layers are quantized with SYQ pixel-wise scaling, FC layers with layer-wise scaling and the activations of all layers using (6.5).

5.6.2 Changing Granularity Via Weight Subgroups

Weight subgroups can be arbitrarily designed for a given hardware application. Table 5.2 shows accuracy differences between using row/layer-wise vs pixel-wise scaling on AlexNet and suggests pixel-wise and row-wise are marginally different, especially for higher precisions, but both are considerably more accurate than layer-wise. This demonstrates the effectiveness of fine-grained quantization of CONV layers over layer-wise and promotes the exploration for efficient representations of scalar computation. It also shows the effectiveness of row-wise quantization as it typically incurs a smaller memory requirement with a small accuracy drop, for a significant gain in the potential parallelism of the network.

TABLE 5.2. AlexNet accuracy differences between using row/layer-wise and pixel-wise symmetric quantization

		Row-wise		Layer-wise	
Weights	**Act.**	**Top-1**	**Top-5**	**Top-1**	**Top-5**
1	2	-0.7	-0.5	-1.4	-2.2
1	8	-0.1	-0.3	-0.4	-2.2
2	2	+0.1	-0.0	-1.3	-1.5
2	8	-0.1	-0.1	-1.9	-1.7

TABLE 5.3. Comparison to previously published AlexNet results

Model	**Weights**	**Act.**	**Top-1**	**Top-5**
DoReFa-Net [156]	1	2	49.8	-
QNN [63]	1	2	51.0	73.7
HWGQ [12]	1	2	52.7	76.3
SYQ	**1**	**2**	**55.4**	**78.6**
DoReFa-Net [156]	1	4	53.0	-
SYQ	**1**	**4**	**56.2**	**79.4**
BWN [110]	1	32	56.8	79.4
SYQ	**1**	**8**	**56.6**	**79.4**
SYQ	**2**	**2**	**55.8**	**79.2**
FGQ [92]	2	8	49.04	-
TTQ [158]	2	32	57.5	79.7
SYQ	**2**	**8**	**58.1**	**80.8**

5.6.3 Comparisons To Previous Work

We compare SYQ explicitly using AlexNet, ResNet-18 and ResNet-50 in Tables 5.3, 5.4 & 5.5 as they've been extensively studied in the literature. Our ternary results with 8 bit activations (2w-8act) improves on the state-of-the-art for all three networks. Our 2w-4act for ResNet-50 also improves on the state-of-the-art FGQ. This is also the case for binary weights, such as 1w-8act ResNet-18 and AlexNet with 1w-2/4act. For extremely low 1w-2act representations, SYQ also has a 2.7% increase in Top-1 accuracy over the state-of-the-art HWGQ. This demonstrates SYQ's superiority for producing high accuracy. Additionally, it shows that multiple learnable scaling factors effectively reduce the gradient mismatch in the forward and backward paths, translating to efficient learning under low-precision constraints.

TABLE 5.4. Comparison to previously published ResNet-18 results

Model	Weights	Act.	Top-1	Top-5
BWN [110]	1	32	60.8	83.0
SYQ	**1**	**8**	**62.9**	**84.6**
TWN [82]	2	32	65.3	86.2
INQ [155]	2	32	66.0	87.1
TTQ [158]	2	32	66.6	87.2
SYQ	**2**	**8**	**67.7**	**87.8**

TABLE 5.5. Comparison to previously published ResNet-50 results

Model	Weights	Act.	Top-1	Top-5
HWGQ [12]	1	2	64.6	85.9
SYQ	**1**	**4**	**68.8**	**88.7**
SYQ	**1**	**8**	**70.6**	**89.6**
FGQ [92]	2	4	68.4	-
SYQ	**2**	**4**	**70.9**	**90.2**
FGQ [92]	2	8	70.8	-
SYQ	**2**	**8**	**72.3**	**90.9**

FIGURE 5.2. Top-1 training and validation error for binary AlexNet with varying activation precisions

5.6.4 Varying Activation Bitwidth

The most important result is that SYQ efficiently quantizes networks with low-precisions for both weights and activations. From Figure 5.2, we can see that lowering the precision of the activations does not severely alter the training curve, suggesting that the gradient information from pixel-wise scaling factors in SYQ compensates well for the loss of information. However, when quantizing down to 2-bits, the training error curve does become more volatile, demonstrating instabilities in network learning. We also report the classification accuracies for varying activations and bitwidths on AlexNet and ResNet-50 in Tables 5.3 & 5.5, which shows that there is minimal discrepancy from the full-precision networks with as low as 4-bit activations. These results are extremely promising and have strong implications for specialized hardware implementations of low-power DNNs.

TABLE 5.6. Number of scaling factors and operations per layer, for different techniques

Method	Scalars	Ops
Layer (DoReFa)	1	P
Row (SYQ)	K	P
Pixel (SYQ)	K^2	P
Asymmetric (TTQ)	2	$P + Z$
Grouping (FGQ)	$K^2N/4$	P
Channel (HWGQ/BWN)	N	P

5.7 Hardware Implications

In this section we discuss the computational implications of different scaling operations and present a design for specialized hardware implementations.

5.7.1 Computational and Memory Complexity

Considering a CONV layer with Ops, $P = K \times K \times I \times N \times F \times F$, where F is the IFM dimension. The layer-wise scaling, as in DoReFa-Net, requires one scaling factor per P operations. For channel-wise scaling in HWGQ and BWN, it requires N scaling factors as there is one per output feature map, where typically $N \gg 1$. TTQ implements asymmetric layer-wise quantization which requires two scaling factors per layer and $P + Z$ operations as we add a branching operation for each weight due to irregular codebook indices, as described in Section 3.3. FGQ uses pixel-wise scaling for every 4 filters, whereas SYQ uses pixel-wise scaling per N filters, hence it requires $K^2N/4$ scaling factors and P operations. For pixel-wise SYQ scaling, K^2 scaling factors and P operations are required, where $K = 3$ for most CONV layers in modern networks. For row-wise SYQ scaling it requires K scaling factors and P operations. These results are displayed in Table 5.6, demonstrating the benefits of maintaining a diagonal representation for the scalar matrix multiplication of each layer as we either improve computational or memory complexity against all other fine-grained methods. Another key benefit of SYQ is its amenability to highly parallel processors.

5.7.2 Architectural Design

For the CONV layer, the operations are a sum of dot products between the input and kernel filter. In order to reduce compute complexity, we increase the number of operations in each dot product, while significantly decreasing the complexity of each operation. For example, the size of the input vector, in the calculation of each dot product is: $L_v = K^2 I$. The number of operations is $Op^L_{mul} = L_v$ for multiplies and $Op^L_{add} = L_v - 1$ for additions. Given that we have a limited codebook for our weights, we can break it into sub-dot products where we apply the scaling factor, α^i, after we have computed the sub-dot product for that set of symmetrically constrained weights. For pixel-wise quantization, the total multiplies becomes $Op^P_{mul} = L_v + K^2$ and the total adds become $Op^P_{add} = K^2(L_v/K^2 - 1) + (K^2 - 1) = L_v - 1$. However, the first term in each of these calculations can be done at significantly lower precision. For multiplies this means a binary or ternary multiple - which can often be implemented as a bit-flip. To compute this in specialized hardware, for layer-wise scaling, we have a parallel MAC tree which consists of a multiply of an input and binary/ternary number (represented as a dot) followed by an adder tree to sum up the outputs. Outputs of these are fed into a multiplier to compute the scale, followed by an accumulator to store the outputs before being fed into the activation function. This architecture is shown in Figure 5.3. For every hardware block of this type, our per-pixel/row scaling only requires one additional ring counter which stores scaling factors and shifts the input to the scaling multiplier through an index counter as each row/pixel is finished computing which is computationally inexpensive. As in the equivalent layer-wise scaling architecture, we can still maintain one multiplier in hardware and only increase memory slightly to store the scaling factors. Table 5.7 shows the resource and hardware performance estimates provided by Vivado HLS of the described hardware architecture for a target Xilinx ZU3 embedded FPGA device at an estimated clock frequency of over 300 MHz. The main design is based on the MVTU described in FINN [126], with an extension to 2-bit activations and pixel-wise and row-wise SYQ. The layer-wise baseline uses no multiplies, as these can absorb into quantization thresholds for activations [126]. The MVTU was configured for a convolution layer with $I = 384$, $N = 256$, $K = 3$, while scaling the size of the MAC tree (SIMD) and the number of parallel processors (PE). As shown,

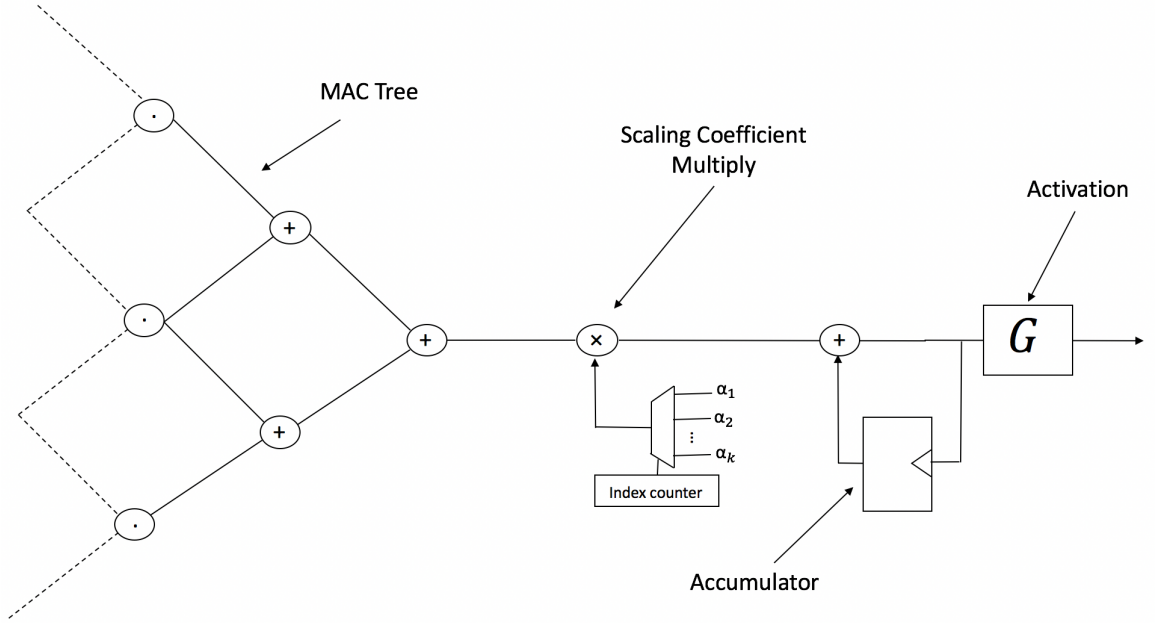


FIGURE 5.3. Hardware description of MAC for SYQ layers

TABLE 5.7. Resource Usage of a Matrix-Vector Processing Unit with Layer-wise and Pixel-wise Quantization for target Xilinx ZU3

Config	SIMD	PE	BRAMs	LUTs (k)	DSPs
Layer	32	32	64	29.8	4
Layer	64	32	64	56.5	4
Layer	32	64	64	58.9	4
SYQ(P)	32	32	64	29.4	36
SYQ(P)	64	32	64	56.1	36
SYQ(P)	32	64	64	57.7	68
ZU3	-	-	432	70.6	360

the BRAM (memory blocks on an FPGA (18k)) and LUT usage is almost identical, while the DSP usage increases proportionally with the number of parallel output channels which are processed. The increase in DSPs is not necessarily costly for the ZU3 as we are able to utilize more of the total available resources. Resource usage is only shown for pixel-wise SYQ, as row-wise only differed in LUT usage by less than 2%. The resource exploration shown here is intended to demonstrate the potential hardware advantage of the proposed SYQ representation. A full implementation verifying improvements in frames per second or latency is beyond the scope of the research.

5.8 Summary

In the previous chapters, we focused on techniques for compressing bitwise networks using sparse representations to improve hardware performance and maintain accuracy. However, achieving higher accuracy with low-precision is pertinent for DNN applications requiring mission-critical decisions, especially on embedded platforms. In this chapter, we explored improving the accuracy of bitwise networks whilst maintaining their hardware simplicity. The SYQ training methodology focuses on designing efficient quantization methods for bitwise neural networks. A representation consisting of fine-grained scaling factors was derived. These scaling factors were learnt via gradient-based optimizations. The arithmetic includes scaling factors such that the product of the scaling factor matrix and quantization matrix maintains regular data access patterns in hardware. This is ensured by ensuring the scaling factor matrix is a diagonal matrix. This helps maximize the potential parallelism of implementing the network in hardware. The training setup is described and results are evaluated on several benchmark networks such as VGG, AlexNet and ResNet on the ImageNet dataset. We then describe an architecture for implementing this type of arithmetic on specialized hardware. This was followed by a resource exploration of the architecture on an FPGA.

CHAPTER 6

Increasing Precision With Low Hardware Cost

In the previous chapter, we discussed how we can alter the numerical representation of our networks to improve accuracy whilst maintaining hardware performance. This was done through the addition of ordered scaling factors to more accurately quantize the network. The improvements from this resulting representation can likely be attained on a broad range of hardware platforms such as CPUs, GPUs, FPGAs and ASICs. In this chapter, we discuss explore a new technique with a similar idea. We aim to improve accuracy further without impinging on the hardware cost. Conventional methods to increase accuracy will maintain the same fixed-point arithmetic and increase precision. This typically comes at at a cost in hardware which is especially not ideal for resource-constrained environments such as embedded platforms. In this Chapter, we increase the precision of our networks, however we design a custom arithmetic using non-uniform quantization as a strategy to minimize the hardware cost. This enables us to optimize the accuracy-hardware tradeoff. To achieve this for DNN applications, the new design must not exceed the device's capacity and the implementation should not significantly degrade accuracy.

More specifically, in this chapter, we firstly design our custom arithmetic method for FPGA hardware using reconfigurable constant coefficient multipliers (RCCMs) which offer a better alternative for saving silicon area than utilizing low-precision arithmetic. RCCMs multiply input values by a restricted choice of coefficients and can be implemented using only adders, subtractors, bit shifts and multiplexers, meaning they can be heavily optimised for FPGAs. We propose a family of RCCMs tailored to FPGA logic elements to ensure their efficient utilization. To minimize information loss from quantization, we then develop novel training techniques which map the possible coefficient representations of the RCCMs to neural

network weight parameter distributions. This enables usage of the RCCMs in hardware, while maintaining high accuracy. In Chapter 5, we introduced the concept of learnable scaling factors. This inspired the learnable scaling factors in this training methodology, however coarse-grained factors in the form of layer-wise are used.

We demonstrate the benefits of these techniques using AlexNet, ResNet-18 and ResNet-50 networks. The resulting implementations achieve up to 50% resource savings over traditional 8-bit quantized networks, translating to significant speedups and power savings. Our RCCM with the lowest resource requirements exceeds 6-bit fixed point accuracy, while all other implementations with RCCMs achieve at least similar accuracy to an 8-bit uniformly quantized design, while achieving significant resource savings.

6.1 AddNet: DNNs Using FPGA-Optimized Multipliers

One limitation with traditional fixed point quantization is that it is uniformly quantized. However, it has been demonstrated that a non-uniform quantization with the same number of potential weights in the codebook can result in better accuracy. By alleviating the uniformity restriction, the codebook may represent the desired full-precision neural network weight distribution with less quantization error [95, 86].

It follows that reducing precision may not be the best method to save silicon area. Reconfigurable constant coefficient multiplications (RCCMs) are an alternative method to reduce FPGA resources through time-multiplexing and resource sharing [23]. They are usually realized using additions, subtractions, bit shifts and multiplexers, meaning multiplies are implemented without requiring digital signal processing (DSP) blocks on an FPGA. However, RCCMs are restricted to a given number of target coefficients; this has restricted their use to digital signal processing application domains including digital filtering and linear transformations, e.g., [54]. We propose a method, AddNet, to design RCCMs with coefficient sets that approximate the desired distribution of neural network weights [31]. Furthermore, we develop a method to train neural networks to take advantage of RCCMs. In doing so, we demonstrate that

using AddNet to optimize neural networks outperforms low precision arithmetic in terms of accuracy for a given silicon area budget.

AddNet consists of the following stages. First, we design a family of RCCMs which are customized to the underlying logic elements on the FPGA. These exhibit very low resource usage and have varying coefficient sets. The RCCM coefficient set whose distribution best replicates the weight distribution of a pre-trained network is chosen and the network is re-trained with weights restricted to these coefficients. This allows the optimizer to update network weight parameters during training while incorporating information about the underlying hardware. This study does not consider the embedded multipliers present in all modern FPGAs; in practical implementations, we envisage different CNN layers using embedded multipliers or our RCCM, depending on resource and throughput requirements.

The trained network is able to learn a representation compatible with the underlying optimized RCCM, achieving both high performance and accuracy. This allows a significant reduction in resource usage for a given throughput, making our designs suitable for resource-constrained implementations. Additionally, we can scale the parallelism of the design to achieve much higher frame rates for similar resource usages. Specifically, our work makes the following contributions:

- A novel family of arithmetic RCCM circuits tailored to the FPGA fabric for neural network applications which significantly reduces resource requirements.
- A distribution matching technique which allows a specific RCCM to be selected based on the required distribution of weights in a CNN, and a training algorithm which finds solutions compatible with the selected RCCM.
- We demonstrate our method achieves significant improvement in accuracy over low-precision (1-6 bit) implementations, and significant reductions in Look-up Table (LUT) usage over 8-bit fixed point precision with no loss in accuracy for state-of-the-art networks such as ResNet [55] implemented in fixed-point. Moreover, weight storage requirements are reduced through implicit weight sharing.

The remainder of the this Chapter is structured as follows: Section 6.2 provides a background to training CNNs and constant coefficient multipliers. In Section 6.3, recent state-of-the-art research on quantization training and hardware architectures for the implementation of CNNs is reviewed. Our methodology for designing our RCCMs is described in Section 6.4. Our training techniques and selection of RCCM is presented in Section 6.5. The hardware architecture used for evaluating the effects of our methods is described in Section 6.6, followed by training and resource usage results in Section 6.7. Finally, we conclude the article in Section 6.8.

6.2 Background

An alternative to reducing the precision of weight parameters, is to use a *weight sharing* approach [13, 137]. Weight sharing involves choosing a finite set of full-precision weights indexed by a codebook. Typically, these weights are chosen to match the desired distribution to reduce information loss, unlike traditional fixed-point quantization where weights are uniformly distributed. Keeping the number of different weights in the codebook small reduces the word size of the indices leading to a small memory footprint. However, weight sharing is normally not applied in FPGA implementations as the weight mapping process introduces additional delays in the critical path of the circuit and requires extra hardware. Furthermore, higher precision arithmetic units also consume more area. For the proposed RCCM, an implicit weight sharing is utilized, reducing coefficient memory without requiring any mapping hardware. Meanwhile our RCCMs are optimized for FPGA hardware meaning they consume less area than fixed-point equivalents.

6.2.1 Small Softcore Multipliers

Due to the low precision requirements of neural networks, efficient implementations of small multipliers recently have gained growing interest [76, 40]. As FPGAs provide embedded multipliers it seems natural to use them. For small multiplications, there is a way to perform two multiplications up to 8×8 bit in a single DSP of typically 18 bit [40]. In case the embedded

multipliers are not sufficient, efficient logic-based (i.e. softcore) multiplier implementations are necessary. The use of radix-4 Booth encoding together with an FPGA mapping that maps both Booth encoder and decoders in the same LUT showed to be the most efficient way to implement softcore multipliers leading to up to 50% resource reductions [134, 73] on Xilinx FPGAs. Unfortunately, they are only this efficient for large word sizes of 16 bit and above. For lower word sizes, Xilinx Coregen showed the best results [73]. An optimization which is particularly suited for small multipliers by re-structuring common multiplier algorithms was recently proposed in [76]. They were optimized for Intel Stratix 10 FPGAs showing the smallest resources and latency. The optimizations described in our work add further constraints designing multipliers which do not allow arbitrary fixed-point number support. This is achieved by applying concepts from reconfigurable multipliers.

6.3 Related Work

As discussed in Chapters 3, 4 & 5 quantization methods for neural networks have been explored with the aim of achieving efficient inference in hardware. An efficient way of training networks with different forward and backward function was introduced in [9]. This led to new derivations of uniform quantization functions for low-precision neural networks in [19, 156]. In Chapter 5, SYQ [34] further explored the importance of initializations and designing a quantization function which reduces the forward and backward mismatch. State-of-the-art accuracies were achieved under low-precision weights and activations. This inspired the derivation of the distribution matching initialization method for efficient quantization. Effective non-uniform quantization forms were also explored in the form of log representations [95]. This form can also compute multiplierless multiply-accumulates (MACs), however the distribution of the representations is restricted to the log domain.

There have been several accelerator architecture designs for low-precision CNNs with uniform quantization arithmetic. Recent literature includes commercial architectures [1, 138] and also academic approaches [25, 53, 130, 2, 46]. The benefits, in terms of power and throughput, of

fitting a design on-chip was described in [127]. Other FPGA architectures have been implemented to utilize the highly amenable nature of CNNs which constrain weight parameters to be only binary or ternary representations [101, 132]. With restrictions in the efficiency of both software and hardware implementations of neural networks, software-hardware co-design is considered an effective approach to achieve optimal performance [75], [144]. A method for designing a quantization function for both increasing accuracy of binarized CNNs while maintaining efficient multiplierless hardware was proposed in [41]. Additionally, an efficient LSTM implementation in [51] utilized load-balance-aware pruning to achieve both network compression and high hardware utilization. Similarly, training highly sparse ternary networks and designing efficient CNN hardware for exploitation was described in [132]. To the best of our knowledge, AddNet is the first quantization scheme which embeds reconfigurability directly into its representations.

6.4 AddNet Reconfigurable Multipliers

In this section, we introduce reconfigurable multipliers and describe their design in AddNet.

6.4.1 Reconfigurable Multipliers

A constant coefficient multiplier is a circuit which computes $y = cx$, using only additions, subtractions and bit shifts, where c is some pre-defined number. For example, to compute $y = 6x$ in terms of additions and shifts, we can use

$$(x << 2) + (x << 1) = 6x \tag{6.1}$$

The "$<<$" operator represents an arithmetic left shift.

An RCCM is a circuit which computes $y = c_s x$ where c_s is an element from a discrete coefficient set $C = \{c_0, c_1, \ldots, c_{N-1}\}$, chosen from a $\lceil \log_2(N) \rceil$ bit select signal s [125]. RCCMs are usually realized using additions, subtractions, bit shifts and multiplexers (MUXes). Previous work has shown potential for reducing resource usage compared to a generic multiplier, especially for small values of N [23, 125, 97, 98].

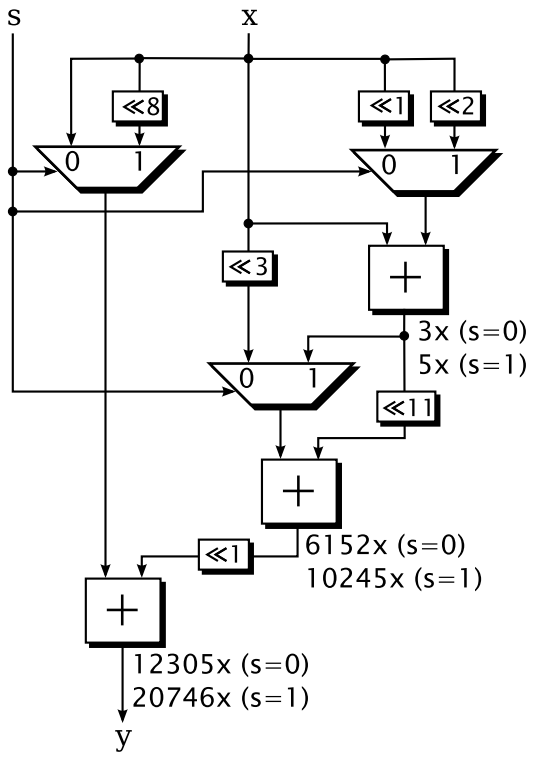


FIGURE 6.1. Example of a reconfigurable multiplier with the coefficient set $\{12305, 20746\}$

figure 6.1 shows an example of an RCCM with coefficient set $C = \{12305, 20746\}$. In this example, there is one 2:1 multiplexer for each adder, each having s as the select line input. The three adders sum various shifted versions of x. Each adder is assigned a coefficient set where the value of each row corresponds to the multiple for each configuration. For instance, the top-most adder computes:

$$\begin{cases} x + (x << 1) = 3x & \text{if} \quad s = 0 \\ x + (x << 2) = 5x & \text{if} \quad s = 1 \end{cases}$$

The bottom-most adder outputs the final output y with coefficient set $C = \{c_0, c_1\} = \{12305, 20746\}$ multiplier-less by:

$$y = \begin{cases} x + ((x << 3) + ((x + x << 1) & \text{if} \quad s = 0 \\ << 11) << 1) = 12305x & \\ (x << 8) + ((x + x << 2) + & \text{if} \quad s = 1 \\ ((x + x << 2) << 11) << 1) = 20746x & \end{cases}$$

By utilizing multiplexers in this way, the computation of $c_1 = 12305$ is able to reuse the adders from computing $c_2 = 20746$, and vice-versa.

To date, prior research with RCCMs has focused on the design of an RCCM for a predefined set of target constants (e.g., obtained from a digital filter design). This design using minimal resources is an NP-complete optimization problem [97]. However, we want to use RCCMs in neural networks where the coefficients (weights) are not known in advance. As a result, we invert the RCCM design, and instead of searching for an RCCM circuit for a given coefficient set, this work aims to find one with very low resource usage and a maximum of "useful" coefficients. This low-cost RCCM then replaces multipliers in a conventional CNN implementation. Instead of storing the coefficients, the corresponding select values are stored, which also has the side-effect that it requires fewer bits of storage than the direct coefficient value.

6.4.2 FPGA Multiplier Mapping

We searched for building blocks that efficiently map to the logic fabric of an FPGA. Our designs are optimized for the latest Xilinx FPGAs (Virtex 5+6, the 7th generation FPGAs and UltraScale/UltraScale+ FPGAs), but similar circuits can be found for other FPGAs. For these devices, a slice provides either 6-input LUTs with a single output (used in Topology A) or two 5-input LUTs with shared inputs (used in Topology B) (Refer to Section 2.6.2 for definitions of *LUTs* and *slice*). As such, we designed our base topologies to ensure the MUXes fit into the same LUTs that are required for the adders.

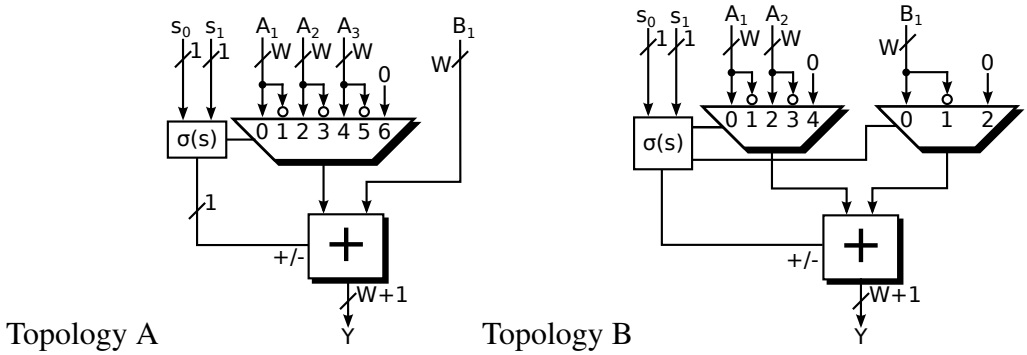
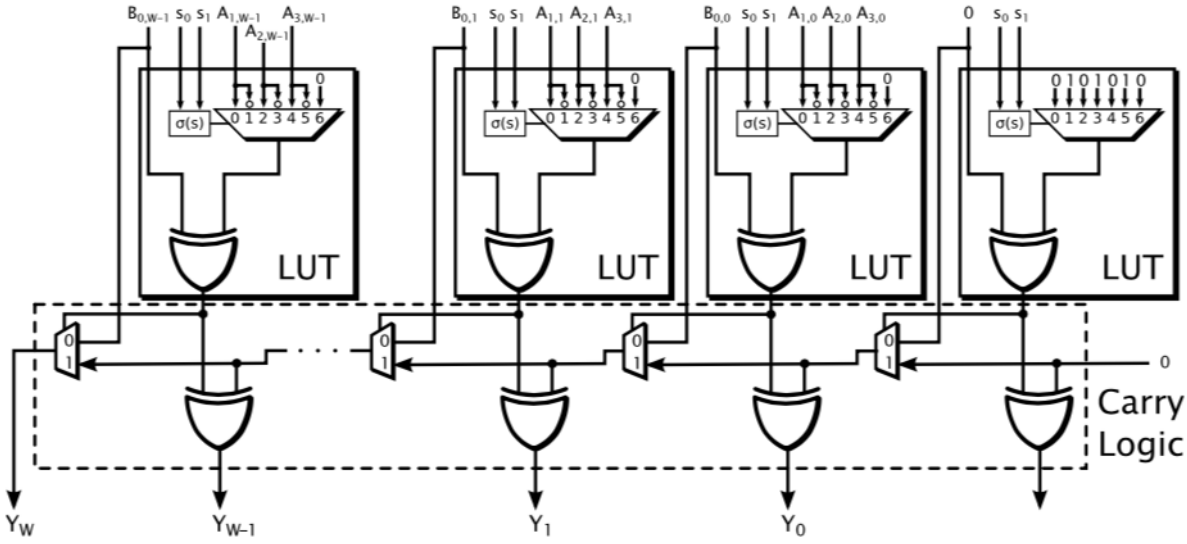


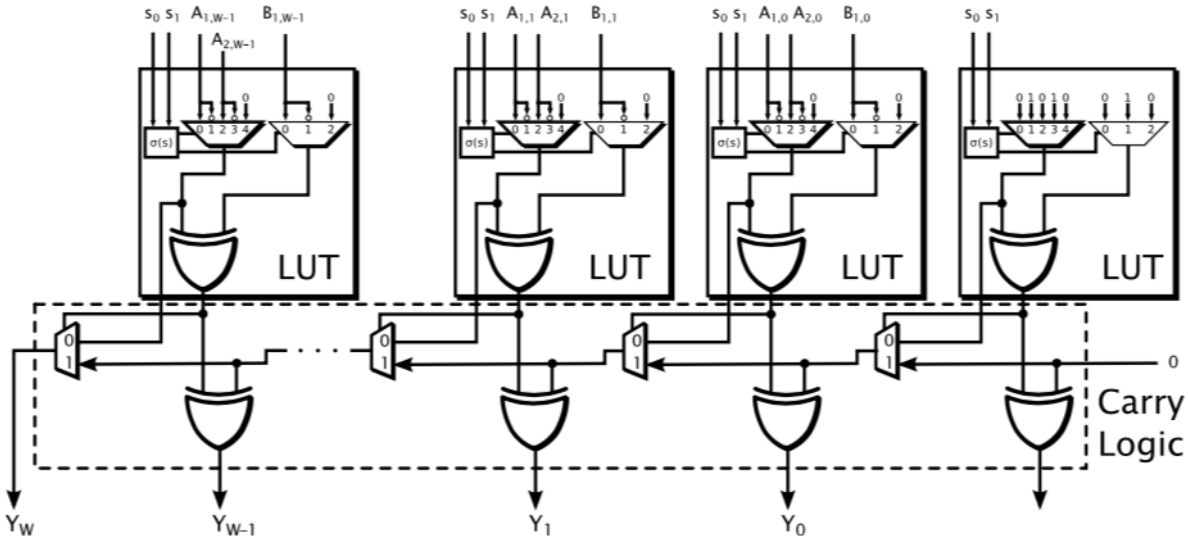
FIGURE 6.2. Base topologies used to build reconfigurable multipliers

figure 6.2 shows the two *base topologies* used to build the RCCM units in this work. Each of these consists of an adder with at least one input being the output of a MUX. These topologies allow operations of the form $\pm A_p \pm B_q$. For Topology A, A_p can consist of up to four different input values $(p \in 1, .., 4)$ with $A_4 = 0$ and B_q can only take one value, $(q = 1)$. For Topology B, $p \in 1, .., 3$ with $A_3 = 0$ and $q \in 1, 2$, with $B_2 = 0$. The sign and source signals are selected using a 2-bit input signal s. Since there are more possibilities than MUX inputs, a function $\sigma(s)$ is used to choose the actual operation, where $\sigma(s)$ is determined at design time, but may be different for each individual RCCM. Note that there is another possibility to map more input sources to the adder as described in [96], however to ensure the topology fits into a single LUT, this comes at a cost of less select inputs. Through our experimentation, we found that the chosen topologies were sufficient for creating RCCMs with a desired coefficient set to simplify the training process. This is further described in Section 6.5.

All contemporary FPGA devices are similar in that their logic blocks consist of LUTs followed by a fast carry chain. Hence, a simple adder can be extended by multiplexers with no additional cost for certain multiplexer sizes when carefully selected for the target device. The detailed slice mappings of our base topologies are shown in figure 6.3, highlighting how our design consumes exactly the same silicon area as a traditional ripple-carry adder with the same word size on that FPGA (which would only implement the XOR gate to complete the carry logic to a full adder).



(a) Topology A



(b) Topology B

FIGURE 6.3. Bit level FPGA slice mapping of base topologies of figure 6.2. This is applicable to any FPGA using 6-input LUTs, including Xilinx Ultrascale and Intel Stratix X devices

.

6.4.3 Architectures Considered

The base topologies described above can be combined in many ways to design RCCM units. Topology A has the advantage of a potentially larger coefficient set as it allows three different sources at input A_p. On the other hand, Topology B has the property that input B_q can be negated or zeroed, which provides symmetric coefficients around zero (as

TABLE 6.1. Properties of the evaluation of the proposed RCCM units with maximum possible set size $S = 2^{w_s}$

RCCM	w_s	#unique coefficient sets
2-Add	4	1145
3-Add	6	44198
4-Add	8	4040952

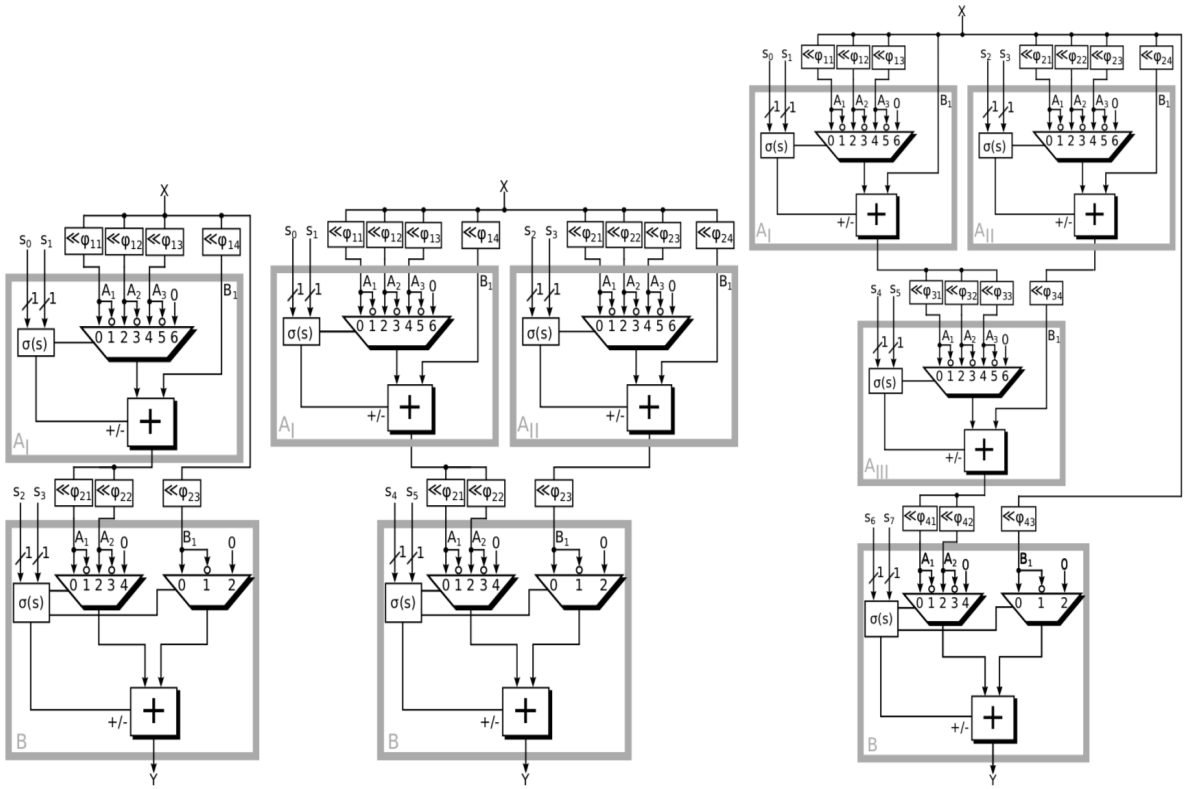


FIGURE 6.4. Selected RCCM circuits

$A_p - B_1 = -(-A_p + B_1)$). We designed three different RCCM architectures from these topologies shown in figure 6.4. These consist of one to three elements of Topology A in the early stages and Topology B at the output stage to ensure symmetric coefficients. Symmetric coefficients improve the ability to match the distribution of the coefficient sets to the pre-trained neural network weights which are typically also approximately symmetric around zero. The benefits of this are further discussed in Section 6.5. Note also that these designs can be trivially pipelined.

As can be seen in figure 6.4, the A_p inputs to the topologies are all connected to left shift operations φ_{ij}, which are all hard-wired since these do not require any LUT resources. It follows that the supported coefficient set depends on the operation mapping function $\sigma(s)$ and the fixed bit shifts φ_{ij}. As mentioned in Section 6.4.2, each instance of base Topology A or B consumes the same area as a traditional ripple-carry adder. Hence, as the RCCMs of figure 6.4 consists of 2, 3 and 4 base topologies, they are, respectively, called 2-Add, 3-Add and 4-Add RCCMs in the following.

The obtained RCCM architectures can multiply with up to 2^{w_s} different coefficients where w_s denotes the total number of bits used for the select signal. For the 2-Add, 3-Add and 4-Add RCCMs, this translates to $w_s = 4, 6$ and 8 respectively, as can be seen in figure 6.4. We chose to evaluate coefficient sets where Topology A had 4 different mapping functions and Topology B a single one. In addition all maximum bit shifts were set to $\varphi_{\text{max}} = 3$. This limits the total number of unique combinations, as shown in table 6.1. With these coefficient sets, an exhaustive enumeration of possible coefficient combinations is feasible with a few minutes of computation time. This allows us to then find the desired coefficient set based on its similarity to the pre-trained neural network weight distribution. We note that it may be possible to improve on our results by exploring more mapping functions, which would generate a larger number of unique coefficient sets, but at the cost of longer execution time.

6.5 AddNet Training

The previous section described a family of optimized multipliers. In this section, we now address the issue of finding the best coefficient set for a given neural network. As discussed in Chapter 1, neural networks can typically tolerate a certain amount of regularization for their weight representations before the accuracy is impinged upon. Thus our strategy is to utilize this knowledge and select an RCCM coefficient set which exhibits a distribution similar to the distributions of the neural network weights and re-train the network to learn the representation of the coefficient set.

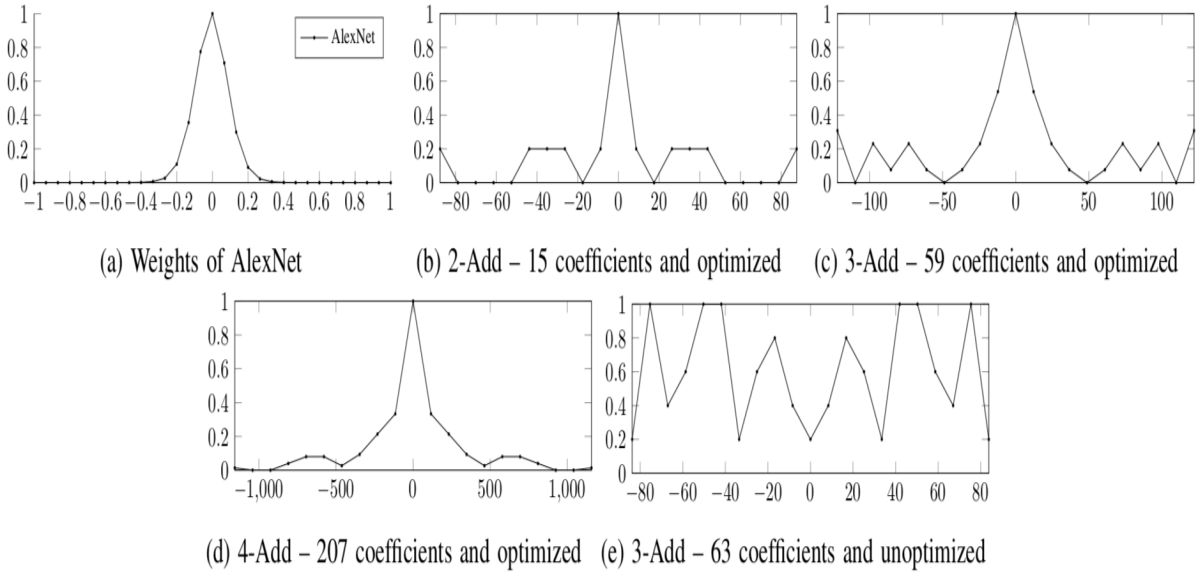


FIGURE 6.5. Distribution for CNN weights and constant multiplier coefficients

6.5.1 Distribution Matching

To achieve high accuracy in quantized neural network training, it is important to reduce quantization error by using a function which can efficiently map its representations to the full-precision values. This is important to minimize information loss and to achieve a good initialization for training [34]. Fixed-point representations using quantization typically uniformly partition the weight parameter space. However, representations using RCCM coefficient sets discussed in Section 6.4 are non-uniformly partitioned and can vary in size, range and the nature of the distribution. Thus, for efficient training, we choose an RCCM with a coefficient set to match the distribution of a pre-trained model. We use the Kullback-Leibler divergence [71] as a measure of the similarity of two distributions. Let R denote the distribution of the coefficient set of the RCCM, and P is the reference distribution of the pre-trained model weights and N the total number of weights. The Kullback-Leibler divergence D_{KL} is defined as

$$D_{\mathrm{KL}}(P\|R) = \sum_{i=0}^{N-1} P(i)\, \log \frac{P(i)}{R(i)}\,. \tag{6.2}$$

Thus, for each enumeration of the coefficient sets, we measured the divergence D_{KL} to the pre-trained network weights and selected the top-5 sets with the smallest divergence. We call

TABLE 6.2. Optimized RCCM coefficients

arch.	#coeff	Coefficient set (±)
2-Add	15	0 1 2 8 28 36 44 92
3-Add	59	0 1 2 3 4 5 6 7 9 10 12 13 14 16 23 29 30 32 63 69 70 72 87 93 94 96 119 125 126 128
4-Add	207	0 1 2 4 5 7 8 9 11 13 14 15 16 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 36 37 38 39 40 46 48 54 58 64 69 70 71 74 75 76 78 80 81 82 84 85 87 94 96 102 114 118 126 134 142 150 166 174 182 190 194 198 206 214 222 230 238 246 258 262 270 278 286 302 310 318 326 334 382 398 446 450 526 566 574 582 614 622 654 662 670 686 694 710 766 782 830 1214

this technique *distribution matching*. From the top-5 sets, we selected the set with the largest number of coefficients, to maximize the number of representable states for the weights during re-training. As a secondary criteria, we only selected coefficient sets that include zero. Note that a zero weight could alternatively also be realized by resetting the output flip-flop in a pipelined implementation. Since this leads to an additional select-bit, that has to be stored in the coefficient memory or a separate decoder, this was not further investigated.

To give an example, the weight distribution (using 31 bins) from AlexNet on ImageNet is given in figure 6.5 (a). As shown, the weight parameters in this example follow a distribution similar to a Gaussian distribution, meaning that small weight values near zero occur much more often than large values. The coefficient sets of the RCCM circuits of figure 6.4 with the best distribution matching are given in table 6.2. Their distributions are shown in figure 6.5 (b), (c) and (d), which are similar to the pre-trained model. We call these optimized 2-Add, 3-Add and 4-Add RCCM circuits.

The exhaustive search for coefficient combinations yields distributions of different nature, meaning this method would most likely be able to efficiently map to other potential network

TABLE 6.3. Configuration parameters of the the RCCM units

RCCM	type	s_1 s_0 00	01	10	11	shifts φ_{i1}	φ_{i2}	φ_{i3}	φ_{i4}
2-Add	A_I	$A1+B1$	$A2+B1$	$A3+B1$	$B1$	0	1	3	2
	B	$-A1+B1$	$-A2+B1$	$A1-B1$	$A2-B1$	0	3	2	–
3-Add	A_I	$A1+B1$	$A2+B1$	$A3+B1$	$-A2+B1$	0	2	3	3
	A_{II}	$A1+B1$	$A2+B1$	$A3+B1$	$-A1+B1$	0	1	3	0
	B	$-A1+B1$	$-A2+B1$	$A1-B1$	$A2-B1$	0	3	0	–
4-Add	A_I	$A1+B1$	$A2+B1$	$A3+B1$	$-A3+B1$	0	1	3	0
	A_{II}	$A1+B1$	$A2+B1$	$A3+B1$	$-A2+B1$	0	1	3	1
	A_{III}	$A1+B1$	$A2+B1$	$A3+B1$	$-A1+B1$	0	1	3	3
	B	$-A1+B1$	$-A2+B1$	$A1-B1$	$A2-B1$	0	3	1	–

TABLE 6.4. Accuracy change from optimized distribution matching on AlexNet for the 2-Add case

	Unoptimized	Distribution Matching	32bit Float.
Top-1	53.8%	**55.8%**	55.1%
Top-5	76.9%	**79.8%**	79.2%

weight distributions. To further justify our approach of distribution matching, we also study an RCCM with an unoptimized choice of coefficient set with differing distribution nature. figure 6.5 (e) shows the distribution with the worst (i.e., largest D_{KL}) divergence score for 63 coefficients. It is not obvious that the corresponding coefficient set, $C = \{0\ 8\ 12\ 14\ 16\ 18\ 20\ 21\ 23\ 24\ 36\ 38\ 40\ 42\ 44\ 45\ 47\ 49\ 51\ 52\ 54\ 56\ 58\ 60\ 68\ 70\ 72\ 74\ 76\ 77\ 79\ 88\}$, would lead to poor CNN inference accuracy. However, as shown in table 6.4, when used with AlexNet [70] in the 2-Add case, Top-1/Top-5 accuracy is 53.8%/76.9% (results are presented as a Top-k percentage, where a classification is considered correct if the actual class is among the highest k probabilities). With distribution matching, the accuracy is 55.8%/79.8%, which is significantly better than the unoptimized set and equivalent to full-precision floating point accuracy.

6.5.2 Weight Quantization

To both exploit our RCCM and achieve high accuracy, our network should be trained to match the underlying inference hardware. During our fixed-point training, for each layer l, we firstly clip the weights so that $w_l \in (-M, M)$, where M is a range hyperparameter, at each inference step and then quantize them to fixed-point representations. As discussed in Section 6.4 our multiplier consists of a fixed point input and a value from C. During AddNet training, we introduce a function whereby every floating point weight is quantized according to

$$q(w_l) = \arg\min_{c_i \in C'} |c_i - |w_l|| \tag{6.3}$$

where $c_i \in C'$ represents the possible positive coefficients of C scaled by λ_l. Here, (6.3) aims to minimize the quantization error between the quantized weight values and the representations in our coefficient set. The scaling with λ_l is done so that $q_i \in (-M, M)$ where M is initially the range of the pre-trained model. By using *distribution matching*, we minimize this quantization error to achieve an efficient initialization. We then re-train the network using the straight through estimator (STE) approach as described in [9]. This approach allows a non-differentiable function defined in the forward path to use a non-zero surrogate derivative function in the backward path gradient calculations. Thus, in our case, we allow:

$$\frac{\partial L}{\partial q} = \frac{\partial L}{\partial w} \tag{6.4}$$

where L is the loss function. The quantized weights $q(w_l)$ are used for inference in the forward path and the floating point weights w_l are updated in the backward path. During training, λ_l becomes a parameter which is also updated during backpropagation. By using a representation compatible with the multiplier in the forward path, the network learns a representation both high in accuracy and hardware efficiency. After training, the floating point weights are discarded and $q(w_l)$ is used for hardware deployment.

Algorithm 5 Training a CNN using AddNet representations

Initialize: Pre-train model
Set adder size
c = DistributionMatching($\sigma(s)$) using (6.2)
Inputs: Minibatch of inputs & targets (I, Y), Loss function $L(Y, \hat{Y})$, current weights $\boldsymbol{W_t}$ and learning rate, γ_t
Outputs: Updated $\boldsymbol{W_{t+1}}$, $\boldsymbol{\lambda_{t+1}}$ and γ_{t+1}

Forward propagation:
for l=1 to L **do**
  $\boldsymbol{Q}_l$ = Quantize($\boldsymbol{W}_l$) using (6.3) and (6.5)
end for
$\hat{Y}$ = **ForwardPropagation** $(I, Y, \boldsymbol{Q}_l)$ using (6.5)

Backward Propagation:
$\frac{\partial \hat{L}}{\partial \boldsymbol{Q}_l}$ = **WeightBackward**($\boldsymbol{Q}_l$, $\frac{\partial \hat{L}}{\partial \hat{Y}}$)
$\frac{\partial \hat{L}}{\partial \boldsymbol{\lambda}_l}$ = **ScalarBackward**($\frac{\partial \hat{L}}{\partial \boldsymbol{Q}_l}$, $\boldsymbol{\lambda}_l$, $\frac{\partial \hat{L}}{\partial \hat{Y}}$)
$\boldsymbol{W_{t+1}}$ = **UpdateWeights**($\boldsymbol{W_t}$, $\frac{\partial \hat{L}}{\partial \boldsymbol{Q}_l}$, γ)
$\boldsymbol{\lambda_{t+1}}$ = **UpdateScalars**($\boldsymbol{\lambda_t}$, $\frac{\partial \hat{L}}{\partial \boldsymbol{\lambda}_l}$, γ)
γ_{t+1} = **UpdateLearningRate**(γ_t, t)

6.5.3 Activation Quantization

As initial training results did not show accuracy degradations compared to activations larger than 8-bit two's complement, we first uniformly quantize the activations to 8-bit. We also selected the input word size of the RCCM accordingly. In the forward path, we approximate the function g in (**??**) with G, which uniformly quantizes a real number $x \in [0, m]$ to a k-bit number:

$$G(x) = \frac{1}{2^f} \left\lfloor 2^f x + \frac{1}{2} \right\rfloor \tag{6.5}$$

where $\lfloor \cdot \rfloor$ returns the greatest integer less than or equal to the argument and m is the upper bound. m itself is bounded by its arbitrary unsigned two's complement fixed point representation where f is the number of fractional bits and hence $m = 2^{k-f} - 2^{-f}$. A summary of the training process is given in Algorithm 5, which is similar to references [18, 127], with the addition of distribution matching and incorporating the quantization scheme of (6.3).

6.6 Experimental Setup

In this section, we present the system used to evaluate the benefits of our AddNet optimizations. We implemented the circuits in Figures 6.2, 6.3 and 6.4 in the hardware description language (HDL) VHDL, as they were more naturally described in a HDL than using other high level synthesis tools. We then chose to integrate it into the open source FPGA CNN Library by Alpha Data in VHDL [3], which provides basic neural network layers for generating custom 8-bit fixed point CNN implementations. We instantiate the multiplier in replace of the traditional VHDL fixed point multiplication used in the original Alpha Data source code. This is used as our hardware library.

The bitstream generation workflow is illustrated in figure 6.6. After defining the CNN architecture and pre-training the network, the RCCM coefficients are calculated using distribution matching. The user provides this information to our AddNet tensorflow software library which then trains the network for a specified adder size. Once trained, the weights are written to a file. These weights, along with the parallelism factors and architectural preferences are provided to the hardware library. The bitstream is then generated using Vivado 2018.1 with both the Peripheral Component Interconnect Express (PCIe) interface and Network Accelerator Core which are downloaded onto the FPGA. We tested both our tensorflow inference and hardware accelerator output to ensure correctness of the design.

6.6.1 System Overview

The Network Accelerator core is integrated with a PCIe interface as illustrated in Figure 6.7 (a) which uses a streaming approach to direct memory access (DMA) data at an efficient rate across the PCIe bus and back. The design is targeted to the Alpha Data ADM-PCIe-8K5 board with a Xilinx KU115 FPGA, which consists of 2160 BRAMs (36K), 5520 DSPs and 663,000 LUTs. A board-specific PCIe Alpha Data IP core is used to interface between PCIe and our network accelerator core. This IP core can be configured to provide and consume AXI4 DMA streams of width 256 bits at a clock rate of 250 MHz in response to API function calls from the host. This stream width is reduced to match the buffer sizes for the inputs (24 bits)

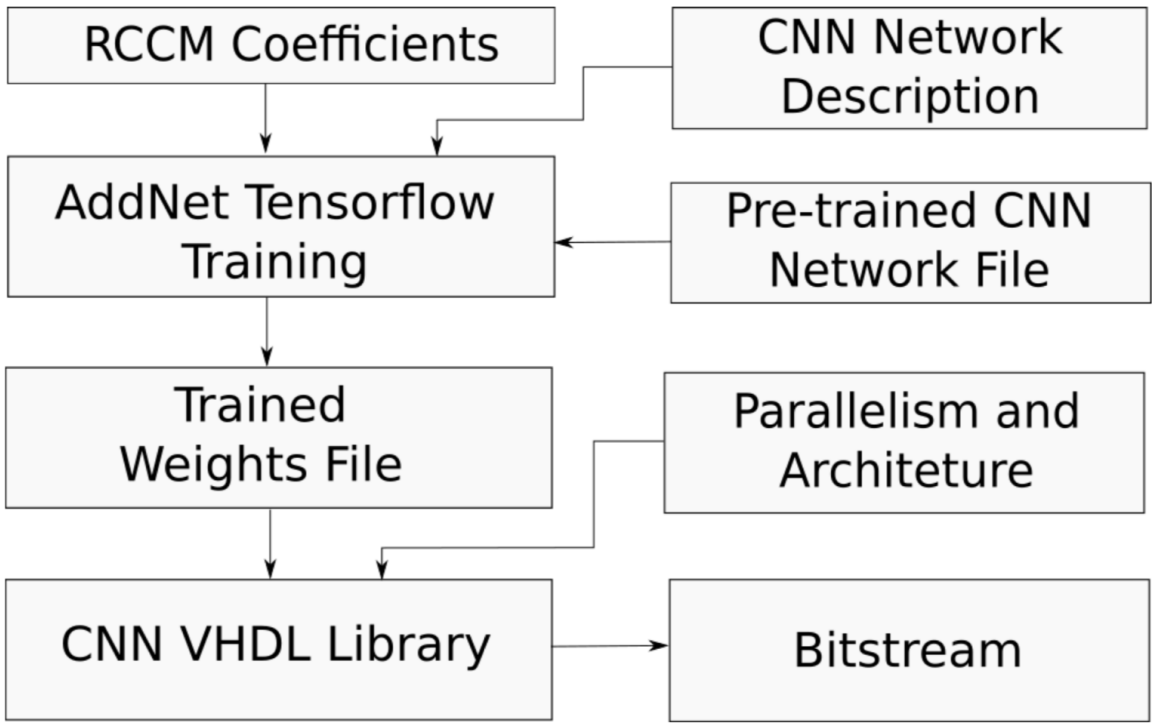
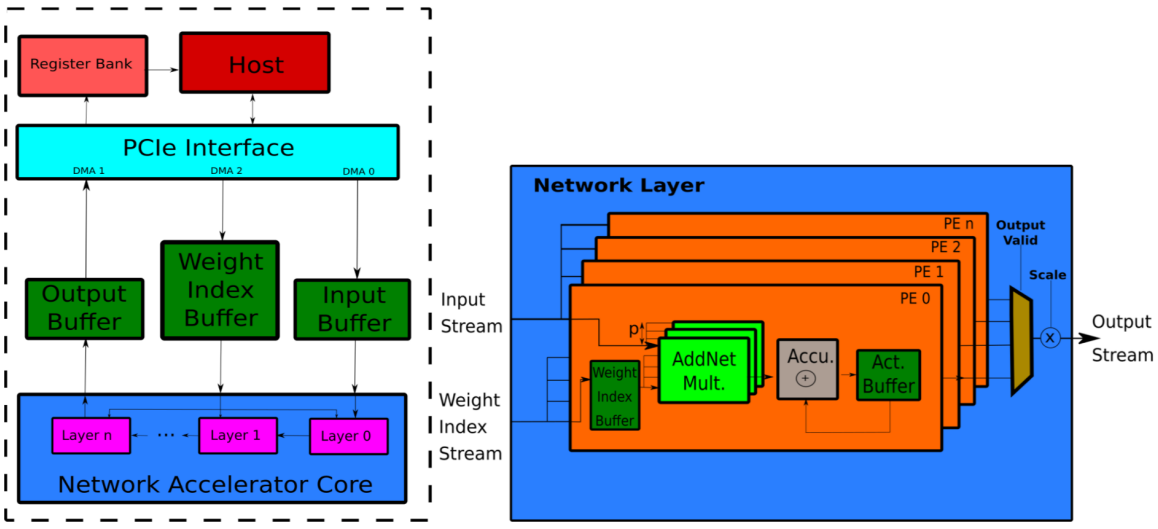


FIGURE 6.6. Bitstream generation design flow



(a) AlphaData Library System Design

(b) Network Layer Accelerator Core including AddNet multipliers

FIGURE 6.7. Hardware Accelerator System Design

and weights ($4 \ldots 8$ bits) which control data ingress. The weight data is sent in contiguous bursts to each layer in the Network Accelerator core to match the expected input behavior. DMA channel 0 is used to provide the input data for layer 0 and weights are initialized on DMA channel 1. The layer output is sent back over PCIe using a separate DMA channel.

Additionally, a memory mapped direct slave port is used to access a bank of registers which can be read by the host to measure performance.

6.6.2 Network Layer Accelerator Core

The Network Layer Accelerator Core performs the MAC operations in parallel to compute the convolution as in equation (2.10). The core receives input from the feature and weight buffers and writes to the output buffer. The feature and weight buffers stream data into a serial to parallel converter to fan-out the data to n parallel Processing Elements (PEs). This is shown in Figure 6.7 (a). Once all data reaches the PE, up to p multiplications between features and weights are performed in parallel, and the results accumulated. Here, we replace the standard 8-bit multiplier with our AddNet constant coefficient multiplier described in Section 4 to reduce the cost per MAC over fixed point implementations. The data is then accumulated before being fed into a ReLU activation function. The output then fans-in via a parallel to serial converter before being streamed out of the current layer and into the subsequent layer. After fanning in, the feature stream data is multiplied by an 8-bit scaling constant λ_l, which is pre-computed. The number of multipliers is significantly larger than the number of layers in neural network designs. Hence, although we add one additional scale operation per layer, it only constitutes a tiny proportion of the overall area in comparison to high precision architectures.

6.6.3 Architectures

To quantify the benefits of the AddNet optimizations, we use two different architectures with 2-Add, 3-Add and 4-Add RCCMs, as well as traditional 8-bit fixed point. The architectures we study are a single layer CNN accelerator and a full AlexNet-variant network [70] which reduces the filter size in the first layer to 7x7 and changes the stride of the first and second layers to 2. The single layer implementation represents a loopback architecture. To implement a full network using this architecture, data is sent between the host and FPGA after each layer is computed sequentially. To minimize the amount of loopback iterations, we instantiate 2048

PEs as this equates to the number of neurons in the largest layer for all our networks. As such, we can compute all layer output feature maps for any of our networks during each loopback iteration. The AlexNet implementation represents a full dataflow where all convolutional layers are processed on the FPGA. For all AlexNet implementations using RCCMs, the first layer uses the 4-Add RCCM. This was because a higher number of coefficients was required in the first layer to achieve higher accuracy.

Both architectures were developed by AlphaData, we have not added any optimizations aside from our arithmetic operators. This enables us to focus on the benefits of our optimizations; we believe AddNet could improve on any 8-bit fixed-point deep learning circuit.

6.6.4 Memory Use

One important advantage of the RCCM designs is the reduction in the number of coefficients required for storage. Instead of storing the coefficients c_s, only the index s has to be stored. This is similar to the weight sharing approach. However, no decoder circuit is necessary to realize the codebook as this is implicitly done by the proposed RCCM. While the coefficients in table 6.2 would require 8, 10 and 12 bit to represent in two's complement, storing the index requires only 4, 6 and 8 bits for the 2-Add, 3-Add and 4-Add, respectively. So, significant savings in storage and memory bandwidth are possible for the 2-Add and 3-Add cases. For the Kintex Ultrascale devices, BRAMs can be a 36K unit or 18K units. As the number of 36K BRAMs are reported, the weight buffers at each PE are calculated as 0.5 to represent 18k BRAMs or 1 to represent 36K BRAMs.

6.7 Results

We now display various hardware utilization and accuracy results to demonstrate applicability in neural network computation. The hardware results were obtained after place and route (PAR) using the Vivado 2018.1 design tool.

6.7.1 Reconfigurable Multiplier Resources

First, we made a comparison of the resource usage of our proposed RCCM compared to a generic multiplier. As generic multiplier, we selected the native Xilinx multiplier as it showed the best results for low word sizes [73]. figure 6.8 shows the LUT resources for varying input (activation) word sizes w_{in} from 3 to 16 bits. While the generic multiplier grows at about 7.4 LUTs/bit, the proposed 2-Add, 3-Add and 4-Add RCCMs only grow at 2, 3 and 4 LUTs/bit, respectively. It can be seen that the 2-Add and 3-Add RCCMs always outperform the generic multiplier for $w_{\text{in}} > 4$ bit while the 4-Add RCCM is only interesting for larger word sizes of $w_{\text{in}} > 9$ bits. For the considered 9 bit activation, 55.2% and 32.8% of the LUTs can be saved by using the 2-Add and 3-Add RCCMs. This improves further as we increase the activation precision, suggesting that this multiplier and quantization method can be very effective for CNN inference applications and potentially on-chip neural network training, which both benefit from higher activations precision. For the multipliers used in this experiment, the pipelined RCCMs can operate between 350 MHz (4 bit) and 250 MHz (16 bit) while the generic multiplier can be clocked at between 200 MHz (4 bit) and 150 MHz (16 bit). Here, it is expected that the generic multiplier can be faster for faster timing constraints at the cost of additional resources.

6.7.2 Architecture Resource Utilization

In this section, we ran PAR experiments to compare our RCCMs against conventional 8-bit multipliers in a single CNN layer. Table 6.5 shows the resource utilization as well as the obtained speed.

The first row uses the fewest LUTs as multiplication is done in the DSPs. When DSPs are disabled, LUT usage dramatically increases. Our 2-Add design achieves the highest frequency at a significantly reduced LUT count compared to the 8-bit DSP disabled implementation. However, we note that the LUT usage could be reduced if implemented with tree-structured optimizations as in [76]. The 3-Add and 4-Add designs have more flexibility compared with the 2-Add, but require slightly more LUTs and operate with reduced frequency.

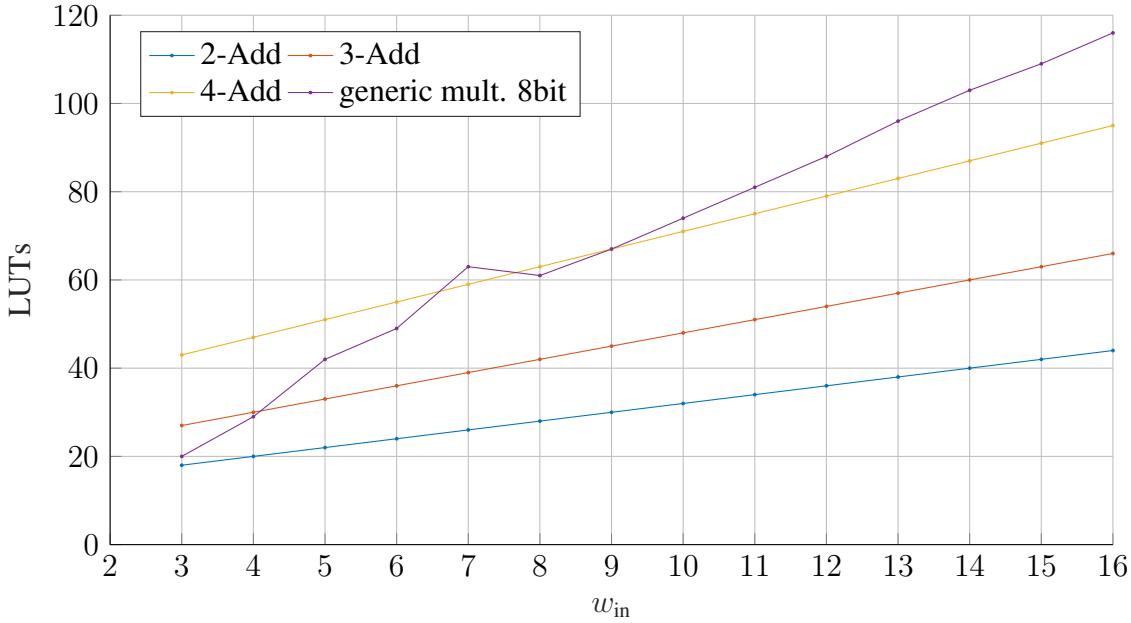


FIGURE 6.8. LUT results from synthesis for the proposed RCCMs and a generic $8 \times w_{\text{in}}$ multiplier

TABLE 6.5. PAR result comparison one layer with 10 neurons (with and without DSP mapping enabled)

Method	LUTs	FFs	DSPs	BRAM	Freq. [MHz]
8-bit, with DSPs	238	1015	10	5	446.63
8-bit, no DSPs	1407	1515	0	5	355.11
2-Add	818	1421	0	5	464.11
3-Add	928	1365	0	5	415.15
4-Add	1179	1385	0	5	342.68

table 6.6 shows resource utilization for the two architectures, using the 2,3,4-Add and 8-bit DSP disabled designs. The 2,3,4-Add cases all achieve significant LUT savings. The PCIe interface uses 48 DSPs and 100 BRAMs. Weight storage memory reduction is also apparent in the form of a decrease in BRAM utilization from 1557 in the 8-bit model to 1365 for 2-Add. This is because the reduction in bits per weight by using 2-Adders results in a 50% savings in BRAMs in the 3rd convolutional layer, as highlighted in table 6.7. This large savings is due to the discrete size of Xilinx BRAMs. Xilinx BRAMs can be configured to have a data width of 1, 2, 4, 9 and 18 bits. The wider the data width, the fewer number of words that can be stored per BRAM. The required data width for each PE is given by the bits stored per

TABLE 6.6. Summary Of PAR Utilization on the Xilinx KU115 for all arithmetic types with PCIe interface included.

Xilinx KU115	**Architecture**	**2-Add**	**3-Add**	**4-Add**	**8-bit disab. [3].**	**8-bit enab. [3]**
BRAM (2160)	Conv Layer	1154	1154	1154	1154	170
	AlexNet	1365	1557	1557	1557	1229
DSP (5520)	Conv Layer	48	48	48	48	96
	AlexNet	48	48	48	48	3760
LUTs (663K)	Conv Layer	187.0	205.6	255.8	383.0	36.2
	AlexNet	331.7	372.8	430.7	467.1	128.8
Estim. Power	Conv Layer	7.6W	7.6W	7.8W	7.5W	7.2W
	AlexNet	39W	44W	48W	52W	29W

TABLE 6.7. Per Layer BRAM usage, p represents the parallelism of the PE and b represents the bits required to store each coefficient

	p	PE	b		BRAMs		Memory (MB)	
			2-Add	8-bit	2-Add	8-bit	2-Add	8-bit
Conv1	4	96	8	8	96	96	0.04	0.04
Conv2	4	256	4	8	256	256	0.15	0.31
Conv3	**1**	**384**	**4**	**8**	**192**	**384**	**0.44**	**0.88**
Conv4	2	384	4	8	384	384	0.33	0.66
Conv5	2	256	4	8	256	256	0.44	0.88

weight, b multiplied by the parallelism of the PE, p. In the case of Conv3, where $p = 1$, by reducing b to 4-bits, it is possible to store all the required weights for a PE using only a single 18K BRAM, in contrast to a 36K BRAM for the 8-bit case. In other layers where p is higher, reduced memory use of 2-adders does not result in fewer BRAMs used due to their discrete sizes.

In Table 6.6, we also present the estimated overall board power. Evidently there is an increase in board power when we disable DSPs. This is due to the increase of LUT resources which typically leads to more switching. However, comparing the 8-bit with DSPs disabled with AddNet implementations, we see reductions in power. Again, this is largely due to the reduction in LUTs.

The silicon area savings could also be used to scale-up the parallelism to improve throughput, reduce latency or fit the design on a smaller FPGA. For example, while the AlexNet and Conv Layer implementations already have one PE per output feature map, we can increase p_l and compute more output feature map pixels in parallel to reduce the number of PE iterations required to compute a layer. This trivial optimization could be applied when accelerating most large neural networks. Typically such networks have lots of inherent parallelism and high computational requirements, with FPGA accelerators implementing some form of layer folding due to resource restrictions. This is especially the case for higher precision implementations when accuracy preservation is paramount. Thus, the AddNet multiplier is a very widely applicable tool for improving the parallelism of existing FPGA DNN architectures. Alternatively, these area reductions allow the current design to fit on a smaller device, such as the Xilinx VU3P, which would lead to expected reductions in power consumption as less hardware is being used.

6.7.3 Frequency

The AlphaData CNN Library can operate conservatively at 250 MHz [3] and the critical path lies in the PCIe interface. Therefore, since our RCCMs can operate at a higher frequency, it does not translate to an increase in operating frequency of the overall system. However, as mentioned in Section 6.4.3, the RCCMs designed can also be trivially pipelined to improve their operating frequencies. As the multipliers were additionally implemented without the PCIe interface as both standalone components and within a CNN layer, we explored the frequencies of pipelined versions. The post PAR frequencies with a clock constraint of 250 Mhz (more aggressive time constraints will lead to higher frequencies than what is reported) for the standalone multipliers are shown in Table 6.8. Significant frequency improvements are demonstrated from the pipelined versions which would lead to designs achieving higher frequencies when the multiplier lies in the critical path of the system. As the frequency is maintained at 250 MHz for all our implementations, the throughput remains constant.

TABLE 6.8. PAR frequencies for pipelined versions of the RCCMs

Type	2-Add	3-Add	4-Add
Original	447.43 MHz	483.09 MHz	342.82 MHz
Pipelined	770.42 MHz	578.03 MHz	623.83 MHz

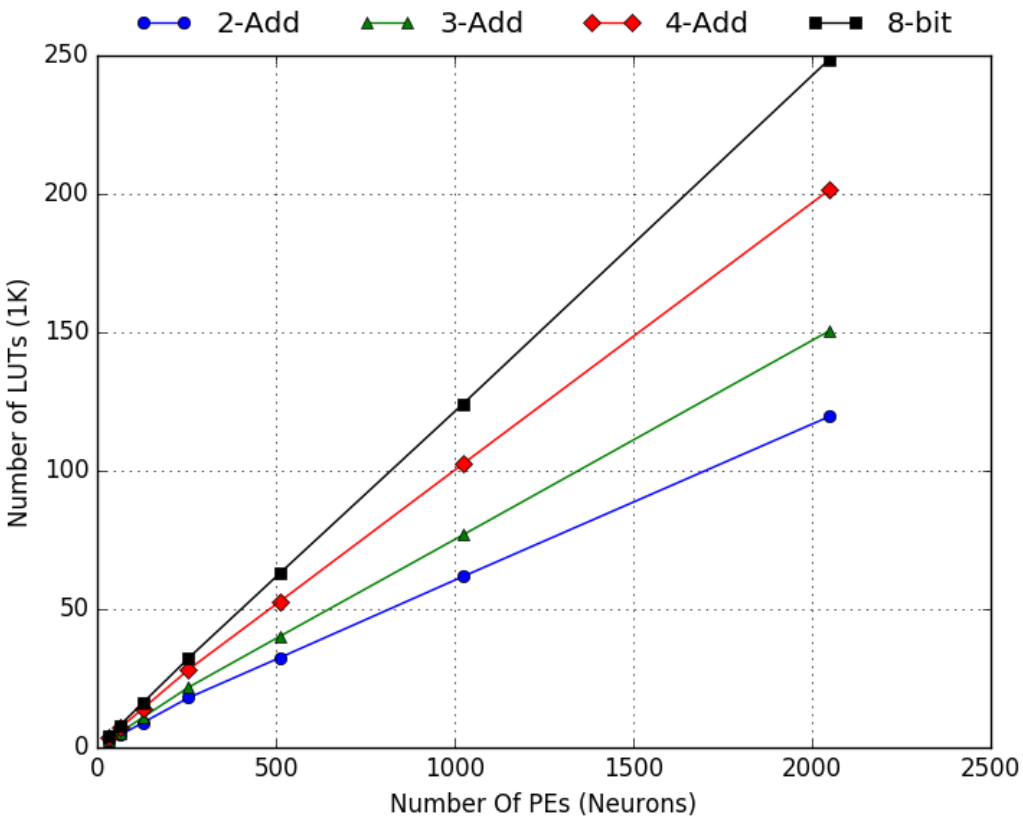


FIGURE 6.9. Relationship between LUTs and amount of parallelism for different arithmetic

6.7.4 Effect of Layer Size

We now explore how the resources usage scales with parallelism for a single layer convolutional core without the PCIe interface. This is an important metric for data flow implementations as we want to instantiate a higher number of PEs in layers with the most operations to achieve load balancing with less operationally intensive layers. Figure 6.9 shows LUT usage from PAR where the number of parallel PEs is equal to typical neuron layer sizes used in our trained networks. Using such sizes allows us to simulate computing all output feature maps

TABLE 6.9. Accuracy results [%] for AddNet, floating-point (32 bit) and fixed-point training over various ImageNet models

Model		2-Add	3-Add	4-Add	float.	8-bit	6-bit	4-bit	Ternary	Binary
AlexNet	Top-1	55.8	55.8	55.9	55.1	55.5	54.7	53.9	53.2	52.0
	Top-5	79.8	79.8	80.0	79.2	78.6	78.5	78.3	78.1	76.9
ResNet-18	Top-1	65.1	67.4	68.1	68.6	66.0	63.5	62.0	61.6	57.5
	Top-5	86.4	87.6	87.8	88.2	87.5	85.9	85.4	84.2	81.2
ResNet-50	Top-1	72.1	72.7	73.8	76.0	72.5	69.6	68.4	67.0	65.0
	Top-5	91.2	91.5	92.0	92.9	91.6	89.5	89.1	88.7	86.5

of a layer in parallel. As expected, all implementations scale linearly with the number of PEs. However, for the AddNet multipliers, as we increase the number of PEs we see smaller increase in LUTs in comparison to the 8-bit version. This is amplified further with the smaller multiplier implementations which demonstrate smaller gradients to the 4-Add version. For example, with 2048 PEs instantiated, we achieve a substantial 52% LUT reduction. Typically neural network implementations are constrained by the number of PEs we can instantiate per layer due to resource scaling.

6.7.5 Accuracy

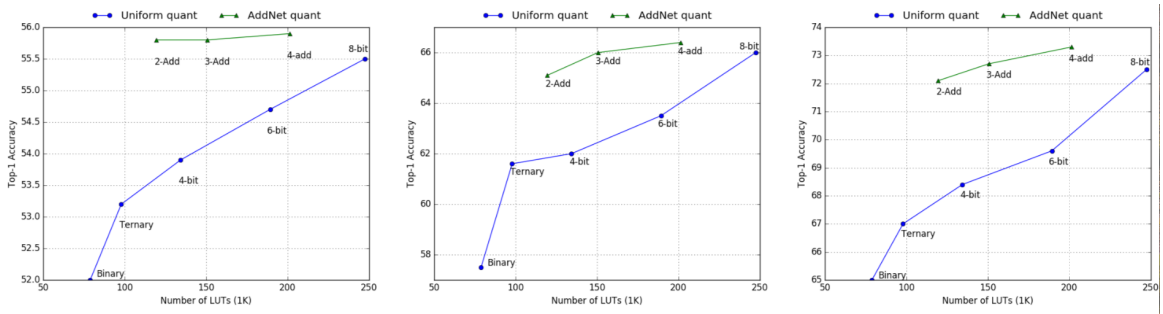
To demonstrate the robustness of our quantization strategy, we implement the training on several benchmark networks for image classification. The proposed method is evaluated on the ILSVRC-2012 ImageNet dataset which contains natural high resolution visual classification dataset consisting of 1000 classes, 1.28 million training images and 50K validation images. The images are preprocessed as per the reference models by resizing the inputs to 256×256 before being randomly cropped to 224×224. We report our single-crop performance evaluation results using Top-1 and Top-5 accuracy, where the cross-entropy loss of the predicted classification against the actual classification is minimized during training. The AlexNet network consists of 5 convolutional and 3 fully-connected layers. ResNet networks consist of blocks of two or three convolutional layers and a residual connection [55]. Two models are explored with varying depths of these blocks.

In Table 6.9, we display the accuracies of quantizing for different multiplier sizes and compare them to fixed point re-training and floating point network accuracies. All results were trained with the 4-Add RCCM in the first and last layers to preserve accuracy and were trained for a fixed number of epochs. For all these networks we achieve at least 8-bit accuracy with resource savings through our multiplier. This demonstrates the effectiveness of AddNet. In particular, we can achieve equivalent to floating point accuracy for AlexNet with only 2-Add multipliers which translates to large resource savings. In some instances, the accuracy is improved and this is due to the regularization effect of the quantization which improves the generalization of the network.

6.7.6 Accuracy vs Area

Fundamentally, our goal is to achieve the highest possible accuracy while consuming the smallest amount of resources. Thus, it is important to evaluate the accuracy achieved against the amount of resources used. To do this, we have analyzed the area consumed for different precisions of traditional fixed-point training against AddNet training. figure 6.10 shows these evaluations for each network. The closer data points are to the top left corner of the graphs, the more optimized and more efficient the method. We see that both the 2-Add and 3-Add cases show improvements over the traditional quantization methods. This demonstrates the effectiveness of our training methodology. The 4-Add case achieves the same or greater accuracy than the 8-bit but with significantly less resources. Additionally, for all three networks, the 2-Add and 3-Add case significantly improve accuracy and area over 6-bit implementations and the 2-Add case significantly improves accuracy and area over the 4-bit implementations. This is a very important contribution of this work: instead of reducing precision, which is a standard approach to save silicon area, our method gets better area savings and much better accuracy for all networks.

After investigating the effect of weight precision, we also analyze the effect of activation precision on both accuracy and area. table 6.10 shows the accuracy against different sizes for w_{in} using the 2-Add multiplier coefficients for ResNet-18. We particularly analyze 2-Add ResNet-18 as it has the highest discrepancy from our 8-bit and full-precision models.



(a) AlexNet (b) ResNet-18 (c) ResNet-50

FIGURE 6.10. Accuracy-Area comparison of uniform and AddNet quantization for AlexNet and ResNet

TABLE 6.10. Accuraciy results from changing activation bit width for 2-Add ResNet-18

	2-bit	4-bit	8-bit	12-bit	16-bit
Top-1	54.3%	57.7%	58.2%	58.2%	58.2%
Top-5	79.8%	81.4%	82.0%	82.0%	82.0%

As shown, increasing the activation to 16-bits does not close the accuracy gap. Observing figure 6.8, the area of the 2-Add with $w_{\text{in}} = 16$ is roughly equivalent to the 3-Add with $w_{\text{in}} = 8$. Thus, in this case, it is much more effective to use the 3-Add with $w_{\text{in}} = 8$ as the accuracy is improved.

6.8 Summary

In this Chapter, RCCMs are designed to utilize low-level FPGA architecture properties. RCCMs restrict their inputs by only allowing a certain types of coefficient sets. Thus, following the design of RCCMs, a training methdology is described to train CNNs, such that their representations are compatible with the multiplier. The is done via a distribution matching technique and quantization training using the STE method. We then describe our FPGA hardware design for evaluating the performance of our training methods. We compare different multiplier sizes which allow for different coefficient set sizes. We also compare against using standard fixed-point arithmetic. Hardware results regarding resource usage, estimated power consumption and frequency are reported. We also investigate the effect

of multiplier sizes have on network accuracy and present results for an accuracy-hardware trade-off between using our RCCM arithmetic against conventional fixed-point arithmetic.

CHAPTER 7

Conclusion

The overall goal of this thesis was to design DNN representations that are amenable to hardware implementation, but still maintain high accuracy. Over the course of this thesis, different training techniques were developed to achieve this. Their accuracy was evaluated on benchmark image classification/object detection datasets and resource utilisation/performance on specialized hardware. They were able to improve on previous state-of-the-art methods for different accuracy-hardware performance tradeoffs, making these techniques suitable to various applications on embedded systems. The contributions, and to what extent they satisfied our original aims, are explored in greater detail in the following paragraphs.

The first aim of this thesis was to explore whether the use of sparse representations could enhance DNN hardware performance on embedded platforms. To simplify the problem, bitwise networks were the focus. A representation with high sparsity was trained, without impinging on accuracy, and then lossless compression algorithms were used to compress the model. This work was effective in reducing memory and improving throughput of state-of-the-art bitwise networks via hardware cost models. Additionally, these models improved or maintained accuracy on the MNIST and CIFAR10 datasets. This illustrates that for very high amounts of sparsity, bitwise networks can be compressed and/or accelerated further on FPGA technology and other specialized hardware platforms. Implementations of DNNs on smaller devices and/or with higher throughput is therefore possible.

The next aim of this thesis was to design sparse representations in a way that requires minimal additional hardware for its implementation. Sparsity is induced in bitwise networks by making all weights in certain convolutional kernels equal to zero. Since computation for those kernels can be ignored, the regular data access patterns of dense matrices are maintained,

meaning no decompressor is required in hardware. This ensures these are applicable to improving many existing DNN hardware implementations. This is done via a hardware-aware filter pruning framework for customizing low-precision CNNs to underlying FPGA architectures. By prioritizing hardware implications of pruning rather than sensitivities of each layer, performance improvements are achieved for bitwise networks over existing state-of-the-art pruning techniques. Additionally, state-of-the-art FPGA performance was achieved for several metrics in terms of FPS, FPS/kLUT and FPS/BRAM for the AlexNet and TinyYolo networks on the ImageNet dataset. These results illustrated that by using a software-hardware co-design approach, performance improvements were possible on various hardware platforms. This broadens the applicability of DNNs, making them more useful for high-throughput applications.

Another aim of this thesis was to improve the accuracy of hardware-friendly DNN implementations, with minimal additional overhead. This was studied once again in the context of improving the accuracy of bitwise networks, which previously achieved high hardware performance at the cost of a hit in terms of accuracy. Learning scaling factors was proposed using gradient-based optimizations and ordering them via a diagonal scalar matrix for efficient computation. In contrast to other quantization approaches, this reduced the computational requirements of fine-grained quantization and significantly improved state-of-the-art accuracy on modern benchmark networks such as AlexNet, ResNet and VGG, on the ImageNet dataset. In addition, to demonstrate the amenability of our representations on specialized hardware, an architecture was designed and implemented on an FPGA for varying parallelisms which incurred minimal hardware complexity. These results show that achieving equivalent floating point accuracy is possible whilst maintaining similar computational complexity to bitwise networks, meaning DNN implementations on embedded platforms may be suitable for life-critical decision making applications.

The final aim of this thesis was to increase precision of the network without incurring the typical hardware costs associated with conventional fixed point arithmetic. To do this, a custom arithmetic was developed which utilized low-level hardware optimizations to efficiently utilize FPGA resources. Following this, a training algorithm was described to make CNN

representations which are compatible with the hardware. More specifically, reconfigurable constant coefficient multipliers were explored for CNN inference. A novel distribution matching scheme which restricts the allowable coefficient values in a computationally tractable manner is proposed, as well as an associated training algorithm. Our results showed that this approach achieves better accuracy on the ImageNet dataset than bitwise networks, while allowing the expensive multipliers usually used in fixed-point implementations to be replaced by shifts, adds and small multiplexers. Overall, the approach reduces mismatch between CNN computation and existing FPGA device architectures, making more efficient implementations possible. It also demonstrates the benefits of designing a custom arithmetic and training technique for a given hardware platform. This also motivates an investigation into new low-level DNN hardware designs which don't rely on conventional arithmetic methods.

7.1 Future outlook

In Chapters 3, the direct hardware architecture performance values were not integrated into the training algorithm. Instead, sparsity was used in the training algorithm as the metric for considering hardware performance. It was assumed the more sparsity, the better the hardware performance. However, a potentially more effective approach to these, would be to implement automated hardware-software co-design. More specifically, Neural Architecture (NAS) search methods [26] which use a multi-objective cost function. By directly embedding the target hardware's performance into the learning algorithm, it is possible to rely on gradient-based optimizations to find better accuracy-hardware performance trade-offs [58]. This could potentially guide the training process to choose sparsity in more specific areas. It would also be interesting for researchers to extend this work by applying these techniques for more complex datasets and models and hence potentially improve the accuracy and its applicability to real-world problems.

In Chapter 4, the layers pruned were chosen based on their hardware performance impacts. I.e. layers which were bottlenecks were targeted. Again, this type of optimization may benefit from NAS methods by embedding the hardware costs of each layer into the training algorithm

and using gradient-based methods to determine which layers, how many filters and which filters to prune. This is similarly done in [142]. Additionally, further work in this area could explore using various other network layers such as the addition of residual layers to improve accuracy with potentially a small hardware cost. Hardware cost models could be designed for other DNN implementations on other hardware platforms such as CPUs, GPUs and ASICs with similar pruning techniques applied.

In Chapter 5, learnable scaling factors were introduced in a pixel-wise/row-wise. However, exploring scaling factor granularity along the kernel axis, is only suitable for some hardware architectures. Conducting further research in this area could investigate relaxing the restriction of scaling factor placement having to satisfy a diagonal scalar matrix. This could either improve accuracy further or generalize this technique to a broader range of hardware architectures. Exploring different granularities along the input and/or output feature map channels may achieve this. Again, the placement may be learnt via NAS methods which search for optimal quantization strategies such as in [11].

In Chapter 6, while the benefits of developing customised training techniques for RCCMs were demonstrated on CNNs, it is expected that training any type of neural network to make use of RCCMs is possible. This technique introduces a new dimension for optimization of neural networks in which the arithmetic is directly customized, and is orthogonal to matrix decomposition and sparsity-inducing approaches. This approach should be explored as an alternative to simply studying the use of reduced precision fixed point arithmetic. Further research in this area may also explore designing custom arithmetic for various hardware platforms and other FPGA architectures. Additionally, for the same FPGA architecture, other more efficient multipliers, or alternative arithmetic operators, may be designed and an accompanying training algorithm derived that builds on our presented approach.

Bibliography

[1] Mohamed S. Abdelfattah et al. 'DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration'. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (2018), pp. 411–4117.

[2] Kamel Abdelouahab et al. 'Accelerating CNN inference on FPGAs: A Survey'. In: *CoRR* abs/1806.01683 (2018). arXiv: 1806.01683. URL: http://arxiv.org/abs/1806.01683.

[3] Alpha Data. *An Open Source FPGA CNN Library*. May 2017. URL: ftp://ftp.alpha-data.com/pub/appnotes/cnn/ad-an-0055_v1_0.pdf.

[4] Altera. 'FPGA Architecture White Paper'. In: (). URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf.

[5] Arash Ardakani, Carlo Condo and Warren J Gross. 'Sparsely-connected neural networks: towards efficient vlsi implementation of deep neural networks'. In: *arXiv preprint arXiv:1611.01427* (2016).

[6] Utku Aydonat et al. 'An OpenCL(TM) Deep Learning Accelerator on Arria 10'. In: *CoRR* abs/1701.03534 (2017).

[7] Mohammad Babaeizadeh, Paris Smaragdis and Roy H Campbell. 'Noiseout: A simple way to prune neural networks'. In: *arXiv preprint arXiv:1611.06211* (2016).

[8] Philip Bachman, Alessandro Sordoni and Adam Trischler. 'Learning Algorithms for Active Learning'. In: *Proceedings of the 34th International Conference on Machine Learning*. Ed. by Doina Precup and Yee Whye Teh. Vol. 70. Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR, June 2017, pp. 301–310. URL: http://proceedings.mlr.press/v70/bachman17a.html.

[9] Yoshua Bengio, Nicholas Léonard and Aaron C. Courville. 'Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation'. In: *CoRR* abs/1308.3432 (2013). arXiv: 1308.3432. URL: http://arxiv.org/abs/1308.3432.

[10] Kiran Bhageshpur. 'Data Is The New Oil – And That's A Good Thing'. In: 2019. URL: https://www.forbes.com/sites/forbestechcouncil/2019/11/15/data-is-the-new-oil-and-thats-a-good-thing/?sh=62ea340e7304.

[11] Zhaowei Cai and N. Vasconcelos. 'Rethinking Differentiable Search for Mixed-Precision Neural Networks'. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), pp. 2346–2355.

[12] Zhaowei Cai et al. 'Deep Learning with Low Precision by Half-wave Gaussian Quantization'. In: *CoRR* abs/1702.00953 (2017).

[13] Wenlin Chen et al. 'Compressing Neural Networks with the Hashing Trick'. In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 2285–2294. URL: http://dl.acm.org/citation.cfm?id=3045118.3045361.

[14] Sharan Chetlur et al. 'cuDNN: Efficient Primitives for Deep Learning'. In: *arXiv preprint arXiv:1410.0759* (2014).

[15] Yoni Choukroun et al. 'Low-bit quantization of neural networks for efficient inference'. In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE. 2019, pp. 3009–3018.

[16] 'Computer Vision Market Size, Share Trends Analysis Report By Component (Hardware, Software), By Product Type (Smart Camera-based, PC-based), By Application, By Vertical, By Region, And Segment Forecasts, 2020 - 2027'. In: *Grandview Research* (2020). URL: https://www.grandviewresearch.com/industry-analysis/computer-vision-market.

[17] Corinna Cortes, Mehryar Mohri and Afshin Rostamizadeh. 'L2 regularization for learning kernels'. In: *arXiv preprint arXiv:1205.2653* (2012).

[18] Matthieu Courbariaux and Yoshua Bengio. 'BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1'. In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830. URL: http://arxiv.org/abs/1602.02830.

[19] Matthieu Courbariaux, Yoshua Bengio and Jean-Pierre David. 'BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations'. In: *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 3123–3131. URL: http://dl.acm.org/citation.cfm?id=2969442.2969588.

[20] Matthieu Courbariaux, Yoshua Bengio and Jean-Pierre David. 'BinaryConnect: Training Deep Neural Networks with binary weights during propagations'. In: *CoRR* abs/1511.00363 (2015). URL: http://arxiv.org/abs/1511.00363.

[21] 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*. 2015, pp. 3123–3131.

[22] Matthieu Courbariaux et al. 'Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1'. In: *arXiv preprint arXiv:1602.02830* (2016).

[23] S S Demirsoy, A.G. Dempster and I Kale. 'Design guidelines for reconfigurable multiplier blocks'. In: *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*. 2003.

[24] J. Deng et al. 'ImageNet: A Large-Scale Hierarchical Image Database'. In: *CVPR09*. 2009.

[25] Caiwen Ding et al. 'REQ-YOLO: A Resource-Aware, Efficient Quantization Framework for Object Detection on FPGAs'. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: ACM, 2019, pp. 33–42. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293904. URL: http://doi.acm.org/10.1145/3289602.3293904.

[26] Xuanyi Dong and Yi Yang. 'Network pruning via transformable architecture search'. In: *Advances in Neural Information Processing Systems*. 2019, pp. 760–771.

[27] Yong Dou et al. '64-Bit Floating-Point FPGA Matrix Multiplication'. In: FPGA '05. Monterey, California, USA: Association for Computing Machinery, 2005, pp. 86–95. ISBN: 1595930299. DOI: 10.1145/1046192.1046204. URL: https://doi.org/10.1145/1046192.1046204.

[28] Yueqi Duan et al. 'Learning Deep Binary Descriptor With Multi-Quantization'. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[29] The Economist. 'The world's most valuable resource is no longer oil, but data'. In: 2017. URL: https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data.

[30] M. Everingham et al. 'The Pascal Visual Object Classes (VOC) Challenge'. In: *International Journal of Computer Vision* 88.2 (June 2010), pp. 303–338.

[31] Julian Faraone et al. 'AddNet: Deep Neural Networks Using FPGA-Optimized Multipliers'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28 (2020), pp. 115–128.

[32] Julian Faraone et al. 'Compressing Low Precision Deep Neural Networks Using Sparsity-Induced Regularization in Ternary Networks'. In: *CoRR* abs/1709.06262 (2017).

[33] Julian Faraone et al. 'Customizing Low-Precision Deep Neural Networks for FPGAs'. In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)* (2018), pp. 97–973.

[34] Julian Faraone et al. 'SYQ: Learning Symmetric Quantization for Efficient Deep Neural Networks'. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[35] Johannes de Fine Licht, Grzegorz Kwasniewski and Torsten Hoefler. 'Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis'. In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, pp. 244–254. ISBN: 9781450370998. DOI: 10.1145/3373087.3375296. URL: https://doi.org/10.1145/3373087.3375296.

[36] Sean Fox et al. ‘A Block Minifloat Representation for Training Deep Neural Networks’. In: *International Conference on Learning Representations*. 2021. URL: https://openreview.net/forum?id=6zaTwpNSsQ2.

[37] Sean Fox et al. ‘Training Deep Neural Networks in Low-Precision with High Accuracy Using FPGAs’. In: *2019 International Conference on Field-Programmable Technology (ICFPT)*. 2019, pp. 1–9. DOI: 10.1109/ICFPT47387.2019.00009.

[38] Nicholas J. Fraser et al. ‘Scaling Binarized Neural Networks on Reconfigurable Logic’. In: *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. PARMA-DITAM ’17. Stockholm, Sweden: ACM, 2017, pp. 25–30. ISBN: 978-1-4503-4877-5. DOI: 10.1145/3029580.3029586. URL: http://doi.acm.org/10.1145/3029580.3029586.

[39] Nicholas J Fraser et al. ‘Scaling binarized neural networks on reconfigurable logic’. In: *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*. 2017, pp. 25–30.

[40] Yao Fu et al. *Deep Learning with INT8 Optimization on Xilinx Devices (White Paper)*. Tech. rep. Xilinx, Inc., 2017.

[41] Mohammad Ghasemzadeh, Mohammad Samragh and Farinaz Koushanfar. ‘ReBNet: Residual Binarized Neural Network’. In: *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. 2018, pp. 57–64. DOI: 10.1109/FCCM.2018.00018. URL: https://doi.org/10.1109/FCCM.2018.00018.

[42] Xavier Glorot and Yoshua Bengio. ‘Understanding the difficulty of training deep feedforward neural networks’. In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS’10). Society for Artificial Intelligence and Statistics*. 2010.

[43] David Goldberg. ‘What every computer scientist should know about floating-point arithmetic’. In: *ACM Computing Surveys (CSUR)* 23.1 (1991), pp. 5–48.

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

[45] Robert M. Gray and David L. Neuhoff. 'Quantization'. In: *IEEE transactions on information theory* 44.6 (1998), pp. 2325–2383.

[46] Kaiyuan Guo et al. 'A Survey of FPGA Based Neural Network Accelerator'. In: *CoRR* abs/1712.08934 (2017). arXiv: 1712.08934. URL: http://arxiv.org/abs/1712.08934.

[47] Suyog Gupta et al. 'Deep Learning with Limited Numerical Precision'. In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2015, pp. 1737–1746. URL: http://dl.acm.org/citation.cfm?id=3045118.3045303.

[48] Song Han et al. 'Dsd: Dense-sparse-dense training for deep neural networks'. In: *arXiv preprint arXiv:1607.04381* (2016).

[49] Song Han, Huizi Mao and William J. Dally. 'Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding'. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1510.00149.

[50] Song Han et al. 'EIE: Efficient Inference Engine on Compressed Deep Neural Network'. In: *CoRR* abs/1602.01528 (2016). arXiv: 1602.01528. URL: http://arxiv.org/abs/1602.01528.

[51] Song Han et al. 'ESE: Efficient Speech Recognition Engine with Compressed LSTM on FPGA'. In: *CoRR* abs/1612.00694 (2016).

[52] Song Han et al. 'Learning both weights and connections for efficient neural network'. In: *Advances in neural information processing systems*. 2015, pp. 1135–1143.

[53] Martin Hardieck et al. 'Reconfigurable Convolutional Kernels for Neural Networks on FPGAs'. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*. 2019, pp. 43–52. DOI: 10.1145/3289602.3293905. URL: https://doi.org/10.1145/3289602.3293905.

[54] R. I. Hartley. 'Subexpression sharing in filters using canonic signed digit multipliers'. In: *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing* 43.10 (Oct. 1996), pp. 677–688. ISSN: 1057-7130. DOI: 10.1109/82.539000.

[55] Kaiming He et al. 'Deep Residual Learning for Image Recognition'. In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[56] Kaiming He et al. 'Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification'. In: *CoRR* abs/1502.01852 (2015). arXiv: 1502.01852. URL: http://arxiv.org/abs/1502.01852.

[57] Yang He et al. 'Progressive deep neural networks acceleration via soft filter pruning'. In: *arXiv preprint arXiv:1808.07471* 1.2 (2018), p. 8.

[58] Yihui He et al. 'AMC: AutoML for Model Compression and Acceleration on Mobile Devices'. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Sept. 2018.

[59] Itamar Ben Hemo. 'Big Data Statistics: How Much Data Is There in the World?' In: 2020. URL: https://rivery.io/big-data-statistics-how-much-data-is-there-in-the-world/.

[60] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[61] Catherine F. Higham and D. Higham. 'Deep Learning: An Introduction for Applied Mathematicians'. In: *ArXiv* abs/1801.05894 (2019).

[62] Andrew G. Howard et al. 'MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications'. In: *CoRR* abs/1704.04861 (2017).

[63] Itay Hubara et al. 'Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations'. In: *CoRR* abs/1609.07061 (2016). arXiv: 1609.07061. URL: http://arxiv.org/abs/1609.07061.

[64] Itay Hubara et al. 'Quantized neural networks: Training neural networks with low precision weights and activations'. In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6869–6898.

[65] IEEE. *IEEE standard for binary floating-point arithmetic*. Note: Standard 754–1985. New York: Institute of Electrical and Electronics Engineers, 1985.

[66] Sergey Ioffe and Christian Szegedy. 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift'. In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[67] Alex Krizhevsky, Vinod Nair and Geoffrey Hinton. 'CIFAR-10 (Canadian Institute for Advanced Research)'. In: (). URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[68] Alex Krizhevsky, Ilya Sutskever and Geoffrey E Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[69] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Proc. Int. Conf. on Neural Information Processing Systems*. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[70] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: http://doi.acm.org/10.1145/3065386.

[71] S Kullback and R A Leibler. 'On information and sufficiency'. In: *The Annals of Mathematical Statistics* 22.1 (1951), pp. 79–86.

[72] M. Kumm et al. 'Multiple constant multiplication with ternary adders'. In: *2013 23rd International Conference on Field programmable Logic and Applications* (2013), pp. 1–8.

[73] M Kumm, Shahid Abbas and Peter Zipf. 'An Efficient Softcore Multiplier Architecture for Xilinx FPGAs'. In: *IEEE Symposium on Computer Arithmetic (ARITH)*. 2015, pp. 18–25.

[74] Ian Kuon and Jonathan Rose. ‘Measuring the gap between FPGAs and ASICs’. In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 26.2 (2007), pp. 203–215.

[75] Kiseok Kwon et al. ‘Co-design of Deep Neural Nets and Neural Net Accelerators for Embedded Vision Applications’. In: *Proceedings of the 55th Annual Design Automation Conference*. DAC ’18. San Francisco, California: ACM, 2018, 148:1–148:6. ISBN: 978-1-4503-5700-5. DOI: 10.1145/3195970.3199849. URL: http://doi.acm.org/10.1145/3195970.3199849.

[76] Martin Langhammer and Gregg Baeckler. ‘High Density and Performance Multiplication for FPGA’. In: *IEEE Symposium on Computer Arithmetic*. 2018.

[77] Guillaume Leclerc et al. ‘Smallify: Learning network size while training’. In: *arXiv preprint arXiv:1806.03723* (2018).

[78] Yann LeCun. ‘The MNIST database of handwritten digits’. In: *http://yann. lecun. com/exdb/mnist/* (1998).

[79] Yann LeCun, Yoshua Bengio et al. ‘Convolutional networks for images, speech, and time series’. In: ().

[80] Yann LeCun, Yoshua Bengio and Geoffrey Hinton. ‘Deep learning’. In: *nature* 521.7553 (2015), pp. 436–444.

[81] Yann LeCun, Yoshua Bengio and Geoffrey Hinton. ‘Deep learning’. In: *Nature* 521 (May 2015), 436 EP -. URL: http://dx.doi.org/10.1038/nature14539.

[82] Fengfu Li and Bin Liu. ‘Ternary Weight Networks’. In: *CoRR* abs/1605.04711 (2016). URL: http://arxiv.org/abs/1605.04711.

[83] Huimin Li et al. ‘A high performance FPGA-based accelerator for large-scale convolutional neural networks’. In: *Proc. Int. Conf. on Field Programmable Logic and Applications* (2016), pp. 1–9.

[84] Sicheng Li et al. ‘An FPGA Design Framework for CNN Sparsification and Acceleration’. In: *FCCM*. IEEE Computer Society, 2017, p. 28.

[85] Ling Liang et al. ‘Crossbar-aware neural network pruning’. In: *IEEE Access* 6 (2018), pp. 58324–58337.

[86] Darryl D. Lin, Sachin S. Talathi and V. Sreekanth Annapureddy. 'Fixed Point Quantization of Deep Convolutional Networks'. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. New York, NY, USA: JMLR.org, 2016, pp. 2849–2858. URL: http://dl.acm.org/citation.cfm?id=3045390.3045690.

[87] Zhouhan Lin et al. 'Neural Networks with Few Multiplications'. In: *CoRR* abs/1510.03009 (2015). URL: http://arxiv.org/abs/1510.03009.

[88] Clark S Lindsey and Thomas Lindblad. 'Review of Hardward Neural Networks: A User's Perspective'. In: *INTERNATIONAL JOURNAL OF NEURAL SYSTEMS* 6 (1995), pp. 215–224.

[89] Christos Louizos, Karen Ullrich and Max Welling. 'Bayesian compression for deep learning'. In: *Advances in neural information processing systems*. 2017, pp. 3288–3298.

[90] Huizi Mao et al. 'Exploring the Regularity of Sparse Structure in Convolutional Neural Networks'. In: *CoRR* abs/1705.08922 (2017).

[91] Michael C. McFarland, Alice C. Parker and Raul Camposano. 'Tutorial on High-Level Synthesis'. In: *Proceedings of the 25th ACM/IEEE Design Automation Conference*. DAC '88. Atlantic City, New Jersey, USA: IEEE Computer Society Press, 1988, pp. 330–336. ISBN: 0818688645.

[92] Naveen Mellempudi et al. 'Ternary Neural Networks with Fine-Grained Quantization'. In: *CoRR* abs/1705.01462 (2017). URL: http://arxiv.org/abs/1705.01462.

[93] Asit K. Mishra et al. 'WRPN: Wide Reduced-Precision Networks'. In: *CoRR* abs/1709.01134 (2017). arXiv: 1709.01134. URL: http://arxiv.org/abs/1709.01134.

[94] Janardan Misra and Indranil Saha. 'Artificial Neural Networks in Hardware: A Survey of Two Decades of Progress'. In: *Neurocomput.* 74.1–3 (Dec. 2010), pp. 239–255. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2010.03.021. URL: https://doi.org/10.1016/j.neucom.2010.03.021.

[95] Daisuke Miyashita, Edward H. Lee and Boris Murmann. 'Convolutional Neural Networks using Logarithmic Data Representation'. In: *CoRR* abs/1603.01025 (2016). arXiv: 1603.01025. URL: http://arxiv.org/abs/1603.01025.

[96] Konrad Möller et al. 'Dynamically Reconfigurable Constant Multiplication on FPGAs'. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. 2014, pp. 159–169.

[97] Konrad Möller et al. 'Reconfigurable Constant Multiplication for FPGAs'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 36.6 (2017), pp. 927–937.

[98] K Möller et al. 'Optimal Shift Reassignment in Reconfigurable Constant Multiplication Circuits'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37.3 (Mar. 2018), pp. 710–714.

[99] Gareth W Morris, David B Thomas and Wayne Luk. 'FPGA accelerated low-latency market data feed processing'. In: *2009 17th IEEE Symposium on High Performance Interconnects*. IEEE. 2009, pp. 83–89.

[100] Duncan J. M. Moss et al. 'High performance binary neural networks on the Xeon+FPGA™ platform'. In: *FPL*. IEEE, 2017, pp. 1–4.

[101] Duncan J.M Moss et al. 'A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study'. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '18. Monterey, CALIFORNIA, USA: ACM, 2018, pp. 107–116. ISBN: 978-1-4503-5614-5. DOI: 10.1145/3174243.3174258. URL: http://doi.acm.org/10.1145/3174243.3174258.

[102] Rene Mueller, Jens Teubner and Gustavo Alonso. 'Data processing on FPGAs'. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 910–921.

[103] Markus Nagel et al. 'Data-free quantization through weight equalization and bias correction'. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 1325–1334.

[104] Marcelo Gennari do Nascimento, Roger Fawcett and Victor Adrian Prisacariu. 'DSConv: Efficient Convolution Operator'. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 5148–5157.

[105] Stephen Neuendorffer and Fernando Martinez-Vallina. 'Building zynq® accelerators with Vivado® high level synthesis'. In: *The 2013 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '13, Monterey, CA, USA, February 11-13, 2013*. Ed. by Brad L. Hutchings and Vaughn Betz. ACM, 2013, pp. 1–2. DOI: 10.1145/2435264.2435266. URL: https://doi.org/10.1145/2435264.2435266.

[106] Andrew Y. Ng. 'Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance'. In: *Proceedings of the Twenty-First International Conference on Machine Learning*. ICML '04. Banff, Alberta, Canada: Association for Computing Machinery, 2004, p. 78. ISBN: 1581138385. DOI: 10.1145/1015330.1015435. URL: https://doi.org/10.1145/1015330.1015435.

[107] Eriko Nurvitadhi et al. 'Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC'. In: *Int. Conf. on Field-Programmable Technology* (2016), pp. 77–84.

[108] Eriko Nurvitadhi et al. 'Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC'. In: *FPL*. IEEE, 2016, pp. 1–4.

[109] Eunhyeok Park, Junwhan Ahn and Sungjoo Yoo. 'Weighted-Entropy-Based Quantization for Deep Neural Networks'. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[110] Mohammad Rastegari et al. 'XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks'. In: *CoRR* abs/1603.05279 (2016). URL: http://arxiv.org/abs/1603.05279.

[111] Mohammad Rastegari et al. 'Xnor-net: Imagenet classification using binary convolutional neural networks'. In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[112] Joseph Redmon and Ali Farhadi. 'YOLO9000: Better, Faster, Stronger'. In: *CoRR* abs/1612.08242 (2016).

[113] Sébastien Ricard. ‘AI’s Effect On Productivity Now And In The Future’. In: 2020. URL: https://www.forbes.com/sites/forbestechcouncil/2020/03/20/ais-effect-on-productivity-now-and-in-the-future/?sh=586676237591.

[114] Juan J Rodrıguez-Andina, Maria D Valdes-Pena and Maria J Moure. ‘Advanced features and industrial applications of FPGAs—A review’. In: *IEEE Transactions on Industrial Informatics* 11.4 (2015), pp. 853–864.

[115] David E Rumelhart, Geoffrey E Hinton and Ronald J Williams. ‘Learning representations by back-propagating errors’. In: *nature* 323.6088 (1986), pp. 533–536.

[116] Olga Russakovsky et al. ‘ImageNet Large Scale Visual Recognition Challenge’. In: *Int. J. Comput. Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0816-y. URL: http://dx.doi.org/10.1007/s11263-015-0816-y.

[117] Herman Schmit. ‘Extra-dimensional island-style FPGAs’. In: *New Algorithms, Architectures and Applications for Reconfigurable Computing*. Springer, 2005, pp. 3–13.

[118] Siddhartha et al. ‘Long Short-Term Memory for Radio Frequency Spectral Prediction and its Real-Time FPGA Implementation’. In: *MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM)*. 2018, pp. 1–9. DOI: 10.1109/MILCOM.2018.8599833.

[119] Bruno da Silva et al. ‘A Multimode SoC FPGA-based acoustic camera for wireless sensor networks’. In: *2018 13th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE. 2018, pp. 1–8.

[120] Karen Simonyan and Andrew Zisserman. ‘Very Deep Convolutional Networks for Large-Scale Image Recognition’. In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[121] Yi Sun, Xiaogang Wang and Xiaoou Tang. ‘Sparsifying neural network connections for face recognition’. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4856–4864.

[122] Vivienne Sze et al. 'Efficient Processing of Deep Neural Networks: A Tutorial and Survey'. In: *CoRR* abs/1703.09039 (2017). arXiv: 1703.09039. URL: http://arxiv.org/abs/1703.09039.

[123] Mingxing Tan and Quoc Le. 'EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks'. In: *International Conference on Machine Learning*. 2019, pp. 6105–6114.

[124] Wei Tang, Gang Hua and Liang Wang. 'How to Train a Compact Binary Neural Network with High Accuracy?' In: *AAAI*. 2017.

[125] P Tummeltshammer, J C Hoe and M Püschel. 'Time-Multiplexed Multiple-Constant Multiplication'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.9 (Sept. 2007), pp. 1551–1563.

[126] Yaman Umuroglu et al. 'FINN: A Framework for Fast, Scalable Binarized Neural Network Inference'. In: *CoRR* abs/1612.07119 (2016). arXiv: 1612.07119. URL: http://arxiv.org/abs/1612.07119.

[127] Yaman Umuroglu et al. 'FINN: A Framework for Fast, Scalable Binarized Neural Network Inference'. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: ACM, 2017, pp. 65–74. ISBN: 978-1-4503-4354-1. DOI: 10.1145/3020078.3021744. URL: http://doi.acm.org/10.1145/3020078.3021744.

[128] Yaman Umuroglu et al. 'Finn: A framework for fast, scalable binarized neural network inference'. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 65–74.

[129] Stylianos I. Venieris and Christos-Savvas Bouganis. 'fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs'. In: *Int. Symp. on Field-Programmable Custom Computing Machines*. May 2016, pp. 40–47.

[130] Stylianos I. Venieris, Alexandros Kouris and Christos-Savvas Bouganis. 'Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions'. In: *ACM Comput. Surv.* 51.3 (June 2018), 56:1–56:39. ISSN: 0360-0300. DOI: 10.1145/3186332. URL: http://doi.acm.org/10.1145/3186332.

[131] Ganesh Venkatesh, Eriko Nurvitadhi and Debbie Marr. 'Accelerating Deep Convolutional Networks using low-precision and sparsity'. In: *CoRR* abs/1610.00324 (2016). arXiv: 1610.00324. URL: http://arxiv.org/abs/1610.00324.

[132] Ganesh Venkatesh, Eriko Nurvitadhi and Debbie Marr. 'Accelerating Deep Convolutional Networks using low-precision and sparsity'. In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017*. 2017, pp. 2861–2865. DOI: 10.1109/ICASSP.2017.7952679. URL: https://doi.org/10.1109/ICASSP.2017.7952679.

[133] Ganesh Venkatesh, Eriko Nurvitadhi and Debbie Marr. 'Accelerating deep convolutional networks using low-precision and sparsity'. In: *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2017, pp. 2861–2865.

[134] E G Walters. 'Partial-Product Generation and Addition for Multiplication in FPGAs with 6-Input LUTs'. In: *Asilomar Conference on Signals, Systems and Computers* (2014), pp. 1247–1251.

[135] Haohan Wang and Bhiksha Raj. 'On the origin of deep learning'. In: *arXiv preprint arXiv:1702.07800* (2017).

[136] Paul Webster. 'Patient data in the cloud'. In: 1 (Dec. 2019). URL: https://doi.org/10.1016/S2589-7500(19)30202-X.

[137] Jiaxiang Wu et al. 'Quantized convolutional neural networks for mobile devices'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 4820–4828.

[138] *Xilinx DPU*. Accessed: 2019-05-09. URL: https://www.xilinx.com/products/intellectual-property/dpu.html.

[139] Yuhui Xu et al. 'Trained rank pruning for efficient deep neural networks'. In: *arXiv preprint arXiv:1812.02402* (2018).

[140] Jiwei Yang et al. 'Quantization networks'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 7308–7316.

[141] Tien-Ju Yang, Yu-Hsin Chen and Vivienne Sze. 'Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning'. In: *CoRR* abs/1611.05128 (2016).

[142] Tien-Ju Yang et al. 'NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications'. In: *ArXiv* abs/1804.03230 (2018).

[143] Tien-Ju Yang et al. 'Netadapt: Platform-aware neural network adaptation for mobile applications'. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 285–300.

[144] Yifan Yang et al. 'Synetgy: Algorithm-hardware Co-design for ConvNet Accelerators on Embedded FPGAs'. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '19. Seaside, CA, USA: ACM, 2019, pp. 23–32. ISBN: 978-1-4503-6137-8. DOI: 10.1145/3289602.3293902. URL: http://doi.acm.org/10.1145/3289602.3293902.

[145] Jiecao Yu et al. 'Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism'. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 548–560. ISSN: 0163-5964.

[146] Chen Zhang et al. 'Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks'. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '15. Monterey, California, USA: ACM, 2015, pp. 161–170. ISBN: 978-1-4503-3315-3. DOI: 10.1145/2684746.2689060. URL: http://doi.acm.org/10.1145/2684746.2689060.

[147] Dejiao Zhang et al. 'Learning to share: Simultaneous parameter tying and sparsification in deep learning'. In: *International Conference on Learning Representations*. 2018.

[148] Dongqing Zhang et al. 'LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks'. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. Sept. 2018.

[149] Tianyun Zhang et al. 'Adam-admm: A unified, systematic framework of structured weight pruning for dnns'. In: *arXiv preprint arXiv:1807.11091* 2.3 (2018).

[150] Ritchie Zhao, Christopher De Sa and Zhiru Zhang. 'Overwrite Quantization: Opportunistic Outlier Handling for Neural Network Accelerators'. In: *arXiv preprint arXiv:1910.06909* (2019).

[151] Ritchie Zhao et al. 'Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs'. In: *Proc. Int. Symp. on Field-Programmable Gate Arrays*. 2017, pp. 15–24.

[152] Ritchie Zhao et al. 'Improving neural network quantization without retraining using outlier channel splitting'. In: *arXiv preprint arXiv:1901.09504* (2019).

[153] Zhong-Qiu Zhao et al. 'Object detection with deep learning: A review'. In: *IEEE transactions on neural networks and learning systems* 30.11 (2019), pp. 3212–3232.

[154] Aojun Zhou et al. 'Explicit loss-error-aware quantization for low-bit deep neural networks'. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 9426–9435.

[155] Aojun Zhou et al. 'Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights'. In: *CoRR* abs/1702.03044 (2017). URL: http://arxiv.org/abs/1702.03044.

[156] Shuchang Zhou et al. 'DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients'. In: *CoRR* abs/1606.06160 (2016).

[157] Zhengguang Zhou et al. 'Progressive Learning of Low-Precision Networks for Image Classification'. In: *IEEE Transactions on Multimedia* (2020).

[158] Chenzhuo Zhu et al. 'Trained Ternary Quantization'. In: *CoRR* abs/1612.01064 (2016). URL: http://arxiv.org/abs/1612.01064.

[159] Ling Zhuo and V. Prasanna. 'Scalable and modular algorithms for floating-point matrix multiplication on FPGAs'. In: *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* (2004), pp. 92–.