Improving Efficiency and Accuracy for Training and Inference of Hardware-aware Machine Learning Systems

Submitted in partial fulfillment of the requirements for
the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Ruizhou Ding

B.S., Electronics Engineering and Computer Science, Peking University

Carnegie Mellon University
Pittsburgh, PA

May 2020
Acknowledgements

Pursuing the PhD at Carnegie Mellon University for me is a journey of pursuing the truth. It is memorable, unique, and wonderful. It would not have been possible without the support and guidance that I received from many people.

First and foremost, I would like to express my great appreciation to my advisor, Prof. Diana Marculescu. She has been very supportive to my ideas and has given me the freedom to pursue the projects that I am interested in. Her scientific advice and knowledge has also helped me a lot via many insightful discussions. During the most difficult times of doing research, she gave me great support and helped me move towards the right direction. When my paper about LightNNs got rejected at the first time, she told me immediately “I’m sure it’s a good paper" and encouraged me to continue working on it. It’s been proved that she is right, when an improved version of this paper got the best paper award by another conference. I feel very fortunate to have such a wonderful advisor, and I sincerely thank her for the great mentorship.

I would also like to thank Prof. R.D. (Shawn) Blanton, Prof. Radu Marculescu and Dr. Vikas Chandra for being my committee members and providing me with insightful suggestions and invaluable guidance through the process.

I would like to thank my collaborators: Prof. R.D. (Shawn) Blanton, Prof. Radu Marculescu, Dr. Zhuo Chen, Mr. Zeye Liu, Mr. Dimitrios Stamoulis, Mr. Ting-Wu Chin, Dr. Rongye Shi and Dr. Kartikeya Bhardwaj for their inspiring ideas and great contributions. It has been a wonderful experience to work with all of them.

I would also like to thank my internship mentor: Dr. Hoang Do, Dr. Feng Yang and Mr. Kevin Li for helping me build real-world applications with my research skills.

In addition, I would like to thank my friends for making my life full of joy. They are: Dr. Ermao Cai, Mr. Ahmet Fatih Inci, Dr. Da-Cheng Juan and Mr. Guangshuo Liu from EnyAC; Mr. Tian Tong, Mr. Yuwei Qin, Mr. Jiaqi Liu, Mr. Xi He, Ms. Qicheng Huang, Ms. Chenlei Fang, Mr. Chao Rong, Mr. Kai-Chun Lin, Mr. Qing-Yi Lin, Mr. Xiaoliang Li, Mr. Vincent Chung from CMU ECE.

I would like to thank Mr. Zhaoxing Lang, Mr. Wei Peng, and Ms. Wanying Fu for hosting the board games.
I would also like to acknowledge the funding support received from National Science Foundation, Carnegie Mellon University, CyLab IoT initiative, and Infineon that has made pursuing my research possible.

Finally, I would like to thank my parents who have always been believing in me and encouraging me to pursue my dream. Most of all, I truly thank my girlfriend, Ms. Sixuan Ren, for sticking by my side and making all of these possible.
Abstract

Deep Neural Networks (DNNs) have been adopted in many systems because of their higher classification accuracy. While progress in achieving large scale, highly accurate DNNs has been made, however, the huge time and energy requirement for both inference and training pose a challenge to any DNN implementation. The large DNN size causes significant energy and area due to massive memory accesses and computations. In this thesis, we propose a new DNN architecture, LightNN, which replaces the multiplications to one shift or a constrained number of shifts and adds. Thus, LightNN inference uses hardware-efficient shift-based operations, and needs fewer memory accesses due to the fewer bits to represent each weight. Our experiment using image datasets show that LightNNs can achieve much higher hardware efficiency with a slight accuracy loss.

LightNNs constrain all the weights of DNNs to be a limited combination (denoted as $k \in \{1, 2\}$) of powers of 2, and by varying the $k$ one can obtain a group of LightNNs with different levels of accuracy and energy cost. To provide even more design flexibility, the $k$ for each convolutional filter can be optimally chosen instead of being fixed for every filter. In this thesis, we formulate the selection of $k$ to be differentiable, and describe model training for determining $k$-based weights on a per-filter basis. Over 46 FPGA-design experiments involving eight configurations and four data sets reveal that lightweight neural networks with a flexible $k$ value (dubbed FLightNNs) fully utilize the hardware resources on Field Programmable Gate Arrays (FPGAs), our experimental results show that FLightNNs can achieve $2 \times$ speedup when compared to lightweight NNs with $k = 2$, with only 0.1% accuracy degradation. Compared to a 4-bit fixed-point quantization, FLightNNs achieve higher accuracy and up to $2 \times$ inference speedup, due to their lightweight shift operations. In addition, our experiments also demonstrate that FLightNNs can achieve higher computational energy efficiency for ASIC implementation.

Binarized Neural Networks (BNNs) quantize the weights and activations to 1 bit, which has lower precision than LightNNs and FLightNNs, and therefore can significantly reduce the inference latency and energy consumption in resource-constrained devices due to their pure-logical computation and fewer memory accesses. However, training BNNs is difficult since the activation flow encounters degeneration, saturation, and gradient mismatch problems. Prior work alleviates these issues by
increasing activation bits and adding floating-point scaling factors, thereby sacrificing BNN’s energy efficiency. In this thesis, we propose to use distribution loss to explicitly regularize the activation flow, and develop a framework to systematically formulate the loss. Our experiments show that the distribution loss can consistently improve the accuracy of BNNs without losing their energy benefits. Moreover, equipped with the proposed regularization, BNN training is shown to be robust to the selection of hyper-parameters including optimizer and learning rate.
Contents

List of Tables x

List of Figures xii

1 Introduction 1
   1.1 Contributions ................................................. 4
   1.2 Thesis Organization .......................................... 5

2 Background 6
   2.1 Conventional DNNs ............................................. 6
   2.2 Energy Consumption of DNN Inference ......................... 7
   2.3 Quantized DNNs ................................................. 8
   2.4 BNNs ............................................................ 10

3 Hardware-Efficient Inference via LightNNs 13
   3.1 Chapter Overview .............................................. 13
   3.2 LightNN Model Architecture .................................. 14
   3.3 LightNN Training .............................................. 16
   3.4 LightNN Accuracy Analysis ................................... 16
       3.4.1 Bounding Expected Error ................................. 17
       3.4.2 Convergence Analysis .................................... 20
3.5 Training and Testing Experiments

3.5.1 Accuracy

3.5.2 Storage Requirements

3.5.3 ImageNet Results

3.5.4 Image Segmentation Results

3.5.5 Comparison with Fixed-point Weights

3.5.6 Varying Number of Shifts

3.5.7 Varying Number of Shifting Bits

3.6 Hardware Evaluation

3.6.1 Set-Up

3.6.2 Multipliers vs. Approximate Multiply Units

3.6.3 Energy Consumption

3.6.4 Area Comparison

3.6.5 Guidelines for Architecture Selection

3.6.6 Non-pipeline Implementation

3.7 FPGA Implementation

3.7.1 HLS Accelerator Implementation

3.7.2 Experiment Setup

3.7.3 Experimental Results

4 FLightNN: Improving LightNN Accuracy and Flexibility

4.1 From LightNN to FLightNN

4.2 Differentiable Training for FLightNNs

4.2.1 FLightNN quantization

4.2.2 Differentiable training

4.2.3 Regularization

4.3 FLightNN Accuracy Analysis

4.4 Experimental Results

4.4.1 Setup
CONTENTS

4.4.2 Accuracy-throughput trade-off on FPGA ........................................ 53
4.4.3 Accuracy-energy trade-off on ASIC ............................................. 56
4.5 Discussion ...................................................................................... 59

5 Improving Accuracy for Binarized Networks via Distribution Loss 62
5.1 Motivation ..................................................................................... 62
  5.1.1 Low-Power BNNs ................................................................. 62
  5.1.2 Accuracy Loss of BNNs ......................................................... 63
  5.1.3 Chapter Contribution ............................................................. 65
5.2 BNN Activation Regularization ................................................... 66
  5.2.1 Difference Types of Binarized DNNs ...................................... 67
  5.2.2 Energy Cost for Different Types of Binarization ...................... 68
  5.2.3 Regularizing Activation Distribution ...................................... 69
  5.2.4 Intuition for the Proposed Distribution Loss ......................... 71
5.3 Experimental Results ................................................................. 73
  5.3.1 Accuracy Improvement ......................................................... 73
  5.3.2 Regularized Activation Distribution ...................................... 80
  5.3.3 Robustness to Hyper-parameter Selection ............................... 81
5.4 Discussion ..................................................................................... 84

6 Related Work ................................................................................... 85
6.1 Efficient DNN Inference .............................................................. 85
  6.1.1 Algorithmic Improvement ...................................................... 85
  6.1.2 Hardware Implementation ..................................................... 88
6.2 Efficient DNN Training ................................................................. 90
6.3 BNN Improvements ..................................................................... 91
  6.3.1 Activation Flow Regularization ............................................. 91
  6.3.2 Accuracy Improvement for BNNs .......................................... 92
  6.3.3 BNN Hardware Implementation ............................................. 92
List of Tables

3.1 Architecture characteristics for the experiments. .......................... 23
3.2 The five architecture configurations for the MNIST and CIFAR-10 experiments. .... 24
3.3 Test error and the number of parameters for all DNN architectures. ............... 24
3.4 Test error for conventional DNNs, LightNN-1 and BinaryConnect on ImageNet. The 
    accuracy for BinaryConnect is borrowed from Rastegari et al. [104] where only AlexNet 
    results are available. ............................................................................. 27
3.5 Accuracy results for conventional DNNs and LightNNs on Pascal VOC dataset. .... 27
3.6 Accuracy results for conventional DNNs and LightNNs on Cityscapes dataset. .... 28
3.7 Comparison of LightNN and fixed-point weights in terms of weight storage and test 
    error. Network size indicates the number of filters for the largest convolutional layer in 
    the network. "xW" means x bits for weights, and "yA" means y bits for activations. .... 28
3.8 Per-weight bit storage (PWBS) and test error at varied $k$ (number of shifts-and-add 
    operators for an equivalent multiplier). .................................................. 29
3.9 Per-weight bit storage (PWBS) and test error of LightNN-1 at varied $c$ (maximum 
    number of bits to shift). ................................................................. 29
3.10 Test error of conventional DNNs and LightNNs with 32-bit and 12-bit implementation. 36
3.11 Data sharing relations of convolutional operations. ..................................... 38
3.12 DNN configuration for CIFAR-10. ..................................................... 40
3.13 Timing performance comparison between conventional DNN, fixed-point DNN and LightNN. FP\(_{4W8A}\) is short for Fixed-point\(_{4W8A}\), i.e., fixed-point network with 4-bit weights and 8-bit activations. LN-1\(_{4W8A}\) refers to LightNN-1\(_{4W8A}\), i.e., LightNN-1 with 4-bit weights and 8-bit activations. Both LN-1\(_{4W8A}\) and FP\(_{4W8A}\) .................................. 40

3.14 FPGA resource usage comparison between conventional DNN, fixed-point DNN and LightNN. FP\(_{4W8A}\) is short for Fixed-point\(_{4W8A}\), i.e., fixed-point network with 4-bit weights and 8-bit activations. LN-1\(_{4W8A}\) refers to LightNN-1\(_{4W8A}\), i.e., LightNN-1 with 4-bit weights and 8-bit activations. With the same tiling size, LightNN-1 requires fewer resources than conventional DNN and fixed-point DNN. .................................................. 42

3.15 FPGA resource utilization, latency and test error comparison. ............................................. 43

4.1 Network settings. “Depth” is the number of convolutional layers in the network. “Width” is the number of convolutional filters of the largest layer. ......................... 53

4.2 Accuracy and FPGA throughput for CIFAR-10. In the “Model” column, “Full”, “L-2”, “L-1”, “FP”, “FL” indicate full-precision DNN, LightNN-2, LightNN-1, Fixed-point DNN, and FLightNN, respectively. The subscript “\(xWyA\)” indicates \(x\) bits for weights and \(y\) bits for activations. The FLightNN results are shown in bold face. We use subscript \(a\) and \(b\) to denote the two trained FLightNNs for each network. These notations also apply for Table 4.3, 4.4 and 4.5. ................................................. 55

4.3 Accuracy and FPGA throughput for SVHN. ................................................................. 56

4.4 Accuracy and FPGA throughput for CIFAR-100. ....................................................... 56

4.5 Top-5 Accuracy and FPGA throughput for ImageNet. .............................................. 57

4.6 FPGA resource utilization for different quantized DNN models. ............................... 57

4.7 Testing accuracy for fixed and error-based approaches on CIFAR-10 dataset. \(\alpha\) is the ratio of L1 filters among all the filters per layer. ......................................................... 60

5.1 Computational energy consumption and circuit area for different computation operators using a commercial 65nm process design kit [122]. The multiplier and adder are both 16-bit fixed-point operators. ................................................................. 65
5.2 Computational energy for a convolutional layer with different types of binarizations. 66
5.3 Accuracy improvement with distribution loss. Network depth is defined as the number of convolutional layers, while the network width is defined as the number of filters in the largest layer. The best results are shown in bold face. *All the accuracy for CIFAR-10 and SVHN is averaged over five experiments with different weight initialization.* 75
5.4 Comparison with prior art using 1-bit weights and activations, in terms of accuracy and computation energy on different datasets. The best results are shown in bold face. 77
5.5 Comparison with prior art on ImageNet with AlexNet-based topology. We use the same model structure as prior work, except that Compact Net uses 2 bits for activations while we only use 1 bit. Training with distribution loss outperforms prior work consistently. 79
5.6 Results on a small MobileNet-v1 for the ImageNet dataset. 81
5.7 Robustness to the selection of optimizer, learning rate, and network structure. CIFAR-10 is used for illustrating the results. 83
5.8 Accuracy for BNN-DL on CIFAR-10 with varied regularization levels. $\lambda = 0$ indicates the baseline BNN. 84

**List of Figures**

2.1 A four-layer DNN architecture. 7
3.1 (a) A conventional neuron and, (b) LightNN neuron implemented using a 2-ones approximation. 15
3.2 Storage (for weights) required by different DNN architectures under varying datasets and configurations. 25
3.3 Area and energy of an approximate multiply unit across all DNN architectures. 26
3.4 Comparison of energy consumption of different DNN architectures under varying datasets/configurations. Energy consumption is measured for inferring a single image. 32
3.5 Energy breakdown for all DNN architectures, averaged across different datasets and configurations. 32
3.6 Comparison of area of different DNN architectures under varying datasets and configurations. 33
3.7 Normalized energy and test error of varying DNN architectures and configurations for (a) MNIST and (b) CIFAR-10. Red triangles are Pareto-optimal. 34
3.8 Energy of 12-bit conventional DNNs and LightNNs. 35
3.9 Area of 12-bit conventional DNNs and LightNNs. 35
4.1 A discrete Pareto-optimal curve for LightNN models w.r.t. test error and latency/energy. More continuous Pareto-optimal points are needed to adapt to the latency/energy constraints determined by the hardware and application. 46
4.2 Quantization flow for $k = 2$. 48
4.3 Equivalent conversion from a convolution with a $k > 1$ filter to $k$ convolutions each with a $k = 1$ filter. This transforms the hardware implementation of the FLightNN into LightNN-1. 48
4.4 Regularization loss curve w.r.t. weight value. 50
4.5 Accuracy and computational energy consumption in ASIC for different quantized models on CIFAR-10, SVHN, CIFAR-100 and ImageNet datasets. FLightNNs are marked as red triangles, while the other models are shown as blue dots. 58
4.6 Accuracy-storage front for LightNN-2 LightNN-1 and FLightNN. The Pareto front of FLightNN is the upper bound of LightNNs. 59
5.1 The basic Conv-BN-Act structure for BNN (BinConv: binary convolution; BN: batch normalization). The pre-activation distribution may exhibit from degeneration, saturation or gradient mismatch problem that causes difficulty in training. 64
5.2 Basic block for convolutional BNN [62]. The activations $I_l$ and weights $W$ are binarized to ±1. The inference of this block can be implemented on hardware with only logical operators. 67

5.3 Basic block for XNOR-Net [105]. 68

5.4 Basic block for ABC-Net [83]. 68

5.5 Motivation for adjusting regularization. The loss function directly formulated from hypothesis (e.g., degeneration) relies on the minimum (or maximum) of the pre-activations, and therefore is sensitive to outliers. 70

5.6 Proposed framework for formulating the differentiable loss function to regularize activation distribution. Starting from the three hypotheses (“degeneration”, “saturation” and “gradient mismatch”), we can formulate the loss function $L_D$, $L_S$ and $L_M$ for them, respectively. We omit the superscript for $A$ and $L$ for better representation. 72

5.7 Basic block for ResNet-based BNN [83]. 74

5.8 Accuracy and energy Pareto-optimal curve for CIFAR-10, SVHN and CIFAR-100. The error bars for CIFAR-10 and SVHN show the standard deviation of testing accuracy. 76

5.9 Training loss and testing accuracy curves for different networks with or without distribution loss for CIFAR-10 dataset. The widths of the curves are 2 standard deviation ranges. 77

5.10 Results for SVHN. The widths of the curves are 2 standard deviation ranges. 78

5.11 Results for CIFAR-100. 79

5.12 Activation distribution for BNN trained (a) without or (b) with distribution loss. Each histogram refers to the activations of one channel. In (a), the channel in the left histogram shows a generation problem, the middle two show gradient mismatch, and the right one shows saturation problem. $\sigma$ is standard deviation, and “positive” refers to the ratio of positive activations. 81

5.13 Histogram of standard deviation and positive ratio of per-channel activations. 82

5.14 Per-channel activation quantiles for the small MobileNet-v1 on ImageNet. Channels are sorted by their 50% quantiles for better visualization. 82
Chapter 1

Introduction

In recent years, machine learning has become a useful tool for various applications [50, 17] due to the massive data available for training a model and the increase of computational power. To use a machine learning model, one needs to first train the model given the training data, and then deploy the model on the machines that can perform the inference. Depending on the machine that the model is deployed on, serving the model can be classified into two types, on-cloud inference and on-edge inference. In both cases, the data are collected by the edge devices. On-cloud inference sends the collected data from edge devices to the cloud, and performs the model inference on the cloud, while on-edge inference directly performs the inference on the edge devices. On-edge inference outperforms on-cloud computing for better user privacy, lower transmission latency, and less requirement for an internet connection. However, edge devices may have battery constraints, and thus can only use less powerful hardware (e.g., cores, memory, etc.) than cloud machines. Therefore, in this thesis, we aim to make the machine learning models more lightweight to enable on-edge inference.

As a widely implemented machine learning model, deep neural networks (DNNs) have shown significant efficiency in classification, due to their non-linear characteristics, flexible configurations and self-adaptive features [146]. In this thesis, we focus on on-edge inference for deep learning models. Increasingly in recent years, research has focused on DNNs implemented directly in hardware for a variety of reasons stemming from design requirements or application characteristics.
CHAPTER 1. INTRODUCTION

First, real-time classification applications (such as Siri and Google glass [49]) have great sensitivity to latency, thereby making pure hardware implementations better candidates than conventional architectures based on CPUs or GPUs. Second, neural networks implemented as custom Application-Specific Integrated Circuits (ASICs) or within FPGAs require mostly logic with little complicated control, lending themselves to lower design effort. Third, heterogeneous architectures as a whole appear to be more suitable for DNN implementations due to the combined benefits of CPU, GPU, FPGA and ASIC-based hardware acceleration [35] [139] [147] [103]. In heterogeneous systems, ASICs can handle specific tasks that are required frequently, such as classification tasks in a real-time image recognition application, while CPUs and GPUs can perform the online training. However, hardware implementations of neural networks face difficulties for large scale deployment. With the increased use of DNNs, the number of layers and neurons increases significantly [28]. Google’s AlphaGo adopts a 13-layer architecture, with hundreds of filters per layer [114]. Microsoft implements a 152-layer DNN for image classification [50]. An increasing number of neurons and connections in DNNs require significant energy and power, thereby limiting their wide adoption [28].

To enable more hardware-efficient inference for DNNs, we propose lightweight DNNs to tackle memory access intensity and computation overhead, which are the two main factors limiting the energy efficiency of computing systems in general, and DNN architectures in particular. More specifically, our proposed LightNNs constrain all the weights to be a power of 2, or the sum of a limited numbers of power of 2, and constrain the activations to fixed-point numbers. Thus, the multiplication between weights and activations can be efficiently implemented with shift and add operators on customized hardware. Furthermore, fewer bits are needed to store the weights since each weight only needs to store the number of bits to shift. This indicates that LightNNs can achieve higher speed and energy efficiency than conventional DNNs.

These quantized DNNs usually incur some accuracy loss compared to their full-precision counterparts. This accuracy loss is generally caused by two reasons. First, the weight and activation quantization narrows the solution space to some discrete points, which increases the loss function value of the global optimal point. Second, during the training process, the quantization error causes the gradient of the weights and activations to be inaccurate, making the backpropagation solver more
difficult to converge to a high-quality local optimum. To tackle these two challenges, we propose Flexible LightNNs (FLightNNs), where the number of power of 2 (denoted as $k$) for the weights of each convolutional filter can be optimally chosen instead of being fixed for every filter. Thus, we formulate the selection of $k$ to be differentiable, and describe model training for determining $k$-based weights on a per-filter basis. Thus, on the one hand, the solution space is enlarged due to more flexible choice for $k$ for each convolutional filter. This reduces the loss function value of the global optimal point. On the other hand, formulating the selection of $k$ to be differentiable increases the network sparsity in a gradual manner, which may be helpful for alleviating the inaccurate gradient problem. Therefore, by inducing the flexible selection for $k$ and modifying the training algorithm, FLightNNs can provide a smoother trade-off between accuracy and energy efficiency compared to LightNNs, and may push the Pareto front of LightNNs as well.

In addition to quantizing the weights to be a limited number of power of 2, we also explore the Binarized Neural Networks (BNNs), which binarize the weights and activations to be 1 or -1. Then, the multiplication between weights and activations can be implemented highly efficiently in hardware by just using an XNOR operator. Moreover, this 1-bit representation for weights and activations can significantly reduce memory footprint in the inference phase. Thus, BNNs can reduce the inference latency and energy consumption in resource-constrained devices due to their pure-logical computation and fewer memory accesses. However, they also have big accuracy loss. In addition to the reduced precision that will naturally harm the model accuracy, we find that there are also some training induced issues, including the degeneration, saturation, and gradient mismatch problems, that will affect the accuracy of the trained model. Thus, we propose to use a distribution loss to explicitly regularize the activation flow and tackle these training induced issues. We also develop a framework to systematically formulate a hypothesized training issue into the distribution loss. Experiments show that the distribution loss can regularize the activation distribution well, and improve the accuracy of the BNN models.

Overall, in this thesis, we aim to design accurate and energy-efficient DNNs to enable on-edge inference. We first propose LightNNs that are more energy-efficient than the full-precision models, and then improve the accuracy and flexibility of LightNNs by FLightNNs. After that, we improve
the accuracy of BNNs by proposing a distribution loss for the ultra low-power applications.

## 1.1 Contributions

The contributions of this thesis cover both theoretical and practical aspects.

**Theoretical contribution** In this thesis, we propose lightweight DNN models including LightNNs and FLightNNs, and their associated training algorithms. To train the FLightNNs, we propose to formulate the selection of the number of shifts in differentiable fashion, which enables end-to-end training of the FLightNNs and reduces the search cost for determining the number of shifts per convolutional filter.

We also propose a distribution loss to regularize the activation distribution of BNNs. Prior work usually poses a regularization loss function on the weight parameters, and to the best of our knowledge, we are the first to add regularization to the activation values to help training the DNNs.

The quantization error of LightNNs and FLightNNs is bounded in a provable mathematical fashion. Our theoretical analysis shows that under convexity and Lipschitz assumptions, the LightNN and FLightNN training algorithm can converge at a rate of $O\left(\frac{1}{\sqrt{T}}\right)$, where $T$ is the training iteration.

**Practical contribution** The proposed LightNNs and FLightNNs reduce the energy cost and inference latency of DNNs by quantizing the weights and activations. LightNNs quantize the DNN weights to be a power of 2, or the sum of a limited number of powers of 2. FLightNNs customize the number of powers of 2 for each filter, which improves the accuracy of LightNNs and also provides more flexible options for hardware practitioners. The proposed distribution loss can regularize the activation flow of BNNs and improve the accuracy of the trained models.

We conduct experiments for both software training and hardware simulation on various datasets. Our experiments on ASIC simulation and FPGA implementation show that LightNNs and FLightNNs can achieve significantly higher energy efficiency than conventional DNNs, without accuracy degradation for over-parameterized models.
1.2 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 introduces the background knowledge related to DNNs, quantized DNNs and Binarized Neural Networks (BNNs). Chapter 3 introduces the LightNN and the associated training algorithm, as well as the theoretical proof for its convergence analysis and experimental results on FPGA and ASIC simulations. Chapter 4 introduces the FLightNN and its differentiable training algorithm, and shows the theoretical proof and experiment results for FLightNNs. Chapter 5 describes the distribution loss and demonstrates its effectiveness through experiments. Chapter 6 introduces some related work, and Chapter 7 concludes this thesis.
Chapter 2

Background

In this chapter, we first describe conventional DNNs and prior work that improves the energy efficiency of using the DNN for inference. Then, we introduce the architecture of quantized DNNs. Finally, we describe BNNs which quantize the DNN weights and activations to only 1 bit.

2.1 Conventional DNNs

An artificial neural network (ANN) is usually called a DNN when there are more than four layers, including the input and output layers. Without loss of generality, a 4-layer fully-connected DNN model is presented in Fig. 2.1. Suppose the input data has four features, each of which goes to one input neuron. Then, in each hidden layer and the output layer, a linear combination of the previous neurons is computed, and an activation function is applied to the result, respectively. The activation function (examples include sigmoid or ReLU [18]) applies a non-linear transformation to the output value. In Convolutional Neural Networks (CNNs), the hidden layers include convolutional layers, which compute a convolution between their input feature maps and the weights kernels. In the training phase with a back-propagation algorithm [76], the loss function, such as the $l_2$-norm [43], cross-entropy loss [43], or hinge loss [43], is computed using output values and data labels. Then, the loss function and intermediates results are used to update the weights used for linear combination described above. In the deployment (inference) phase, the output neuron with the largest value
indicates the prediction result. During deployment, the vast majority of computation resources is used for the multiplication within DNNs [35].

2.2 Energy Consumption of DNN Inference

The two main factors limiting the energy efficiency of computing systems in general, and DNN architectures in particular, are memory accesses and computation [10, 92]. Much of the prior research has focused on the computation. For instance, Du et al. adopt approximate circuits for multipliers and adders with less dynamic energy consumption [35]. Li et al. reduce energy with memristor-based approximators [78], while Kim et al. propose an approximate adder using the carry prediction technique to reduce energy [70]. Sarwar et al. use alphabet set multipliers that implement multiplication by selection, shifts and adds, reducing energy of neural networks with a pre-computed bank [112]. This prior work can reduce the energy consumption of DNNs, and some [35] [78] [70] are also compatible with DNN architectures introduced in this chapter.

There has also been significant work on reducing energy of memory accesses. Some work reduces the number of parameters through weight pruning [48], while some other work introduces structural sparsity via filter pruning for Convolutional Neural Networks (CNNs) [133] to enable speedup on general hardware platforms incorporating CPUs and GPUs. To reduce the model size, previous work has also conducted neural architecture search with energy constraint [138, 119, 92,
In addition to algorithmic advances, prior art has also proposed methodologies to achieve fast and energy-efficient DNNs. Some previous work proposes the co-design of the hardware platform and the architecture of the neural network running on it [9]. Some work proposes more lightweight DNN units for faster inference on general-purpose hardware [111], while others propose hardware-friendly DNN computation units to enable energy-efficient implementation on customized hardware [128].

Reducing the bit precision of DNN parameters has been another important trend to reduce both computational cost and memory accesses. In general, this approach is called DNN quantization, which is described in section 2.3.

### 2.3 Quantized DNNs

**What to quantize**

To enable faster and more energy-efficient inference for DNNs, one usually quantizes the weights and activations of the DNNs. The weights are the trainable parameters of the model and the activations are the intermediate results between consecutive layers of the network. Some prior work only quantizes the weights [47] and some quantizes both the weights and activations [11]. In the case of on-device training, the energy consumption of the training process is also of concern. The training process includes many iterations of forward inference and backward gradient computation. Thus, the weights, activations and also gradients of the DNNs need to be quantized [134]. In this thesis, we focus on the inference phase of DNNs, and thereby will only quantize the weights and activations.

**How to quantize**

The weight and activation quantization is to map the continuous real values to discrete levels which can be encoded in only a few bits. To enable efficient inference for DNNs, the quantized representation of the weights and activations needs to be both memory and computationally efficient. In general, the memory efficiency can be naturally satisfied by inducing the bit reduction, while the computational efficiency requires a co-design of the quantized representation and the hardware. Low-bit fixed-point and floating-point numbers are two generally used quantized representations. They can be both bit-efficient, but fixed-point representation
generally has lower energy cost than floating-point representation, under the same number of bits. Random quantization is another quantization approach, where the quantization levels are irregular rather than being equally-distant, as in fixed-point quantization. The levels in random quantization are determined by the probability distribution of the target parameter, with a goal to minimize the expected quantization error. While random quantization can reduce memory requirements for the parameters themselves, the multiplication and addition of two quantized values need to be computed using a hash table which maybe be memory-consuming, or need to be decoded into full precision first and used in full-precision operations. Thus, most work has focused on the fixed- and floating-point quantization.

**Prior Work on Quantized DNNs**  By quantizing the DNN weights and activations to discrete values, both memory accesses and computation cost can be saved. Fixed-point quantization linearly discretizes the weights into multiple levels that are equally distant. Because fixed-point quantization is easy to implement and has hardware support from existing frameworks, most of prior work uses fixed-point quantization for weights and activations. More generally, Gupta et al. replace the floating-point weights and activations with a fixed-point representation \([Q_I, Q_F]\), where \(Q_I\) and \(Q_F\) are the integer and fractional part, respectively [45]. They constrained the number of bits for \(Q_I\) and \(Q_F\) into \(IL\) and \(FL\), where \(IL\) and \(FL\) are both integers. Therefore, the weights and activations can have \(2^{IL+FL}\) value levels, which are linearly aligned in the range \([-2^{IL-1}, -2^{IL-1} - 2^{-FL}]\). In addition, in the training phase, the gradients of the weights are constrained to fixed-point values. Their experiments using the MNIST and CIFAR-10 datasets show that the test error of fixed-point DNNs with \(IL + FL = 16\) is close to that of floating-point implementations, as long as \(FL\) is sufficiently large and stochastic rounding is adopted. To delve deeper in the methodology of fixed-point quantization, Lin et al. proposed a framework to convert a pre-trained full-precision DNN into a fixed-point DNN based on signal-to-quantization-noise-ratio (SQNR) [81]. This framework includes three steps. First, a large set of typical inputs is provided as input to the DNN, and the activations per layer is recorded. Then, the statistics of weights, biases, and activations per layer are collected. Observing that weights and activations approximately follow a Gaussian distribution,
they focus on the mean and standard deviation. Finally, they used these statistics to determine each layer’s $FL$ that can maximize SQNR. Extensive experiments for fixed-point quantization are conducted by Zhou et al. [151]. They tried one-, two-, three-, four-, eight- and 32-bit fixed point quantization for the weights, activations, and gradients on Street View House Number (SVHN) [100] and ImageNet datasets [27]. The resulting accuracy demonstrates a clear reduction with decrease in bit-width, especially the bit-width of activations and gradients. Hubara et al. proposed BNNs that constrain the weights and activations to binary values (+1 or -1); they achieve almost no accuracy loss for MNIST and CIFAR-10 datasets [62]. Moreover, the reported accuracy for MNIST is even better than the conventional DNN architecture, especially when the DNN configuration is very large, with many more layers and neurons than typical. On the other hand, BNNs have notably worse accuracy when the smaller Caffe example configurations [66] for MNIST and CIFAR-10 are implemented. Mellempudi et al. proposed ternary neural networks [93] which constrain weights to be $\alpha$, 0, or $-\alpha$ and use fixed-point quantization for activations.

In addition to fixed-point quantization, Han et al. used random quantization for the weights, and achieved a high compression ratio for the fully-connected layers. [48]. From a hardware perspective, Venkataramani et al. propose an energy-efficient neural network design by reducing bit precision of resilient neurons [131], while Zhang et al. combine computation approximation and memory access reduction to form an approximate computing framework of NNs [148].

2.4 BNNs

As one of the quantized DNNs, Binarized Neural Networks (BNNs) quantize the weights and/or activations to 1 bit. Two types of BNNs have been proposed by Courbariaux et al. BinaryConnect [25], a type of BNN, only constrains the weights to +1 or -1, but leaves the inputs and intermediate results as floating point values. On the other hand, a second BNN known as BinaryNet [62], constrains both weights and intermediate results (activations) to +1 or -1, and only keeps the input values as floating point.
BNN Inference BNN inference is very energy-efficient due to the binarized parameters. For both BinaryConnect and BinaryNet, the multiplication between activations and weights is replaced by an XNOR operation in a hardware implementation. The activations for BinaryConnect are floating-point numbers, but the multiplication between the floating point activations and the binarized weights can be implemented by an XNOR operation between the sign bit of the floating point numbers and the binary ones. Also, because both BinaryConnect and BinaryNet have 1-bit weights, the memory footprint for reading the weights is greatly reduced. Since BinaryNet also has 1-bit activations, the memory footprint for reading the activations of the previous layer and writing the activations of the next layer is also reduced by 32 times compared to BinaryConnect and full-precision networks.

BinaryConnect Training To train a BinaryConnect, the classic back-propagation algorithm is adopted. The only change is that during each forward pass, the weights are copied and binarized. Then, the binarized weights are used to compute the gradients. Note that the updated weights are always stored as floating point values and only in the forward pass of the training phase and in the testing phase, the weights are binarized [25]. Another perspective of viewing this training algorithm is to use a straight-through estimator to compute the approximate gradients for the quantization function. Denote the loss function of a DNN as $F(w; x)$, then the loss function for the corresponding BNN is $F(Q(w); x)$ where $Q(.)$ is the quantization function, and in this case, is the sign function. Then, following the back-propagation algorithm, we need to compute the gradient $\frac{\partial F}{\partial w}$, and update the weights $w$ using the gradient. However, the function $F$ is non-differentiable w.r.t. $w$ because $Q(.)$ is non-differentiable. Therefore, we approximate $Q(.)$ to be an identity function when computing the gradient. Thus, the approximated $Q'(w) = 1$, making $\frac{\partial F}{\partial w} = \frac{\partial F}{\partial Q(w)}$ which is differentiable to compute.

BinaryNet Training In the testing phase, a sign function $f(x) = \text{sign}(x)$ is used as the activation function. In the training phase, since the sign function is not differentiable, the hard tanh function, defined as $H_{\text{tanh}}(x) = \text{clip}(x, -1, 1)$, is used as a substitute for the sign function to compute the gradients [62]. However, this approximation causes a gradient mismatch problem since the accurate gradient for the sign function is the Dirac delta function. Also, the sign activation function has the
degeneration and saturation problems. These problems hinder the training of BNNs, and thus, cause accuracy degradation. In this thesis, we will tackle these training-induced issues by regularizing the activation distribution.
Chapter 3

Hardware-Efficient Inference via LightNNs

In this chapter, we propose a hardware-efficient deep neural network architecture, dubbed LightNN. We will first introduce the model architecture and its training algorithm, and then provide a theoretical proof for LightNN convergence analysis. Finally, we report the experimental results for accuracy and hardware efficiency.

3.1 Chapter Overview

Emerging vision, speech and natural language applications rely on widely adopted deep learning models and, as a result, have achieved state-of-the-art accuracy. Furthermore, recent industrial efforts have focused on implementing the models on mobile devices [1]. However, real-time applications based on these deep models may incur unacceptably large latencies and can easily drain the battery on energy-limited devices. For example, smartphones can only run the AlexNet-based object detection for one hour [138]. Therefore, prior research has proposed model compression techniques including pruning and quantization to satisfy the stringent energy and speed requirements [81].

By reducing the weight and activation precision, DNN quantization has proved to be an effective technique to improve the speed and energy efficiency of DNNs on customized hardware, due to its lower computational cost and fewer memory accesses [45]. Gupta et al. show that a DNN with 16-bit fixed-point representation can achieve competitive accuracy compared to the full-precision
network [45]. In the same vein, Zhou et al. explored the DNN accuracy w.r.t. a wide range of bit widths [151]. These uniform quantization approaches enable fixed-point hardware implementation for DNNs. Courbariaux et al. propose BinaryConnect, which uses only 1 bit for the DNN parameters, turning multiplications into XNOR operations on customized hardware [25]. However, these models require an over-parameterized model size to maintain a high accuracy.

LightNNs constrain the model weights to be a power of 2, or the sum of a limited number of powers of 2, while the activations use fixed-point quantization. Therefore, the multiplication between weights and activations can be implemented in hardware by shift operations and fixed-point additions. Compared to DNNs with fixed-point quantization, LightNNs replace the fixed-point multipliers by more lightweight shift operators, or shift and additions. Since the shift operators can be implemented using Look-Up Table (LUT) on FPGA while fixed-point multipliers require Digital Signal Processing (DSP) units, LightNNs can have higher inference speed than fixed-point DNNs when run on DSP-bounded FPGAs. In addition, in an ASIC implementation, shift operations are more lightweight than multiplications, making LightNNs more energy and area efficient than fixed-point DNNs.

To improve the accuracy for LightNN, we incorporate the weight quantization into the training process rather than directly quantizing a trained full-precision model. A training algorithm for LightNNs is proposed, and a theoretical proof for the quantization error and training convergence speed is also provided under the convexity assumption.

### 3.2 LightNN Model Architecture

In binary representations, any parameter \( w \) can be written as a sum of powers of two \( w = sign(w) \cdot (2^{n_1} + 2^{n_2} + \cdots + 2^{n_K}) \), where \( K \) is the number of 1s in \( w \)’s binary representation. A multiplication of two values \( w \) and \( x \) is equal to several shifts and additions:

\[
w \cdot x = sign(w) \cdot (2^{n_1} + 2^{n_2} + \cdots + 2^{n_K}) \cdot x
= sign(w) \cdot (x << n_1 + x << n_2 + \cdots + x << n_K)
\] (3.1)
where “$x \ll n_1$” indicates left-shifting $x$ by $n_1$ bits. For negative values of $n_1$, right-shifts are used instead. Assuming $n_1 > n_2 > \cdots > n_K$, smaller $n$ values correspond to a less significant part of the result $w \cdot x$. Furthermore, logical shift units are more energy efficient than multipliers. Therefore, the computation energy consumption can be reduced by converting multiplications to approximate versions using a limited number of shifts (and adds). LightNNs change the computation logic of each neuron. A $k$-ones approximation drops the least significant powers of two in equation (1) such that the resulting value has at most $k$ ones in its binary representation. Figure 3.1 illustrates a basic example that utilizes a neuron with two inputs. Two weights $w_1$ and $w_2$ are both converted to a 2-ones approximation: $w_1 \approx 2^{n_{11}} + 2^{n_{12}}, w_2 \approx 2^{n_{21}} + 2^{n_{22}}$. Therefore, a multiplication $w \cdot x$ is changed to two shifts and one addition. Moreover, when $k = 1$, the approximate multiplier unit is only a shift.

Furthermore, we use a stochastic rounding scheme [45]. As opposed to a rounding-to-nearest scheme, the stochastic rounding scheme finds both the nearest higher value $w_h$ and the nearest lower value $w_l$, and stochastically rounds $w$ to either $w_h$ or $w_l$ based on the following:

$$w = \begin{cases} w_h, \text{ with prob } p \\ w_l, \text{ with prob } 1-p \end{cases}$$

where $p = \frac{w - w_l}{w_h - w_l}$. It can be proven that stochastic rounding ensures that the expected error introduced by the rounding scheme is zero.
3.3 LightNN Training

We adopt a training algorithm similar to that of BNNs. As shown in Algorithm 1, during each training epoch, the training data \( x \) and \( y \) are randomly split into mini-batches. For each mini-batch, the forward pass constrains the weights, and provides intermediate results and the value of the loss function. Then, the backward pass computes the derivatives of the loss function over the parameters. The parameters are then updated based on the derivatives. It is worth noting that constraining the weights occurs only in the forward pass. When updating the weights, we use \( w_{t-1} - \eta \frac{\partial F}{\partial w_c} \) instead of \( w_c - \eta \frac{\partial F}{\partial w_c} \), where \( w_{t-1} \) is the real-value weights after \((t-1)\)-th iterations, \( w_c \) is the constrained \( w_{t-1} \), \( \eta \) is the learning rate, and \( F \) is the loss function. Therefore, the weights are always accumulated in a floating point form. As stated by Courbariaux et al. [25], the reason for maintaining high resolution for the weights is that the noise needs to be averaged out by the stochastic gradient contributions accumulated for each weight. After a LightNN is trained, the last step is to perform the \( k \)-ones approximation for all weights. Then, the constrained weights are used for testing.

**LightNN with Binarized Activations** LightNN can also use a binarized activation function, as described in Chapter 2.4. The advantages of doing so is that the multiplication of a weight and an input of a neuron will be limited to \( \pm 1 \) multiplied by a power of two if the weights are constrained to \( k \)-ones approximations where \( k = 1 \). This leads to an increased likelihood of achieving energy reduction within a hardware implementation.

3.4 LightNN Accuracy Analysis

To perform a theoretical analysis for the convergence of LightNNs, we use a similar methodology to Li et al. [79]. Let \( m \) be the number of mini-batches used for backpropagation. Let \( F(\mathbf{w}) \) be the loss function. Then, \( F(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} f_i(\mathbf{w}) \), where \( f_i(\mathbf{w}) \) is the loss function for the \( i \)-th mini-batch. Assume that both \( \nabla f(\cdot) \) and \( F(\cdot) \) are L-Lipschitz smooth, i.e., \( \forall x, y \in [-1, 1]^d \), \(|\nabla f(x) - \nabla f(y)| \leq L_1 |x - y| \) and \(|F(x) - F(y)| \leq L_2 |x - y| \). Let \( \mathbf{w}' \) be the weight vector after \( t \)
Algorithm 1: LightNN Training Epoch

Input: Training dataset \((x, y)\), where \(x\) is input and \(y\) is label; parameters after the \((t - 1)\)-th iteration: \(w_{t-1}\) (weights) and \(b_{t-1}\) (biases); DNN forward computation function \(g(x, w, b)\); \(k\) value used for \(k\)-ones approximation \(\text{approx}_k(\cdot)\); learning rate \(\eta\).

Output: Updated weights \(w_t\) and biases \(b_t\).

for each mini-batch of \(x, y\) do

1. Constrain weights: \(w_c = \text{approx}_k(w_{t-1})\)
2. Forward: compute intermediate results and loss function \(F\) with \(g(\cdot), w_c, b_{t-1}\), and mini-batch of \(x\)
3. Backward: compute derivatives \(\frac{\partial F}{\partial w_c}\) and \(\frac{\partial F}{\partial b_{t-1}}\)
4. Update parameters: \(w_t = w_{t-1} - \eta \frac{\partial F}{\partial w_c}\)

end

iterations \((t \in \{1, 2, \cdots, T\}\), where \(T\) is the total number of iterations). Then, at each iteration the stochastic gradient descent algorithm updates the weights as follows:

\[
 w_{t+1} = w_t - \mu_t \nabla \tilde{f}(Q(w_t)) = w_t - \mu_t \nabla \tilde{f}(w_t) + r_t, \tag{3.2}
\]

where \(r_t = \mu_t \nabla \tilde{f}(w_t) - \mu_t \nabla \tilde{f}(Q(w_t))\) is the error introduced by quantization, \(Q(\cdot)\) is the quantization function using stochastic rounding, and \(\mu_t\) is the learning rate at \(t\)-th iteration. The quantization function can be described by \(k\) and \(c\), where \(k\) indicates that the LightNN uses \(k\)-ones approximation (also denoted as LightNN-\(k\)), and \(\frac{1}{2^c}\) is the highest resolution for the quantization, \(i.e.,\) the smallest positive quantized value. In addition, we assume that the gradient norm is bounded by a constant \(G^2\): \(E||\nabla \tilde{f}(w_t)||^2 \leq G^2\). Let \(D\) be the diameter of the weight domain. Then, the Euclidean distance between any two weights is bounded by \(D^2\). Assume that \(\forall i \in \{1, 2, \cdots, d\}\), where \(d\) is the dimension of \(w_t\), the probability density function (PDF) of the \(i\)-th element of \(w_t\) monotonically decreases with its absolute value, \(i.e.,\) \(p_{w_t}(a) \leq p_{w_t}(b)\) if \(|a| > |b|\).

In the sequel, we first bound \(E[||r_t||^2]\), and then bound the expected distance between the loss function of LightNN and conventional DNN.

### 3.4.1 Bounding Expected Error

To bound \(E[||r_t||^2]\), we first consider the case where \(c\) is very large, so we can show the rate of \(E[||r_t||^2]\) in terms of \(k\). Then, we add the constraint from \(c\).
Let quantization error for weights be denoted by \( \gamma' = Q(w') - w' \). Therefore, \( \mathbb{E}[||r'||^2] \leq \mu_l^2 L^2 \mathbb{E}[||\gamma'||^2] \). Then, we can bound \( \mathbb{E}[||r'||^2] \) by bounding \( \mathbb{E}[||\gamma'||^2] \).

**Lemma 3.4.1.** Assume that \( c \to \infty \). Let \( \mathbb{E}[||\gamma'||^2] \) for LightNN-\( k \) be denoted by \( E(k) \). Then, \( E(k + 1) \leq \frac{1}{7} E(k), \forall k \geq 1 \).

**Proof.** For LightNN-\( k \), consider any two adjacent quantized levels \( a \) and \( b \). Without losing generality, let us assume \( 0 < a < b \). Then, if \( a < w'_i < b \), the quantization error for \( w'_i \) follows: 
\[
\gamma'_i = Q(w'_i) - w'_i \leq b - a.
\]
Therefore,
\[
\mathbb{E}[(\gamma'_i)^2 | a < w'_i < b] \leq (b - a)^2. \tag{3.3}
\]
For LightNN-\( (k + 1) \), \( a \) and \( b \) are still used as quantization levels. Furthermore, the legal quantized values between \( a \) and \( b \) also include: \( a + 2^{-1}(b - a), a + 2^{-2}(b - a), \cdots \). Let \( \beta_j \) denote the \( j \)-th quantized value, \( i.e., \beta_j = a + 2^{-j}(b - a) \), where \( j = 1, 2, \cdots \). Then, \( b > \beta_1 > \beta_2 > \cdots > a \).

Therefore,
\[
\mathbb{E}[(\gamma'_i)^2 | a < w'_i < b] = \sum_{j=1}^{\infty} \mathbb{E}[(\gamma'_i)^2 | \beta_{j+1} < w'_i < \beta_j]P(\beta_{j+1} < w'_i < \beta_j | a < w'_i < b)
\leq \sum_{j=1}^{\infty} 2^{-2j}(b - a)^2 P(\beta_{j+1} < w'_i < \beta_j | a < w'_i < b)
\leq \sum_{j=1}^{\infty} 2^{-2j}(b - a)^2 2^{-j} = \frac{(b - a)^2}{7}. \tag{3.4}
\]
Note that \( P(\beta_{j+1} < w'_i < \beta_j | a < w'_i < b) \leq 2^{-j} \) holds since the PDF of the weight is monotonically decreasing with its absolute value. This means that when we increase the number of shifts used for the \( k \)-approximation from \( k \) to \( k + 1 \), \( \mathbb{E}[(\gamma'_i)^2 | a < w'_i < b] \) will decrease by a factor less than \( \frac{1}{7} \) for any interval \([a, b]\), where \( a \) and \( b \) are two adjacent quantization levels. Let all the quantized values for LightNN-\( k \) be denoted as \( \{a_j\}_{j=1,2,\cdots} \), where \( 1 \geq a_j \geq a_{j+1} > 0 \). Therefore,
\[
E(k + 1) = \sum_{i=1}^{d} \sum_{j=1}^{\infty} \mathbb{E}[(\gamma'_i)^2 | a_{j+1} < w'_i < a_j; k + 1]P(a_{j+1} < w'_i < a_j)
\leq \sum_{i=1}^{d} \sum_{j=1}^{\infty} \frac{1}{7} \mathbb{E}[(\gamma'_i)^2 | a_{j+1} < w'_i < a_j; k]P(a_{j+1} < w'_i < a_j) = \frac{1}{7} E(k). \tag{3.5}
\]
Lemma 3.4.2. Assume that $c \to \infty$. Let $\sigma_i^2 \Delta \max_i \mathbb{E}[(w_i^f)^2]$. LightNN-$k$ quantization error satisfies:

$$
\mathbb{E}[\|\gamma^f\|^2] \leq \frac{\sigma_i^2 d}{4 \cdot 7^{k-1}}.
$$

Proof. We first bound $\mathbb{E}[\|\gamma^f\|^2]$ for LightNN-1. Since the PDF for $w_i^f$ is assumed to be symmetric w.r.t. $w_i^f = 0$, $\mathbb{E}[\|\gamma^f\|^2] = \mathbb{E}[\|\gamma^f\|^2 | w_i^f > 0]$. Therefore, we only need to consider $w_i^f > 0$. Then, $w_i^f$ is rounded to $2^{\lfloor \log_2 w_i^f \rfloor}$ or $2^{\lfloor \log_2 w_i^f \rfloor}$, where $\lfloor \cdot \rfloor$ is floor operation and $\lceil \cdot \rceil$ is ceiling operation. Therefore,

$$
\mathbb{E}[(\gamma_i^f)^2 | w_i^f] = (w_i^f - 2^{\lfloor \log_2 w_i^f \rfloor})^2 (\frac{2^{\lfloor \log_2 w_i^f \rfloor} - w_i^f}{2^{\lfloor \log_2 w_i^f \rfloor}} + (2^{\lfloor \log_2 w_i^f \rfloor} - w_i^f)^2 \cdot \frac{w_i^f - 2^{\lfloor \log_2 w_i^f \rfloor}}{2^{\lfloor \log_2 w_i^f \rfloor}})
$$

\hspace{1cm} = (w_i^f - 2^{\lfloor \log_2 w_i^f \rfloor}) (2^{\lfloor \log_2 w_i^f \rfloor} - w_i^f) \leq \frac{(2^{\lfloor \log_2 w_i^f \rfloor})^2}{4} \leq \frac{(w_i^f)^2}{4}.

Therefore,

$$
\mathbb{E}[\|\gamma^f\|^2] = \sum_{i=1}^{d} \mathbb{E}[(\gamma_i^f)^2] = \sum_{i=1}^{d} \mathbb{E}[\mathbb{E}[(\gamma_i^f)^2 | w_i^f]] \leq \sum_{i=1}^{d} \mathbb{E}\left(\frac{(w_i^f)^2}{4}\right) = \frac{\sigma_i^2}{4}.
$$

Extending the bound to LightNN-$k$ with Lemma 3.4.1, we have:

$$
\mathbb{E}[\|\gamma^f\|^2] \leq \frac{\sigma_i^2 d}{4 \cdot 7^{k-1}}.
$$

Now we add the constraint from $c$, that is, the smallest positive quantized value is $\frac{1}{2^c}$. For example, if $k = 1$ and $c = 3$, then the legal quantized levels include $\pm 1, \pm \frac{1}{2}, \pm \frac{1}{4}, \pm \frac{1}{8}$.

Theorem 3.4.3. For LightNN-$k$, the error introduced by quantization is characterized by:

$$
\mathbb{E}[\|\gamma^f\|^2] \leq \frac{\mu_i^2 L_1^2 \sigma_i^2 d}{4 \cdot 7^{k-1}} + \frac{\mu_i^2 L_1^2 d}{2^{2c+2}}.
$$

Proof. Again, we consider the case $w_i^f > 0$. First, let us check the way $c$ influences the rounding error. Without constraining $c$ (i.e., $c \to \infty$), the quantized levels cut the range of $[0, 1]$ into infinite non-overlapping intervals. Define $\mathcal{A} \subset [0, 1]$ as the set of values that fall into an interval whose length is smaller than $\frac{1}{2^c}$. Then, with the constraint of the smallest possible interval $\frac{1}{2^c}$, any $w_i^f \in \mathcal{A}$ will have a larger expected rounding error, while any $w_i^f \notin \mathcal{A}$ will not be affected. Now suppose
$w_i \in [a, b]$, where $a, b \in [0, 1]$ are two adjacent quantized levels, and $b - a = \frac{1}{2^c}$. The expected rounding error for $w_i$ is bounded by:

$$
\mathbb{E}[(\gamma_i^2) | w_i \in [a, b)] = \frac{w_i - a}{b - a} \cdot (b - w_i)^2 + \frac{b - w_i}{b - a} \cdot (w_i - a)^2
$$

$$
= (w_i - a)(b - w_i) \leq \frac{(b - a)^2}{4} = \frac{1}{2^{2c+2}}
$$

(3.10)

Note that this is larger than the bound in equation (3.4) due to the constraint from $c$. Then, the quantization error bound for the weights in $\mathcal{A}$ will increase to $\frac{1}{2^{2c+2}}$. Therefore, the overall bound is changed to:

$$
\mathbb{E}[||r^i||^2] = \sum_{i=1}^{d} \mathbb{E}[||\gamma_i||^2] = \sum_{i=1}^{d} \mathbb{E}[(\gamma_i^2)|w_i \in \mathcal{A}]P(\mathcal{A}) + \mathbb{E}[(\gamma_i^2)|w_i \notin \mathcal{A}](1 - P(\mathcal{A}))
$$

$$
\leq \sum_{i=1}^{d} \mathbb{E}[(\gamma_i^2)|w_i \in \mathcal{A}] + \mathbb{E}[(\gamma_i^2)|w_i \notin \mathcal{A}] \leq \frac{d}{2^{2c+2}} + \frac{\sigma_i^2 d}{4 \cdot 7^{k-1}}
$$

Therefore,

$$
\mathbb{E}[||r^i||^2] \leq \mu_i^2 L_1^2 \mathbb{E}[||\gamma'||^2] \leq \frac{\mu_i^2 L_1^2 \sigma_i^2 d}{4 \cdot 7^{k-1}} + \frac{\mu_i^2 L_1^2 d}{2^{2c+2}}
$$

(3.11)

(3.12)

3.4.2 Convergence Analysis

**Lemma 3.4.4.** \(\forall i \in \{1, 2, \ldots, d\}, \forall t \in \{1, 2, \ldots, T\}, \text{if applying stochastic rounding on } w_i \text{ leads to error } \gamma_i, \text{ then } \mathbb{E}[\gamma_i w_i] = 0.\)

**Proof.** Since stochastic rounding has the property: \(\mathbb{E}[\gamma_i | w_i] = 0\), we have:

$$
\mathbb{E}[\gamma_i w_i] = \mathbb{E}[\mathbb{E}[\gamma_i w_i | w_i]] = \mathbb{E}[w_i \mathbb{E}[\gamma_i | w_i]] = 0
$$

(3.13)

In other words, the expected bounding error is zero.\[\square\]

We now give the bound for the expected distance between the loss function of LightNNs and corresponding conventional DNNs. The optimal solution in the continuous domain is defined as $w^* \triangleq \arg\min_{w \in [-1, 1]^d} F(w)$. The quantized weights averaged by $T$ training iterations is defined as $\tilde{w}^T \triangleq \frac{1}{T} \sum_{t=1}^{T} w^t$. 
Theorem 3.4.5. Assume that the loss function \( F(\mathbf{w}) \) is convex within domain \([-1, 1]^d\), and the learning rate follows \( \mu_t = \frac{\mu_1}{\sqrt{t}} \). Let \( \sigma^2 = \text{max} \{ \sigma_i^2 \} \leq 1 \). Then, the expected distance between the loss function of the LightNN and the corresponding DNN vanishes for a large number of iterations \( T \):

\[
\mathbb{E}[F(\mathbf{w}_T^T) - F(\mathbf{w}^*)] \leq \frac{D^2(\sqrt{T} + 1)}{2T\mu_1} + \frac{\mu_1\sqrt{T} + 1}{T}(\frac{1}{2}G^2 + \frac{L_1^2 \sigma^2 d}{4 \cdot 7^{k-1}} + \frac{L_1^2 d}{2^{2c+2}}) \sim O\left(\frac{1}{\sqrt{T}}\right) \tag{3.14}
\]

Proof. From equation 3.2, we have:

\[
\mathbb{E}[\|\mathbf{w}^{t+1} - \mathbf{w}^*\|^2] = \mathbb{E}[\|\mathbf{w}' - \mathbf{w}^*\|^2 + \mathbb{E}[\|\mu_t \nabla \tilde{f}(\mathbf{w}') - \mathbf{r}'\|^2 - 2\mathbb{E}[\langle \mathbf{w}' - \mathbf{w}^*, \mu_t \nabla \tilde{f}(\mathbf{w}') \rangle - \mathbf{r}'] > 0 \tag{3.15}
\]

Due to Lemma 3.4.4 and \( \mathbb{E}[\tilde{f}(\mathbf{w}') = \mathbb{E}[F(\mathbf{w}') \), we have:

\[
\mathbb{E}[\|\mathbf{w}^{t+1} - \mathbf{w}^*\|^2] = \mathbb{E}[\|\mathbf{w}' - \mathbf{w}^*\|^2 + \mathbb{E}[\|\nabla \tilde{f}(\mathbf{w}')\|^2 + \mathbb{E}[\|\mathbf{r}'\|^2 - 2\mathbb{E}[\langle \mathbf{w}' - \mathbf{w}^*, \mu_t \nabla F(\mathbf{w}') \rangle]] \tag{3.16}
\]

Analyzing the right hand side, we have that \( \mathbb{E}[\|\nabla \tilde{f}(\mathbf{w}')\|^2 \leq G^2 \), that \( \mathbb{E}[\|\mathbf{r}'\|^2 \) is bounded by 3.4.3, and \( \mathbb{E}[\langle \mathbf{w}' - \mathbf{w}^*, \mu_t \nabla F(\mathbf{w}') \rangle] \geq \mu_t \mathbb{E}[F(\mathbf{w}') - F(\mathbf{w}^*)] \), due to the convexity of function \( F(\cdot) \).

Therefore, we have:

\[
\mathbb{E}[\|\mathbf{w}^{t+1} - \mathbf{w}^*\|^2] \leq \mathbb{E}[\|\mathbf{w}' - \mathbf{w}^*\|^2 + \mu_t^2 G^2 + \frac{\mu_t^2 L_1^2 \sigma^2 d}{4 \cdot 7^{k-1}} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}} - 2\mu_t \mathbb{E}[F(\mathbf{w}') - F(\mathbf{w}^*)] \tag{3.17}
\]

This leads to:

\[
\mathbb{E}[F(\mathbf{w}') - F(\mathbf{w}^*)] \leq \frac{1}{2\mu_t}(\mathbb{E}[\|\mathbf{w}' - \mathbf{w}^*\|^2 - \mathbb{E}[\|\mathbf{w}'^t - \mathbf{w}^*\|^2 + \mu_t^2 G^2 + \frac{\mu_t^2 L_1^2 \sigma^2 d}{4 \cdot 7^{k-1}} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}}]) \tag{3.18}
\]

Summing up for \( t \) from 1 to \( T \), we have:

\[
\sum_{t=1}^{T} \mathbb{E}[F(\mathbf{w}') - F(\mathbf{w}^*)] \leq \sum_{t=1}^{T} \frac{1}{2\mu_t}(\mathbb{E}[\|\mathbf{w}' - \mathbf{w}^*\|^2 - \mathbb{E}[\|\mathbf{w}'^t - \mathbf{w}^*\|^2 + \mu_t^2 G^2 + \frac{\mu_t^2 L_1^2 \sigma^2 d}{4 \cdot 7^{k-1}} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}}]) \leq \frac{1}{2\mu_1} \mathbb{E}[\|\mathbf{w}' - \mathbf{w}^*\|^2 + \sum_{t=2}^{T} (\frac{1}{2\mu_t} - \frac{1}{2\mu_{t-1}})\|\mathbf{w}' - \mathbf{w}^*\|^2 + \sum_{t=1}^{T} \mu_t \langle \frac{1}{2}G^2 + \frac{L_1^2 \sigma^2 d}{4 \cdot 7^{k-1}} + \frac{L_1^2 d}{2^{2c+2}} \rangle \tag{3.19}
\]
Since $\mu_t = \frac{\mu_1}{\sqrt{t}}$, we have $\sum_{t=1}^{T} \mu_t \leq \mu_1 \sqrt{T + 1}$, $\sqrt{t} - \sqrt{t - 1} = \frac{1}{\sqrt{t} + \sqrt{t - 1}} \leq 1$. Also due to the convexity of function $F(\cdot)$, $\mathbb{E}[F\left(\frac{1}{T} \sum_{t=1}^{T} w^t\right) - F(w^*)] \leq \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}[F(w^t) - F(w^*)]$. In addition, $||w^1 - w^*||^2 \leq D^2$. Therefore,

$$\mathbb{E}[F(\bar{w}) - F(w^*)] \leq \frac{D^2 (\sqrt{T} + 1)}{2T \mu_1} + \frac{\mu_1 \sqrt{T + 1}}{T} \left( \frac{1}{2} G^2 + \frac{L_1^2 \sigma^2 d}{4 \cdot 7^{k-1}} + \frac{L_2^2 d}{2^{2c+2}} \right)$$

When $T$ goes to infinity, the expected distance between $F(\bar{w})$ and $F(w^*)$ converges to zero at the rate of $\frac{1}{\sqrt{T}}$. This indicates that at each iteration, although the gradients are computed in an approximate way w.r.t. a close quantized point instead of the full-precision weights, the final trained solution will still converge to the optimal point.

Since we still need a one-step quantization for the weights at the end of the training process, there will be an extra error which remains positive as $T$ goes to infinity. Using equation 3.11, this can be obtained by:

$$\mathbb{E}[F(Q(\bar{w})) - F(w^*)] = \mathbb{E}[F(Q(\bar{w})) - F(\bar{w}) + F(\bar{w}) - F(w^*)]$$

$$\leq L_2 \frac{d}{2^{2c+2}} + \sigma^2 d + O\left(\frac{1}{\sqrt{T}}\right)$$

### 3.5 Training and Testing Experiments

We adopt the pattern of in-software training and on-hardware inference for DNNs. In this section, we describe our experiment setup, and compare conventional DNNs, LightNNs, and BNNs in terms of accuracy and storage. Hardware implementation results are described in subsequent sections.

#### 3.5.1 Accuracy

We first introduce the experiment set-up, including the DNN architectures, datasets, DNN configurations and training approaches. Then, the accuracy results of different architectures are reported and compared. The different DNN architectures, configurations and training settings are described as follows. Our goal is to involve (i) both Multi-Layer Perceptrons (MLPs) and Convolutional
Table 3.1: Architecture characteristics for the experiments.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Weights</th>
<th>Activation Function</th>
<th>Intermediate Results</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional DNN</td>
<td>32-bit floating point</td>
<td>ReLU</td>
<td>floating</td>
<td>floating</td>
</tr>
<tr>
<td>LightNN-2</td>
<td>$\pm (2^{-m_1} + 2^{-m_2})$, $m_1, m_2 = 0, 1, \ldots, 7$</td>
<td>ReLU</td>
<td>floating</td>
<td>floating</td>
</tr>
<tr>
<td>LightNN-1</td>
<td>$\pm 2^{-m}$, $m = 0, 1, \ldots, 7$</td>
<td>ReLU</td>
<td>floating</td>
<td>floating</td>
</tr>
<tr>
<td>BinaryConnect</td>
<td>$\pm 1$</td>
<td>ReLU</td>
<td>floating</td>
<td>floating</td>
</tr>
<tr>
<td>LightNN-2-bin</td>
<td>$\pm (2^{-m_1} + 2^{-m_2})$, $m_1, m_2 = 0, 1, \ldots, 7$</td>
<td>Sign</td>
<td>$\pm 1$</td>
<td>floating</td>
</tr>
<tr>
<td>LightNN-1-bin</td>
<td>$\pm 2^{-m}$, $m = 0, 1, \ldots, 7$</td>
<td>Sign</td>
<td>$\pm 1$</td>
<td>floating</td>
</tr>
<tr>
<td>BinaryNet</td>
<td>$\pm 1$</td>
<td>Sign</td>
<td>$\pm 1$</td>
<td>floating</td>
</tr>
</tbody>
</table>

Neural Networks (CNNs), (ii) both large and small network configurations, to make the results more representative.

**DNN architectures.** Seven DNN architectures are considered - conventional DNN, LightNN-2, LightNN-1, BinaryConnect, LightNN-2-bin, LightNN-1-bin, and BinaryNet. Table 3.1 describes their main characteristics. The ReLU activation function is adopted in the first four architectures, while the last three architectures use the hard tanh function for training and the sign function for testing.

**Datasets.** We test the seven architectures on two datasets - MNIST [77] and CIFAR-10 [74]. The MNIST dataset contains 70,000 gray-scale hand-written images, while CIFAR-10 contains 60,000 colored images for animals and vehicles.

**DNN configurations.** Both MLPs and CNNs are adopted for MNIST and CIFAR-10. We selected five configurations as shown in Table 3.2. The basic idea is to include both small and large DNN configurations, to determine how different architectures perform under varying configurations. 3-hidden for MNIST and 6-conv for CIFAR-10 are two large configurations used by Courbariaux et al. [62] [25]. 2-conv for MNIST and 3-conv for CIFAR-10 are two smaller configurations borrowed from Caffe examples [66]. 1-hidden for MNIST is a simple configuration adopted by prior research [112].

**Training approach.** The training algorithm is described in Section 3.3. Batch normalization
Table 3.2: The five architecture configurations for the MNIST and CIFAR-10 experiments.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Configuration</th>
<th>Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>MNIST</td>
<td>1-hidden</td>
<td>One hidden layer with 100 neurons</td>
</tr>
<tr>
<td></td>
<td>2-conv</td>
<td>Two convolutional layers and two</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fully-connected layers</td>
</tr>
<tr>
<td></td>
<td>3-hidden</td>
<td>Three hidden layers each with 4096 neurons</td>
</tr>
<tr>
<td>CIFAR-10</td>
<td>3-conv</td>
<td>Three convolutional layers and one</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fully-connected layer</td>
</tr>
<tr>
<td></td>
<td>6-conv</td>
<td>Six convolutional layers and three</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fully-connected layers</td>
</tr>
</tbody>
</table>

Table 3.3: Test error and the number of parameters for all DNN architectures.

<table>
<thead>
<tr>
<th></th>
<th>MNIST</th>
<th>CIFAR-10</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of parameters</td>
<td>1 hidden</td>
<td>2-conv</td>
<td>3 hidden</td>
<td>3-conv</td>
<td>6-conv</td>
</tr>
<tr>
<td>Conventional</td>
<td>79,510</td>
<td>431,080</td>
<td>36,818,954</td>
<td>82,208</td>
<td>39,191,690</td>
</tr>
<tr>
<td>LightNN-2</td>
<td>1.72%</td>
<td>0.86%</td>
<td>0.75%</td>
<td>21.16%</td>
<td>8.86%</td>
</tr>
<tr>
<td>LightNN-1</td>
<td>1.86%</td>
<td>1.29%</td>
<td>0.83%</td>
<td>24.62%</td>
<td>8.84%</td>
</tr>
<tr>
<td>BinaryConnect</td>
<td>2.09%</td>
<td>2.31%</td>
<td>0.89%</td>
<td>26.11%</td>
<td>8.79%</td>
</tr>
<tr>
<td>LightNN-2-bin</td>
<td>4.10%</td>
<td>4.63%</td>
<td>1.29%</td>
<td>43.22%</td>
<td>9.90%</td>
</tr>
<tr>
<td>LightNN-1-bin</td>
<td>2.94%</td>
<td>1.67%</td>
<td>0.89%</td>
<td>32.58%</td>
<td>10.12%</td>
</tr>
<tr>
<td>BinaryNet</td>
<td>3.10%</td>
<td>1.86%</td>
<td>0.94%</td>
<td>36.56%</td>
<td>9.05%</td>
</tr>
<tr>
<td></td>
<td>6.79%</td>
<td>3.16%</td>
<td>0.96%</td>
<td>73.82%</td>
<td>11.40%</td>
</tr>
</tbody>
</table>

[65] and dropout [118] techniques are adopted to accelerate training and avoid overfitting, respectively. The dataset is divided into training set, validation set, and test set. The validation set is used for selecting the best epoch. The same number of total training epochs is applied to all architectures and the test error of the epoch with the lowest validation error is reported. These architectures are trained on Theano platform [129]. We use existing open source models [61] to train the conventional DNN, BinaryConnect and BinaryNet. Finally, the hinge loss function [43] and ADAM learning rule [71] are used to train all seven architectures [71].

Results  Results for different DNN architectures and configurations are presented in Table 3.3.

For most configurations (except a few), the accuracy decreases from: conventional, LightNN-2,
LightNN-1, LightNN-2-bin, LightNN-1-bin, BinaryConnect, BinaryNet. This is because when we constrain the weights and activations, the architecture suffers from varying levels of accuracy loss.

### 3.5.2 Storage Requirements

The weight storage requirements for different DNN architectures are compared in Figure 3.2. Since the constraint on activations does not affect weight storage, we only show four DNN architectures. While it has been shown that limited-bit precision can also lead to sufficient accuracy [45], we retain the 32-bit representation for the conventional DNN as baseline. Therefore, a weight in conventional DNNs has four bytes, while BinaryConnect and BinaryNet only require one bit. To store a weight $w = \pm 2^{-m}$, LightNN-1 and LightNN-1-bin need four bits: one bit for $\text{sign}(w)$ and another three bits for $|m|$. LightNN-2 and LightNN-2-bin need seven bits for a weight $w = \pm (2^{-m_1} + 2^{-m_2})$: one bit for $\text{sign}(w)$, three bits for $|m_1|$, and three bits for $|m_2|$. For easier hardware implementation, one byte is used for LightNN-2 or LightNN-2-bin weights. Storage affects the number of memory accesses, and is therefore essential to energy consumption, which is investigated in Section 3.6.
3.5.3 ImageNet Results

To see if LightNNs can perform well on large-scale datasets and large networks, we conduct experiments on ImageNet dataset [27] with AlexNet [75] and VGG-16 [115]. The training settings for LightNN are the same as conventional DNNs, except for the customized learning rates. For AlexNet, we train LightNN from scratch, while for VGG-16 we train from pre-trained full-precision weights due to time constraints. LightNN-1 is used for experiment, where each weight is constrained to be \{±2^0, ±2^{-1}, ..., ±2^{-7}\}. The results are shown in Table 3.4. The test error for conventional DNNs is directly borrowed from prior work [75][115]. LightNN-1 can achieve slightly better accuracy than conventional DNNs for AlexNet, and have competitive accuracy for VGG-16. We also conduct experiment on ResNet, which has much fewer parameters than AlexNet and VGG-16, and find accuracy degradation compared to the full-precision model. This indicates that LightNNs work better for the over-parameterized models.

3.5.4 Image Segmentation Results

In addition to image classification, we also conduct experiment for the image segmentation application. We first experiment on the Pascal VOC2012 dataset [36], which has 8498 training images with
Table 3.4: Test error for conventional DNNs, LightNN-1 and BinaryConnect on ImageNet. The accuracy for BinaryConnect is borrowed from Rastegari et al. [104] where only AlexNet results are available.

<table>
<thead>
<tr>
<th>Model</th>
<th>AlexNet</th>
<th>VGG-16</th>
<th>ResNet-18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional</td>
<td>43.4%</td>
<td>19.8%</td>
<td>27.0%</td>
</tr>
<tr>
<td>LightNN-1</td>
<td>41.4%</td>
<td>19.7%</td>
<td>28.2%</td>
</tr>
<tr>
<td>BinaryConnect</td>
<td>64.6%</td>
<td>39.0%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 3.5: Accuracy results for conventional DNNs and LightNNs on Pascal VOC dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>PLA</th>
<th>ACA</th>
<th>mIoU</th>
<th>fwavacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional</td>
<td>91.1%</td>
<td>77.8%</td>
<td>65.0%</td>
<td>84.5%</td>
</tr>
<tr>
<td>LightNN-2</td>
<td>90.5%</td>
<td>79.3%</td>
<td>64.2%</td>
<td>83.8%</td>
</tr>
<tr>
<td>LightNN-1</td>
<td>90.4%</td>
<td>77.6%</td>
<td>63.2%</td>
<td>83.5%</td>
</tr>
</tbody>
</table>

20 object classes and background. We use a small baseline model, FCN-8s [88], which uses a fully convolutional network for the backbone and the decoder. We use four metrics to measure the accuracy. **Pixel-level accuracy (PLA)** is the average classification of each pixel. **Average class accuracy (ACA)** first computes the pixel-level accuracy per class, and then, averages them. **Mean intersection of union (mIoU)** first computes $\text{IoU} = \frac{TP}{TP + FP + FN}$ for each class where TP, FP and FN indicate the number of true positives, false positives and false negatives, and averages them across classes. **Frequency weighted average accuracy (fwavacc)** averages IoUs across classes, weighted by the prediction frequency of each class.

We also experiment on the CityScapes dataset [24] which consist of 5000 images with fine-grain annotations. Each pixel has a label, which can be one of 30 classes. The FCN-8s backbone is used. The results are shown in Table 3.6. The accuracy degradation caused by LightNN quantization is non-negligible, similar to the case of the 3-conv network for CIFAR-10 dataset.
Table 3.6: Accuracy results for conventional DNNs and LightNNs on Cityscapes dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>PLA</th>
<th>ACA</th>
<th>mIoU</th>
<th>fwavacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional</td>
<td>53.8%</td>
<td>43.8%</td>
<td>23.1%</td>
<td>45.7%</td>
</tr>
<tr>
<td>LightNN-2</td>
<td>51.0%</td>
<td>39.5%</td>
<td>19.7%</td>
<td>42.6%</td>
</tr>
<tr>
<td>LightNN-1</td>
<td>47.2%</td>
<td>36.5%</td>
<td>16.8%</td>
<td>38.6%</td>
</tr>
</tbody>
</table>

Table 3.7: Comparison of LightNN and fixed-point weights in terms of weight storage and test error. Network size indicates the number of filters for the largest convolutional layer in the network. "xW" means x bits for weights, and "yA" means y bits for activations.

| Model          | Network size | | | | |
|----------------|--------------|-----------------|-----------------|-----------------|-----------------|-----------------|
|                | 1536 Storage | 1536 Error | 512 Storage | 512 Error | 128 Storage | 128 Error | 64 Storage | 64 Error |
| Conventional   | 165MB        | 6.45% | 19MB | 7.12% | 1MB | 10.59% | 313KB | 14.02% |
| Fixed-point8W8A| 41MB         | 6.46% | 5MB | 7.33% | 0.3MB | 10.68% | 78KB | 14.58% |
| Fixed-point4W8A| 20MB         | 6.57% | 2MB | 7.35% | 0.1MB | 10.91% | 39KB | 16.50% |
| LightNN-14W8A  | 20MB         | 6.49% | 2MB | 7.14% | 0.1MB | 10.69% | 39KB | 15.40% |
| BinaryConnect  | 5MB          | 6.89% | 0.6MB | 8.77% | 0.04MB | 15.28% | 10KB | 23.14% |

3.5.5 Comparison with Fixed-point Weights

Fixed-point quantization [45][151] for weights and activations has also been used to reduce storage and energy for DNNs. LightNNs, as a variant of quantized DNNs, are compatible with fixed-point activation quantization. We compare the fixed-point weights and LightNN, both with 8-bit fixed-point activations, in Table 3.7. We start from the network configuration by Hubara [59], which is a VGG-like network whose “widest” convolutional layer has 1536 filters, and then reduce the number of filters per layer to see the model accuracy at different network sizes. We have also changed the activation function from ReLU to leaky ReLU [90] since the latter achieves higher accuracy for all the models. From Table 3.7, LightNN-14W8A can achieve slightly higher accuracy than fixed-point4W8A while they have the same weight storage. Further comparison regarding their FPGA resources is shown in Table 3.14.
Table 3.8: Per-weight bit storage (PWBS) and test error at varied $k$ (number of shifts-and-add operators for an equivalent multiplier).

<table>
<thead>
<tr>
<th>$k$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWBS</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>20</td>
<td>32</td>
</tr>
<tr>
<td>Error (%)</td>
<td>26.01</td>
<td>22.01</td>
<td>21.19</td>
<td>20.97</td>
<td>20.89</td>
</tr>
</tbody>
</table>

Table 3.9: Per-weight bit storage (PWBS) and test error of LightNN-1 at varied $c$ (maximum number of bits to shift).

<table>
<thead>
<tr>
<th>$c$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>7</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWBS</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Error (%)</td>
<td>38.51</td>
<td>33.04</td>
<td>26.32</td>
<td>26.01</td>
<td>25.88</td>
</tr>
</tbody>
</table>

3.5.6 Varying Number of Shifts

Since the number of shift-and-add operators, $k$, determines the approximation accuracy per weight, it is worthwhile to analyze how accuracy changes when $k$ changes. We conduct the experiment on a customized small convolutional network with only 19738 parameters, so the gap between $k = 1$ and $k = 32$ is large. We constrain the maximum bits per shift to be 7, which is the same setting used for LightNNs in Table 3.1. The results are shown in Table 3.8. As expected, larger $k$ has higher accuracy, yet in this case the accuracy for $k = 3$ is close to $k = 32$. The per-weight bit storage (PWBS) for $k = 32$ is 32 (not 128) because it can record which bits to shift as the 32-bit fixed-point weight usually does.

3.5.7 Varying Number of Shifting Bits

It is also interesting to see how the maximum number of shifting bits, $c$, affects the accuracy. We gradually increase $c$ for LightNN-1 from 1 to 15, and show the results in Table 3.9. The small network used by Table 3.8 is also adopted here. For this case, the accuracy for $c = 7$ is close to $c = 15$ but can use four bits per weight instead of five bits.
3.6 Hardware Evaluation

In this section, we show the gate-level circuit simulation results for energy and area across different DNN architectures and configurations. We also present a guideline for selecting DNN architectures for hardware implementations to satisfy the varying requirements of different applications.

3.6.1 Set-Up

For all seven DNN architectures under consideration, we designed pipelined implementations with one stage per neuron, where the computation unit is reused for each neuron. The weights and inputs are initially stored in the memory, and fetched into the pipeline stage to compute the output for all neurons in that layer; intermediate results are written back. A 65nm commercial standard library is adopted. The logic computations circuit of one neuron is constructed using Synopsys Designware commercial IP [116] (e.g., floating point multiplication and addition). The Synopsys Design Compiler [91] is used to generate the gate-level netlist and measure the circuit area. The power consumption of one neuron circuit is calculated using Synopsys Primetime [64]. Cacti [99] is used to obtain the energy of memory accesses and registers. While prior work describes approaches (such as data reuse) to optimize the CNN hardware implementation [3], we keep all the DNN architectures implemented an unoptimized fashion because our main objective is to compare how constraining weights impacts both computation and memory access energy. The size of the register files is chosen to accommodate the data size required for the computation of the largest neuron.

3.6.2 Multipliers vs. Approximate Multiply Units

Energy and area of each multiplier or approximate multiply unit in all DNN architectures under consideration are explored first. For BinaryConnect and BinaryNet, a multiply unit is simply an XNOR gate [62]. For LightNN-1 and LightNN-1-bin, it is a shift unit. Since operands (e.g., unbinarized weights, activations and inputs) are represented as single-precision float-point., the shift operation is equivalent to an integer addition for the exponent. LightNN-2 and LightNN-2-bin both rely on two shifts and an add operation. The adder required by LightNN-2 is floating point, while
LightNN-2-bin only needs an integer adder to perform fixed-point addition. The area and energy consumption are reported in Figure 3.3.

3.6.3 Energy Consumption

Figure 3.4 shows the comparison of energy consumption for all seven DNN architectures considered. Under the same DNN configuration, conventional DNNs and BinaryNet are always the most and least energy-consuming architecture, respectively. Furthermore, LightNN-2 is more energy-consuming than LightNN-1 and LightNN-2-bin, both of which consume more energy than LightNN-1-bin. When comparing LightNN-1 and LightNN-2-bin, the former has fewer bits for each weight, and therefore consumes less energy for each memory access, while the latter has more energy-efficient logic. The results in Figure 3.4 show that LightNN-1 has higher energy consumption than LightNN-2-bin in all configurations except MNIST 1-hidden. The same comparison holds for the BinaryConnect and LightNN-1-bin, where BinaryConnect has more energy-consuming logic circuitry (e.g., floating point adder) while LightNN-1-bin has larger weight storage.

Although BinaryNet always has the lowest energy consumption under the same configuration, its high accuracy only occurs when the configuration is very large. For example, a conventional DNN with 2-conv configuration can surpass BinaryNet with 3-hidden configuration in terms of both accuracy and energy consumption for MNIST dataset.

To explore the energy composition for each DNN architecture, we report the results in Figure 3.5. Specifically, the various components of energy are averaged across different configurations and datasets. For the conventional DNNs with floating point circuitry, the most energy-consuming part is the computational portions, while the majority of energy in LightNN-2-bin, LightNN-1-bin and BinaryNet is consumed by memory accesses, though the absolute values are still smaller than that of conventional DNNs. We also break down the logic energy into leakage, switch and internal energy, where the switch energy is caused by switched load capacitance, and the internal energy is due to internal device switching. The leakage, switch and internal energy take 31.6%, 35.2% and 33.2% respectively, averaged on all DNN architectures and configurations.
CHAPTER 3. HARDWARE-EFFICIENT INFERENCE VIA LIGHTNNS

Figure 3.4: Comparison of energy consumption of different DNN architectures under varying datasets/configurations. Energy consumption is measured for inferring a single image.

Figure 3.5: Energy breakdown for all DNN architectures, averaged across different datasets and configurations.

3.6.4 Area Comparison

We also compare the area of the seven DNN architectures under varying configurations in Figure 3.6. Note that the reported area includes both logic circuits and register files. Also note that the DNN inference is implemented in a pipeline fashion, and the logic is set to handle the largest neuron count in each configuration. Since more computation modules indicate larger area, but fewer memory fetches, the absolute values for the area encompass the energy consumption reported in Figure 3.4. However, the comparison of different DNN architectures within a configuration is
still meaningful since they use the same (largest) neuron count. The DNN architectures follow a consistent order (from larger area to smaller): Conventional DNNs, LightNN-2, LightNN-1, BinaryConnect, LightNN-2-bin, LightNN-1-bin, and BinaryNet.

3.6.5 Guidelines for Architecture Selection

An interesting question is whether there is a DNN architecture that always surpasses another in terms of both accuracy and energy, under varying configurations and datasets? The answer is no. However, with constraints on accuracy or energy, some DNN architectures are more preferable than others. Suppose one will implement DNN for MNIST on hardware with the constraint that energy consumption per inference is below 200nJ. In this case, the LightNN-2 architecture with 2-conv configuration is the best, since it has the highest accuracy.

Does higher accuracy always require higher energy? Again, the answer is no. A comparison of different architectures and configurations are presented in Figure 3.7. Only the red triangles are the Pareto-optimal ones in terms of accuracy and energy.
3.6.6 Non-pipeline Implementation

To further explore how LightNNs perform in a non-pipeline implementation, we implement the ANNs for five benchmarks from the UCI machine learning repository [5]: abalone, banknote-authentication, transfusion, sinknonsink, and balance-scale. They are chosen because of their small ANN configurations, making direct implementations of whole ANNs possible. The size of the datasets varies from 600 (balance) to 200,000 (sinknonsink), so a greater coverage is ensured. Instead of computing each neuron at one stage, the non-pipeline implementation builds the whole ANN for each dataset. Moreover, to confirm that LightNNs are compatible with the use of limited bit precision for inputs and intermediate results [45], we use both 32-bit and 12-bit implementations. Similar to Section 3.6, the ANNs are implemented using Synopsys Designware commercial IP [116].

Accuracy Table 3.10 shows the accuracy for the testing phase for the five benchmarks. All of them are trained with 2-ones stochastic approximation. For each benchmark, the accuracy of the 12-bit and 32-bit configurations are shown for both the conventional DNNs and LightNNs. From an accuracy standpoint, LightNNs lose only 0.66% and 0.40% accuracy when compared to conventional DNNs for 32- and 12-bit implementations, respectively, on average across the five benchmarks. Furthermore, the accuracy results for 32- and 12-bit data confirm that the additional
error incurred by limiting numerical precision is quite small [45]. Finally, the small accuracy differences between 32-bit and 12-bit LightNNs prove that they have high tolerance for limited precision, thereby showing their compatibility with prior work [45] [131] [148].

**Energy and Area** 12-bit conventional DNNs and LightNNs for the five benchmarks are implemented and all three types of energy consumption are measured: leakage, internal, and switch energy, all shown in Figure 3.8. The area is also compared in Figure 3.9. Note that the energy and area are reported only for the logic. The use of LightNNs reduces total energy by 38.8% on average. Both leakage and dynamic energy are reduced, benefiting from the more energy efficient
3.7 FPGA Implementation

In this section, the FPGA implementation of conventional DNNs and LightNN are compared. Our goal is to demonstrate the quantization method provided by LightNN leads to a resource efficient and low latency FPGA implementation. There are no detailed analytic models of FPGA architectures and circuitry, so we must evaluate architectures with actual mapped hardware. However, for any DNNs implementation, there are many potential solutions that result in a large design space. Work by Zhang et al. [145] finds that there could be as much as 90% performance difference between two different solutions with the same FPGA logic utilization. It is non-trivial to determine the optimal solution, especially when limitations on computation resources and memory bandwidth of the FPGA platform are considered. Therefore, in this experiment, to evaluate the efficacy, we will use the same design optimization solution for both conventional DNNs and LightNN.

3.7.1 HLS Accelerator Implementation

The Vivado high-level synthesis (HLS) [137] is used to describe the functionality for each layer within a conventional DNN and a LightNN. A primary capability of Vivado HLS lies in the rapid assessment of resources, throughput, and performance tradeoffs. The FPGA implementation is produced according to pragma and directives inserted into the C code (or into a separate directive
**Algorithm 2:** Pseudo code of a convolutional layer implementation using Vivado HLS

**Input:** weights \( W \), inputs for layer \( I \), tiling size for output channels \( T_m \), tiling size for input channels \( T_n \), tiling size for the width of input volume \( T_c \), tiling size for the height of input volume \( T_r \).

**Output:** outputs for layer \( O \).

```plaintext
for (\( t_o = 0; t_o < M; t_o += T_m \)) {
    for (\( t_i = 0; t_i < N; t_i += T_n \)) {
        for (\( row_i = 0; row_i < R; row_i += T_r \)) {
            for (\( col_i = 0; col_i < C; col_i += T_c \)) {
                \( I_t = I[t_i : t_i + T_n][row_i : row_i + T_r][col_i : col_i + T_c] \) // load tiled inputs
                \( W_t = W[t_o : t_o + T_m][t_i : t_i + T_n][0 : K][0 : K] \) // load tiled weights
                for (\( row_k = 0; row_k < K; row_k += 1 \)) {
                    for (\( col_k = 0; col_k < K; col_k += 1 \)) {
                        \#pragma HLS pipeline
                        \#pragma HLS UNROLL
                        for (\( tt_o = t_o; tt_o < t_o + T_m; tt_o += 1 \)) {
                            for (\( tt_i = t_i; tt_i < t_i + T_n; tt_i += 1 \)) {
                                for (\( tt_k = 0; tt_k < K; tt_k += 1 \)) {
                                    \#pragma HLS UNROLL
                                    \( O[t_o][tt_o][tt_i][row_k][col_k] \) += \( W_t[tt_o][tt_i][0 : K][col_k] \times I_t[tt_t][S \times row + row_k][S \times col + col_k] \);
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

In addition, a previous study [145] proved that convolution operations typically take over 90% of the computation time. Therefore, in this work, we focus on the comparison of convolutional layer implementation in conventional DNN and LightNN. More specific design optimization solution for LightNN will be studied in future work.

The pseudo code shown in Algorithm 2 illustrates the detailed design optimization solution for a convolutional layer. The convolutional layer transforms a 3-D input volume to a 3-D output volume of neuron activations. For example, the depth of the 3-D input volume shown in Algorithm 2 is \( N \). In other words, there are \( N \) input channels. Similarly, the depth of the 3-D output volume shown in Algorithm 2 is \( M \), which means that there are \( M \) output channels. The parameters of the convolutional layer consist of a set of trainable weight kernels. Every weight kernel is small spatially (along width and height), but extends through the full depth of the input volume. For example, a
weight kernel of the convolutional layer shown in Algorithm 2 has dimensions $N \times K \times K$ (i.e., $K$ pixels width and height and $N$ input channels because input volume has depth $N$). Although the convolutional layer significantly reduces data volumes by weight sharing, loop tiling is still mandatory to fit a small portion of data on-chip for FPGA implementation. Therefore, we begin by applying loop tiling for the 3-D input volume. To be specific, the depth of the input volume is tiled by $T_n$, the height of the input volume is tiled by $T_r$ and the width of the input volume is tiled by $T_c$ (shown in lines 3 to 4 in Algorithm 2). Therefore, the input volume is tiled into small blocks with dimensions $T_n \times T_r \times T_c$. The depth of the 3-D output volume is tiled by $T_m$ as well (shown in line 1 in Algorithm 2). Then, the tiled inputs, weights (and biases which we did not show in Algorithm 2 for simplicity) are loaded on-chip. However, an improper tiling may degrade the efficiency of data reuse and parallelism of data processing. Thus, different tiling settings (such as, different value of $T_m$, $T_n$, $T_r$ and $T_c$) are exercised to identify the optimal solution in terms of the timing latency and resource usage.

Table 3.11: Data sharing relations of convolutional operations.

<table>
<thead>
<tr>
<th>Loop iterator</th>
<th>Tiled inputs</th>
<th>Tiled weights</th>
<th>Tiled outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$row_k$</td>
<td>dependent</td>
<td>independent</td>
<td>irrelevant</td>
</tr>
<tr>
<td>$col_k$</td>
<td>dependent</td>
<td>independent</td>
<td>irrelevant</td>
</tr>
<tr>
<td>$row$</td>
<td>dependent</td>
<td>irrelevant</td>
<td>independent</td>
</tr>
<tr>
<td>$col$</td>
<td>dependent</td>
<td>irrelevant</td>
<td>independent</td>
</tr>
<tr>
<td>$tt_o$</td>
<td>irrelevant</td>
<td>independent</td>
<td>independent</td>
</tr>
<tr>
<td>$tt_i$</td>
<td>independent</td>
<td>independent</td>
<td>irrelevant</td>
</tr>
</tbody>
</table>

Second, the computation engine is optimized as well for the tiled data. In Algorithm 2, the objective of computation optimization is to enable efficient loop unrolling (shown in line 13 and 15) and pipelining (shown in line 11) while fully utilizing all computation resources provided by the FPGA hardware platform. Loop unrolling can be used to increase the utilization of massive computation resources in FPGA devices. Unrolling along different loop dimensions will generate different implementation variants. Whether and to what extent two unrolling execution instances share data will affect the complexity of generated hardware, and eventually affect the number of unrolled copies and the hardware operation frequency.
Table 3.11 shows the data sharing relation of convolutional computation. The data sharing relations between different loop iterations of a loop dimension on a given array can be classified into three categories, irrelevant, independent and dependent [145]:

- **Irrelevant.** If loop iterator \( i \) does not appear in any access function of an array \( A \), the corresponding loop dimension \( i \) is irrelevant to array \( A \). For example, loop iterator \( \text{row} \) does not appear in any access of the tiled weight \( W_t \) shown in Algorithm 2. Therefore, the corresponding loop of loop iterator \( \text{row} \) is irrelevant to the tiled weight \( W_t \).

- **Independent.** If the data space accessed on an array \( A \) is totally separable along a certain loop dimension \( i \), the loop dimension \( i \) is independent of array \( A \). For example, the first two dimensions accessed on the tiled weight \( W_t \) are totally separable along the certain loops whose iterators are \( tt_o \) and \( tt_i \), respectively. Therefore, the corresponding loop of iterators \( tt_o \) and \( tt_i \) is independent of the tiled weight \( W_t \).

- **Dependent.** If the data space accessed on an array is not separable along a certain loop dimension \( i \), the loop dimension \( i \) is dependent on \( A \). For example, the last two dimensions accessed on the tiled input \( I_t \) cannot be separable along the certain loops whose loop iterators are \( \text{row}, \text{col}, \text{row}_k \) and \( \text{col}_k \). Therefore, the corresponding loop of the iterator \( \text{row}, \text{col}, \text{row}_k \) and \( \text{col}_k \) is dependent on the tiled input \( I_t \).

An independent data sharing relation generates direct connections between buffers and computation engines. An irrelevant data sharing relation generates broadcast connections. A dependent data sharing relation generates interconnections with complex topology. The data sharing relations of the pseudo code is shown in Table 3.11. Loop dimensions \( tt_o \) and \( tt_i \) are selected to be unrolled to avoid complex connection topologies for all arrays. \( tt_o \) and \( tt_i \) are permuted to the inner most loop levels to simplify HLS code generation.

On the other hand, loop pipelining is another key optimization technique in HLS to improve system throughput by overlapping the execution of operations from different loop iterations. The throughput achieved is limited both by resource constraints and data dependencies in the application.
Table 3.12: DNN configuration for CIFAR-10.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Input channel</th>
<th>Output channel</th>
<th>Output dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv1</td>
<td>3</td>
<td>128</td>
<td>32</td>
</tr>
<tr>
<td>Conv2</td>
<td>128</td>
<td>128</td>
<td>32</td>
</tr>
<tr>
<td>Conv3</td>
<td>128</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>Conv4</td>
<td>256</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>Conv5</td>
<td>256</td>
<td>512</td>
<td>8</td>
</tr>
<tr>
<td>Conv6</td>
<td>512</td>
<td>512</td>
<td>8</td>
</tr>
<tr>
<td>FC1</td>
<td>8192</td>
<td>1024</td>
<td>1</td>
</tr>
<tr>
<td>FC2</td>
<td>1024</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>FC3</td>
<td>1024</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.13: Timing performance comparison between conventional DNN, fixed-point DNN and LightNN. FP<sub>4W8A</sub> is short for Fixed-point<sub>4W8A</sub>, i.e., fixed-point network with 4-bit weights and 8-bit activations. LN-1<sub>4W8A</sub> refers to LightNN-1<sub>4W8A</sub>, i.e., LightNN-1 with 4-bit weights and 8-bit activations. Both LN-1<sub>4W8A</sub> and FP<sub>4W8A</sub> can achieve higher speed without running out of FPGA resources.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Tiling size</th>
<th>Execution cycles</th>
<th>Speedup†</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T&lt;sub&gt;m&lt;/sub&gt;</td>
<td>T&lt;sub&gt;n&lt;/sub&gt;</td>
<td>T&lt;sub&gt;r&lt;/sub&gt;</td>
</tr>
<tr>
<td>Conv1</td>
<td>128</td>
<td>3</td>
<td>32</td>
</tr>
<tr>
<td>Conv2</td>
<td>128</td>
<td>128</td>
<td>32</td>
</tr>
<tr>
<td>Conv3</td>
<td>256</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>Conv4</td>
<td>256</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>Conv5</td>
<td>512</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>Conv6</td>
<td>512</td>
<td>512</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>128</td>
<td>8</td>
</tr>
</tbody>
</table>

* N/A indicates that FPGA runs out of its resource for achieving the corresponding tiled setting.
† Speedup describes the speed increase for the FP<sub>4W8A</sub> and the LN-1<sub>4W8A</sub> compared to conventional DNN, under their best tiling size for each layer, respectively.

Loop-carried dependence will prevent loops from being fully pipelined. According to the analysis by Zhang et al. [145], the loop col can be fully pipeline for achieving better performance.
3.7.2 Experiment Setup

As described previously, conventional DNNs and LightNN are implemented with Vivado HLS [137]. The C code of DNN designs are parallelized by adding HLS-defined pragma and the parallel version is validated with the Vivado HLS timing analysis tool. This tool enables fast pre-synthesis simulation for the implemented design. Pre-synthesis resource reports are used for design space exploration and performance estimation. Our implementation is built on the VC709 board which has a Xilinx FPGA chip Virtex7 485t. Its working frequency is 100 MHz. Pre-synthesis is executed on an Intel i7-4790 CPU (3.6GHz) with 16GB RAM.

Table 3.12 summarizes the detailed DNN configuration used in this experiment. This configuration is the 6-conv for CIFAR-10 dataset from Section 3.5, which contains sixty thousand 32x32 3-channel images. The configuration consists of six convolutional layers followed by three fully-connected layers. All convolutional layers use 3x3 filters and edge padding.

3.7.3 Experimental Results

In this section, we first compare design space exploration results between conventional DNNs and LightNN. Then, the resource usage comparison among different design solutions is reported.

Table 3.13 shows different tile size tuples \(<T_m, T_n, T_r, T_c,>\) and their corresponding execution cycles for the DNN configuration shown in Table 3.12. The execution cycles are reported by Vivado pre-synthesis simulation for three different quantization methods: (i) Conventional, which encodes the weight and output of a neuron as 32-bit floating point numbers; (ii) FP\(_{4W8A}\), which uses the 4-bit fixed point representation for weights and the 8-bit fixed point representation for intermediate results; (iii) LN-1\(_{4W8A}\), which uses the 4-bit predefined format to represent weights and the 8-bit fixed-point representation for the intermediate results. The N/A entries in Table 3.13 show that FPGA resources are exhausted for the corresponding tiled setting. The first observation from Table 3.13 shows that with a large tiling size, DNNs can achieve significant advantage in timing latency. Another observation is that for the same tiling size, the FP\(_{4W8A}\) and the LN-1\(_{4W8A}\) have an advantage in execution cycles, which leads to smaller latency. However, the latency improvement achieved for
Table 3.14: FPGA resource usage comparison between conventional DNN, fixed-point DNN and LightNN. FP\(_{4W8A}\) is short for Fixed-point\(_{4W8A}\), i.e., fixed-point network with 4-bit weights and 8-bit activations. LN-1\(_{4W8A}\) refers to LightNN-1\(_{4W8A}\), i.e., LightNN-1 with 4-bit weights and 8-bit activations. With the same tiling size, LightNN-1 requires fewer resources than conventional DNN and fixed-point DNN.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Tiling size</th>
<th>DNN architecture</th>
<th>FPGA resource</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>BRAM</td>
</tr>
<tr>
<td>Conv1</td>
<td>128 3 32 32</td>
<td>Conventional</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>13</td>
</tr>
<tr>
<td>Conv2</td>
<td>128 128 32 32</td>
<td>Conventional</td>
<td>1,040</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>448</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>448</td>
</tr>
<tr>
<td>Conv3</td>
<td>256 128 16 16</td>
<td>Conventional</td>
<td>1,280</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td>128 128 16 16</td>
<td>Conventional</td>
<td>768</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>416</td>
</tr>
<tr>
<td>Conv4</td>
<td>256 256 16 16</td>
<td>Conventional</td>
<td>2,432</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>800</td>
</tr>
<tr>
<td></td>
<td>128 128 16 16</td>
<td>Conventional</td>
<td>768</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>416</td>
</tr>
<tr>
<td>Conv5</td>
<td>512 256 8 8</td>
<td>Conventional</td>
<td>N/A*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>1,040</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>1,040</td>
</tr>
<tr>
<td></td>
<td>256 256 8 8</td>
<td>Conventional</td>
<td>2,368</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>528</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>528</td>
</tr>
<tr>
<td></td>
<td>128 128 8 8</td>
<td>Conventional</td>
<td>704</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>272</td>
</tr>
<tr>
<td>Conv6</td>
<td>512 512 8 8</td>
<td>Conventional</td>
<td>N/A*</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>1,040</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>1,040</td>
</tr>
<tr>
<td></td>
<td>256 256 8 8</td>
<td>Conventional</td>
<td>2,368</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>528</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>528</td>
</tr>
<tr>
<td></td>
<td>128 128 8 8</td>
<td>Conventional</td>
<td>704</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FP(_{4W8A})</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LN-1(_{4W8A})</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td>Available resource</td>
<td></td>
<td>2,940</td>
</tr>
</tbody>
</table>

* N/A indicates that FPGA runs out of its resource for achieving the corresponding tiled setting.
Table 3.15: FPGA resource utilization, latency and test error comparison.

<table>
<thead>
<tr>
<th>DNN architecture</th>
<th>Maximum resource utilization</th>
<th>Normalized latency*</th>
<th>Test error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BRAM</td>
<td>DSP</td>
<td>FF</td>
</tr>
<tr>
<td>Conventional</td>
<td>80.5%</td>
<td>71.1%</td>
<td>27.8%</td>
</tr>
<tr>
<td>FP&lt;sub&gt;4W8A&lt;/sub&gt;</td>
<td>35.3%</td>
<td>28.4%</td>
<td>1.98%</td>
</tr>
<tr>
<td>LN-1&lt;sub&gt;4W8A&lt;/sub&gt;</td>
<td>35.3%</td>
<td>0.03%</td>
<td>1.09%</td>
</tr>
</tbody>
</table>

* Latency is computed by summing up the lowest latency for six convolutional layers under FPGA resource constraint, and normalized by the latency of conventional DNN.

the same tiling size between the conventional DNN and the LN-1<sub>4W8A</sub> is not as large as expected. A possible explanation may be latency improves for the optimized design of the conventional DNN but not LN-1<sub>4W8A</sub>. Therefore, a more specific design optimization solution for the LN-1<sub>4W8A</sub> should be sought for latency improvement.

Table 3.13 also shows the speedup for the FP<sub>4W8A</sub> and the LN-1<sub>4W8A</sub> compared to the conventional DNN, where the best tiling size for each DNN is used. Since the conventional DNN requires much more resources than the FP<sub>4W8A</sub> and the LN-1<sub>4W8A</sub> with the same tiling size (more detail shown in Table 3.14), the conventional DNN runs out of FPGA resources when increasing the tiling size to 512 × 512 × 8 × 8 for the largest two layers (Conv5 and Conv6). Therefore, the FP<sub>4W8A</sub> and the LN-1<sub>4W8A</sub>, which reduce the resource requirement, allow larger tiling sizes, and therefore reduce the latency. In addition, we notice that FP<sub>4W8A</sub> and the LN-1<sub>4W8A</sub> have similar speed ups, which might also be explained by the fact that optimized design for the conventional DNN doesn’t benefit from the significant reduction in FPGA resources that LN-1<sub>4W8A</sub> has.

To be specific, Table 3.14 illustrates the FPGA resource usage comparison between the conventional DNN, the FP<sub>4W8A</sub> and the LN-1<sub>4W8A</sub>. In more detail, the BRAM, DSP, flip-flop and LUT usage are reported by the Vivado tool for different tiling settings. The last row of the Table 3.14 shows the available resources for the Virtex7-VC709 FPGA board used in this experiment. Similar to Table 3.13, an N/A entry means that the FPGA resources are exhausted for the corresponding tiling size. The first observation from Table 3.14 shows that with the same tiling size, the FP<sub>4W8A</sub> and the LN-1<sub>4W8A</sub> require less FPGA resources than the conventional DNN. These reductions can
be explained by the fact that the quantization methods provided by the \( FP_{4W8A} \) and the \( LN-1_{4W8A} \) simplify the computation of DNN. Another observation is that with the same tiling size, the \( LN-1_{4W8A} \) require less DSPs and FFs but more LUTs than the \( FP_{4W8A} \). This is due to the fact that the \( LN-1_{4W8A} \) utilizes the LUT to implement the described multiplier replacement (shift operation) instead of the standard DSP unit. Overall, Table 3.14 indicates that LightNN-1 can reduce latency by using a larger tiling size, while requiring fewer FPGA resources.

Table 4.6 compares three DNNs, \( w.r.t. \) the maximum resource utilization, latency and test error. The maximum resource utilization and normalized latency are computed with the tiling size that (i) does not exhaust of FPGA resources and (ii) minimizes the latency. Therefore, the LightNN-1 can significantly reduce both the FPGA resource usage and latency, while maintaining accuracy.
Chapter 4

FLightNN: Improving LightNN Accuracy and Flexibility

4.1 From LightNN to FLightNN

Compared to the full-precision networks, LightNNs are more storage-efficient in terms of the weight parameters, and also more energy-efficient and have higher throughput for the DNN inference. Also, LightNNs can achieve similar or slightly worse accuracy compared to the full-precision networks on some of the datasets. However, there are still two issues remaining to be addressed. First, while LightNNs fill in the gap between Binarized Neural Networks (BNNs) and the full-precision networks, more design options are still needed since sometimes there is still an accuracy gap between LightNN-1 and LightNN-2. Second, on some datasets and baseline networks, LightNNs can have a big accuracy drop compared to the full-precision models. Thus, an improved training scheme needs to be proposed to achieve higher accuracy. In this chapter, we propose FLightNNs [31] that try to tackle these two issues.

A detailed explanation is shown in Fig. 4.1. LightNNs constrain the weights of DNNs to be a sum of $k$ powers of 2, and therefore can use shift and add operations to replace the multiplications between activations and weights [32]. For LightNN-1\(^1\), all the multiplications of the DNNs will be quantized to the sum of $k$ powers of 2.

\(^{1}\text{LightNN-}k\text{ quantizes weights to the sum of }k\text{ powers of 2.}\)
replaced by a shift operation, while for LightNN-2, two shifts and an add replace the multiplication. Since shift operations are much more lightweight on customized hardware (e.g., FPGA or ASIC), LightNNs can achieve faster speed and lower energy consumption, and generally maintain accuracy for over-parameterized models \[32, 30\]. Although LightNNs provide better energy-efficiency, they lack the flexibility to provide fine-grained trade-offs between energy and accuracy. As shown in Fig. 4.1, the energy efficiency for these models also exhibits gaps, making the Pareto front of accuracy and energy discrete. However, a continuous accuracy and energy/latency trade-off is an important feature for designers to target different market segments (e.g., IoT devices, edge devices, and mobile devices).

To provide a more flexible Pareto front for the LightNN framework, we propose to equip each convolutional filter with the freedom to use a different number of shift-and-add operations to approximate multiplications. Specifically, we introduce a set of free variables \(k = \{k_1, \ldots, k_F\}\) where each element represents the number of shift-and-add for the corresponding convolutional filter. As a result, a more contiguous Pareto front can be achieved. For example, if we constrain \(k \in \{1, 2\}^F\), then the throughput and energy consumption of the new model will sit between LightNN-1 (\(k = \{1\}^F\)) and LightNN-2 (\(k = \{2\}^F\)). Formally, we are solving \(\min_{w,k} \mathcal{L}(w,k)\), where \(\mathcal{L}\) is the loss function and \(w\) is the weights vector. However, the commonly adopted stochastic gradient descent (SGD) algorithm does not apply in this case since \(\mathcal{L}\) is non-differentiable w.r.t. \(k\). In this chapter, we propose a differentiable training algorithm which enables end-to-end optimization with standard SGD. The resulting network is dubbed \(FLightNN\) for its flexible \(k\) values.
In summary, this chapter has the following key contributions:

(i) We propose a differentiable training algorithm for FLightNNs, which provides a continuous Pareto front for hardware designers to search for a highly accurate model under the hardware resource constraints.

(ii) The differentiable training for FLightNNs enables gradual quantization, and further pushes forward the Pareto-optimal curve.

4.2 Differentiable Training for FLightNNs

In this section, we first define the quantization function, and then introduce the end-to-end training algorithm for FLightNNs, equipped with a regularization loss to penalize large \( k \) values.

4.2.1 FLightNN quantization

We first denote the \( i \)-th filter of the network as \( w_i \) and the quantization function for the filter \( w_i \) as \( Q_k(w_i|t) \), where \( k = \max_i k \) is the maximum number of shifts used for this network, and vector \( t \) is a latent variable that controls the approximation (e.g., some threshold value). Also, we denote the residual resulting from the approximation as \( r_{i,k} = w_i - Q_k(w_i|t) \). Then, we formally define the quantization function as follows:

\[
Q_k(w_i|t) = \begin{cases} 
0, & \text{if } k = 0 \\
\sum_{j=0}^{k-1} 1(||r_{i,j}||_2 > t_j)R(r_{i,j}), & \text{if } k \geq 1
\end{cases}
\]

where

\[
R(x) = \text{sign}(x) \times 2^\lfloor \log(|x|) \rfloor
\]

rounds the input variable to a nearest power of 2, and \([.]\) is a rounding-to-integer function. This quantization flow is shown in Fig. 4.2. To interpret the thresholds \( t \), \( t_0 \) determines whether this filter is pruned out, and \( t_1 \) determines whether one shift is enough, etc. Then, the number of shifts for the \( i \)-th filter is \( k_i = \sum_{j=0}^{k-1} 1(||r_{i,j}||_2 > t_j) \). Therefore, choosing \( k_i \) per filter is equivalent to finding optimal thresholds \( t \).

The FLightNN quantization approach targets efficient hardware implementation. Instead of assigning a customized \( k_i \) for each weight, FLightNNs have customized \( k_i \) values per filter, and therefore preserve the structural sparsity. As shown in Fig. 4.3, the convolution with a \( k_i = 2 \)
CHAPTER 4. FLIGHTNN: IMPROVING LIGHTNN ACCURACY AND FLEXIBILITY

\[ r_{i,0} = w_i - Q_0(w_i|t) = w_i \]

\[ r_{i,0} < t_0? \]

Yes

Output: 0

No

\[ Q_0(w_i|t) = 0 \]

\[ r_{i,0} = w_i - Q_0(w_i|t) = w_i \]

\[ |r_{i,0}| < t_0? \]

Yes

Output: 0

No

\[ Q_1(w_i|t) = R(r_{i,0}) \]

\[ r_{i,1} = w_i - Q_1(w_i|t) \]

\[ |r_{i,1}| < t_1? \]

Yes

Output: \( R(r_{i,0}) \)

No

\[ r_{i,0} + R(r_{i,1}) \]

Figure 4.2: Quantization flow for \( k = 2 \).

\[ k_i = 2 \]

\[ k_i = 1 \]

\[ k_i = 1 \]

\[ \begin{array}{ccc}
0.75 & 0.5 & 0.375 \\
0.625 & 0.75 & 0.5 \\
1.25 & 0.625 & 0.25 \\
\end{array} \]

\[ \begin{array}{ccc}
0.5 & 0.25 & 0.25 \\
0.5 & 0.25 & 0.25 \\
1 & 0.5 & 0.125 \\
\end{array} \]

\[ \begin{array}{ccc}
0.125 & 0.25 & 0.125 \\
0.125 & 0.25 & 0.125 \\
0.25 & 0.125 & 0.125 \\
\end{array} \]

Figure 4.3: Equivalent conversion from a convolution with a \( k_i > 1 \) filter to \( k_i \) convolutions each with a \( k_i = 1 \) filter. This transforms the hardware implementation of the FLightNN into LightNN-1.

filter can be equivalently converted to the sum of two convolutions each with a \( k_i = 1 \) filter. Thus, FLightNNs can be efficiently implemented as LightNN-1 with an extra summation of feature maps per layer.

4.2.2 Differentiable training

Instead of picking the thresholds \( t \) by hand, we consider them as trainable parameters. Therefore, the loss function \( \mathcal{L}(w^2, t) \) is a function of both weights and thresholds. Similar to prior work on DNN quantization [151, 25], we use the straight-through estimator (STE) [8] to compute \( \frac{\partial \mathcal{L}}{\partial w_i} \). By defining \( \frac{\partial w_i^q}{\partial w_i} = 1 \) where \( w_i^q = \mathcal{Q}_k(w_i|t) \) is the quantized \( w_i \); therefore, we have \( \frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial w_i^q} \cdot \frac{\partial w_i^q}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial w_i^q} \), which becomes a differentiable expression.

To compute the gradient for thresholds, i.e., \( \frac{\partial w_i^q}{\partial t_j} \), we relax the indicator function \( g(x, t_j) = 1(x > t_j) \) to a sigmoid function [46], \( \sigma(.) \), when computing gradients, i.e., \( \hat{g}(x, t_j) = \sigma(x - t_j) \). In

\( \text{2}\)The bias term is omitted for simplicity.
addition, we use STE to compute the gradient for $R(x)$. Thus, the gradient $\frac{\partial w_i}{\partial t_j}$ can be computed by:

$$\frac{\partial Q_k(w_i|t)}{\partial t_j} = \sum_{l=0}^{k-1} \sigma(||r_{i,l}||_2 - t_l) \frac{\partial R(r_{i,l})}{\partial t_j}$$

where $\frac{\partial ||r_{i,l}||_2}{\partial t_j}$ and $\frac{\partial r_{i,l}}{\partial t_j}$ are 0 for $l < j$; otherwise, they can be computed with the result of $\frac{\partial Q_l(w_i|t)}{\partial t_j}$.

$$\frac{\partial t_l}{\partial t_j} = 1(l = j).$$

### 4.2.3 Regularization

To encourage smaller $k_i$ for the filters, we also add a regularization loss: $L_{reg,k}(w) = \sum_{j=0}^{k-1} \lambda_j \sum_i ||r_{i,j}||_2$ where $\lambda_j$ performs as a handle to balance accuracy and model sparsity. This regularization loss is the sum of several group Lasso losses, since they can introduce structural sparsity [133]. The first item $\lambda_0 \sum_i ||w_i||_2$ is used to prune the whole filters out, while the other items ($j > 0$) regularize the residuals. Fig. 4.4 shows the two items of regularization loss and their sum for the case $k = 2$, with $\lambda_0=1e-5$ and $\lambda_1=3e-5$. Therefore, the total loss for training a FLightNN is:

$$L_{total}(w,t) = L_{CE}(w,t) + L_{reg,k}(w).$$

The new training algorithm is summarized in Algo. 3. This is the same as the conventional backpropagation algorithm for full-precision DNNs, except that in the forward phase, the weights are quantized given the thresholds $t$. Then, due to the differentiability of the quantization function w.r.t. $w$ and $t$, one can compute their gradients and update their values in each training iteration.

### 4.3 FLightNN Accuracy Analysis

The same framework that is used for analyzing LightNN accuracy can be used for FLightNN accuracy analysis as well. The difference between LightNNs and FLightNNs in the training process mainly lies in the number of shift operations for each multiplication. LightNNs have a fixed number of shift operations while FLightNNs have varied numbers of shift operations for different multiplications. Therefore, the expected quantization error for the weights can not be simply
ALGORITHM 3: FLIGHTNN Training Epoch

**Input:** Training dataset \((x, y)\), where \(x\) is input and \(y\) is label; parameters after the \((p-1)\)-th iteration: \(w_{p-1}\) (weights), \(b_{p-1}\) (biases), and quantization thresholds \(t_{p-1}\); quantization function \(Q_k(w|t)\); DNN forward computation function \(g(x, w, b)\); maximum \(k\) value used for all filters; regularization loss coefficients \(\lambda\); learning rate \(\eta\).

**Output:** Updated weights \(w_p\), biases \(b_p\) and thresholds \(t_p\).

```
for each mini-batch of \(x, y\) do
    1. **Quantize weights:** \(w^q = Q_k(w_{p-1}|t_{p-1})\)
    2. **Forward:** compute intermediate results and cross entropy loss function \(L_{CE}\) with \(g(\cdot), w^q, b_{p-1}\), and mini-batch of \(x\); compute regularized loss \(L_{reg,k}\) with \(\lambda\) and \(w_{p-1}\); get the total loss \(L_{total} = L_{CE} + L_{reg,k}\)
    3. **Backward:** compute derivatives \(\frac{\partial L_{total}}{\partial w^q}, \frac{\partial L_{total}}{\partial b_{p-1}}, \frac{\partial L_{total}}{\partial t_{p-1}}\)
    4. **Update parameters:** \(w_p = w_{p-1} - \eta \frac{\partial L_{total}}{\partial w^q}; b_p = b_{p-1} - \eta \frac{\partial L_{total}}{\partial b_{p-1}}; t_p = t_{p-1} - \eta \frac{\partial L_{total}}{\partial t_{p-1}}\)
end```

quantified by equation 3.12 because there is no single \(k\) that describes the number of powers of 2 for all the weights. Thus, we denote the fraction of weights that are a power of 2 at the \(t\)-th iteration as \(\alpha_t\), where \(\alpha_t \in [0, 1]\), and the fraction of weights that are the sum of two powers of 2 is \(1 - \alpha_t\). Note that we only consider FLIGHTNNs that mix LightNN-1 and LightNN-2 for simplicity. Then, the expected quantization error is

\[
\mathbb{E}[||r'||^2|\alpha_t] = \mathbb{E}[||r'||^2|k = 1] \cdot \alpha_t + \mathbb{E}[||r'||^2|k = 2] \cdot (1 - \alpha_t) \\
\leq \left( \frac{\mu^2L^2_1\sigma^2_d}{4} + \frac{\mu^2L^2_1d}{2c+2} \right) \cdot \alpha_t + \left( \frac{\mu^2L^2_1\sigma^2_d}{28} + \frac{\mu^2L^2_1d}{2c+2} \right) \cdot (1 - \alpha_t) \tag{4.1}
\]

\[
= (1 + 6\alpha_t) \cdot \frac{\mu^2L^2_1\sigma^2_d}{28} + \frac{\mu^2L^2_1d}{2c+2}
\]
Therefore, we have

\[ \mathbb{E}[[\|r\|^2]] = \mathbb{E}[\mathbb{E}[[\|r\|^2|\alpha_t]]] \leq (1 + 6\mathbb{E}[\alpha_t]) \cdot \frac{\mu_t^2 L_1^2 \sigma_t^2 d}{28} + \frac{\mu_t^2 L_1^2 d}{2^{2c+2}} \]  \tag{4.2}

Using equation 4.2 and following the proof scheme for Theorem 3.4.5, we can have:

\[ \mathbb{E}[F(\tilde{w}) - F(w^*)] \leq \frac{D^2(\sqrt{T} + 1)}{2T \mu_1} + \frac{\mu_1 \sqrt{T} + 1}{T} \left( \frac{G^2}{2} + \frac{L_1^2 d}{2^{2c+2}} \right) + \frac{1}{T} \sum_{t=1}^{T} \mu_t (1 + 6\mathbb{E}[\alpha_t]) \cdot \frac{L_1^2 \sigma^2 d}{28} \]

\[ = \frac{D^2(\sqrt{T} + 1)}{2T \mu_1} + \frac{\mu_1 \sqrt{T} + 1}{T} \left( \frac{G^2}{2} + \frac{L_1^2 d}{2^{2c+2}} \right) + \frac{\mu_1 L_1^2 \sigma^2 d}{28T} \sum_{t=1}^{T} \frac{1}{\sqrt{t}} (1 + 6\mathbb{E}[\alpha_t]) \]  \tag{4.3}

\( \mathbb{E}[\alpha_t] \) is a function of the iteration \( t \), and is dependent on the regularization strength of the lasso term in the loss function. If the lasso term is very small, then \( \mathbb{E}[\alpha_t] = 0, \forall t \), which indicates that the convergence speed is the same as LightNN-2. On the other hand, if the lasso term is big, making \( \mathbb{E}[\alpha_t] = 1, \forall t \), then the convergence speed is the same as LightNN-1. In both cases, the expected loss difference \( \mathbb{E}[F(\tilde{w}) - F(w^*)] \) follows \( \mathcal{O}\left(\frac{1}{\sqrt{T}}\right) \) convergence speed. For FLightNNs, \( \mathbb{E}[\alpha_t] \) keeps being in the range of \([0, 1]\). Thus, the convergence speed is upper bounded by LightNN-2 and lower bounded by LightNN-1.

While FLightNNs do not change the big-O speed of convergence (\( \mathcal{O}\left(\frac{1}{\sqrt{T}}\right) \)), it is still interesting to analyze the coefficient. Since \( \mathbb{E}[\alpha_t] \) is very difficult to get analytically, we need to make some assumptions for it. Our experiment results show that empirically \( \mathbb{E}[\alpha_t] \) starts from 0, and then gradually increases and saturates at some constant number. For some of the experiments, \( \alpha_t \) saturates at 1 in the end. Therefore, we assume that \( \mathbb{E}[\alpha_t] = 1 - e^{-\gamma t} \) where \( \gamma \) is an coefficient that determines
the speed of $\alpha_t$’s changes. Then, we have

$$\sum_{t=1}^{T} \frac{1}{\sqrt{t}} (1 + 6\mathbb{E}[\alpha_t]) = \sum_{t=1}^{T} \frac{1}{\sqrt{t}} (7 - 6e^{-\gamma})$$

$$\leq 7\sqrt{T+1} - \frac{6}{\sqrt{T}} \sum_{t=1}^{T} e^{-\gamma}$$

$$= 7\sqrt{T+1} - \frac{6}{\sqrt{T}} \cdot \frac{e^{-\gamma(1-e^{-\gamma T})}}{1-e^{-\gamma}}$$

Plugging 4.4 into equation 4.3, we have

$$\mathbb{E}[F(\bar{w}) - F(w^*)] \leq \frac{D^2(\sqrt{T}+1)}{2T\mu_1} + \frac{\mu_1 \sqrt{T}+1}{T} \left( \frac{1}{2} G^2 + \frac{L_1^2 d}{22c+2} \right)$$

$$+ \frac{\mu_1 L_2^2 \sigma^2 d}{28T} (7\sqrt{T+1} - \frac{6}{\sqrt{T}} \cdot \frac{e^{-\gamma(1-e^{-\gamma T})}}{1-e^{-\gamma}})$$

As $T$ increases, the term $7\sqrt{T+1}$ increases but the term $\frac{6}{\sqrt{T}} \cdot \frac{e^{-\gamma(1-e^{-\gamma T})}}{1-e^{-\gamma}}$ decreases, making the second term negligible. Thus, under the assumption that $\mathbb{E}[\alpha_t] = 1 - e^{-\gamma}$, the FLightNN convergence speed is only slightly faster than LightNN-2.

## 4.4 Experimental Results

In this section, we first introduce the experiment setup. Then, we show the accuracy results of different quantized DNN models by software training, as well as their throughput on the FPGA and energy efficiency on the ASIC, to verify the effectiveness of FLightNNs.

### 4.4.1 Setup

We conduct experiments on both small and large CNNs for CIFAR-10, SVHN, CIFAR-100 and ImageNet datasets. The eight adopted network configurations are shown in Table 4.1. To explore the FLightNN performance on different types of network structures, we use a VGG structure with a series of stacked convolutional layers for Network 1, 3, 4 and 5, and adopt the ResNet structure with skip connections across layers for network 2, 6, 7 and 8. Networks 1, 2 and 3 are used for...
experiments on CIFAR-10; networks 4 and 5 are used for SVHN; networks 6 and 7 are used for CIFAR-100; the last one, network 8, is used for ImageNet. For all networks, each convolutional layer is followed by a batch normalization layer and a Leaky ReLU activation function [90], and optionally followed by a max-pooling layer. We use the Adam optimizer [72] to train the network. For each of the networks, we train different quantized models including full-precision DNNs, fixed-point DNNs with 4-bit weights and 8-bit activations, LightNN-2 with 8-bit weights and 8-bit activations, LightNN-1 with 4-bit weights and 8-bit activations, and FLightNNs with 8-bit activations. Due to large training times and limitations in computing resources, we train the ImageNet dataset on a ResNet-10 with reduced width (i.e., network 8), for LightNN-1, LightNN-2 and FLightNNs. For all FLightNNs, we initialize the thresholds $t$ to 0, and set the largest shifts $k$ as 2. For all, except the 32-bit full-precision model, we use 8-bit fixed-point quantization for the activations. By varying $\lambda$, we can have different accuracy-throughput or accuracy-energy trade-offs for FLightNNs. All these networks are trained in software through PyTorch.

Table 4.1: Network settings. “Depth” is the number of convolutional layers in the network. “Width” is the number of convolutional filters of the largest layer.

<table>
<thead>
<tr>
<th>Network ID</th>
<th>Parameters</th>
<th>Structure</th>
<th>Depth</th>
<th>Width</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.08M</td>
<td>VGG</td>
<td>7</td>
<td>64</td>
</tr>
<tr>
<td>2</td>
<td>0.7M</td>
<td>ResNet</td>
<td>18</td>
<td>128</td>
</tr>
<tr>
<td>3</td>
<td>4.6M</td>
<td>VGG</td>
<td>7</td>
<td>512</td>
</tr>
<tr>
<td>4</td>
<td>0.03M</td>
<td>VGG</td>
<td>4</td>
<td>64</td>
</tr>
<tr>
<td>5</td>
<td>0.1M</td>
<td>VGG</td>
<td>4</td>
<td>128</td>
</tr>
<tr>
<td>6</td>
<td>0.7M</td>
<td>ResNet</td>
<td>18</td>
<td>128</td>
</tr>
<tr>
<td>7</td>
<td>2.8M</td>
<td>ResNet</td>
<td>18</td>
<td>256</td>
</tr>
<tr>
<td>8</td>
<td>1.8M</td>
<td>ResNet</td>
<td>10</td>
<td>256</td>
</tr>
</tbody>
</table>

### 4.4.2 Accuracy-throughput trade-off on FPGA

To show the accuracy-throughput trade-off of the models, we implement the inference of each network’s largest convolutional layer for each of the quantized DNN models on FPGA since prior work has shown that convolution operations typically take over 90% of the computation time of a CNN [145]. Our implementation is built on the Xilinx Zynq ZC706 evaluation board. Its working
frequency is 100 MHz. Pre-synthesis is executed on an Intel i7-4790 CPU (3.6GHz) with 16GB RAM. We use Vivado HLS [136] for FPGA implementation. The C code of DNN designs are parallelized by adding HLS-defined pragma and the parallel version is validated with the Vivado HLS timing analysis tool. To make a fair comparison, the same pragma and directives are used for full-precision, fixed-point DNNs, LightNNs and FLightNNs, and we follow the same scheduling settings as prior work [30]. Batched inference is adopted, and the maximum batch size without running out of FPGA resources is set to obtain the highest throughput.

Tables 4.2, 4.3, 4.4 and 4.5 show the accuracy and throughput comparison for full-precision DNNs, fixed-point DNNs, LightNNs and FLightNNs. For all the experimented datasets, LightNNs show the advantage of flexible accuracy-speed trade-offs. In most of the networks (e.g., networks 1, 3, 6 and 7), FLightNNs can achieve an accuracy close to LightNN-2, but have much higher speedup than LightNN-2. Thus, FLightNNs provide continuous trade-offs for accuracy and speed. Compared to the fixed-point quantization, FLightNNs can achieve higher accuracy, and up to 2.0×, 1.8× and 1.8× speedup for CIFAR-10, SVHN, CIFAR-100 datasets, respectively. This is because the multiplication is replaced by shift operators, which require only LUT resources on FPGA while the multipliers require DSP units which are generally more scarce than LUT. Therefore, the computation for FLightNNs allows larger batch sizes than that of fixed-point DNNs, increasing data parallelism, and thus, improving the throughput.

It is also interesting to note that by comparing some FLightNNs (e.g., FL\textsubscript{1a}, FL\textsubscript{2a}, FL\textsubscript{3a}, FL\textsubscript{6a} and FL\textsubscript{7a}) with LightNN-1, we find that FLightNNs can achieve higher accuracy with the same or even lower storage as LightNN-1. This is because initially FLightNNs quantize all the filters with two shifts (since \(t\) is initialized as 0), and gradually add constraints to the filters. This gradual quantization may be better than training a network with only one shift from scratch, as LightNN-1 does. The benefit of gradual quantization has also been observed by prior work [33] which shows that gradually imposing quantization constraints can achieve better accuracy than directly quantizing with a strict constraint.

Table 4.6 shows the FPGA resources utilization for networks 7 and 8. Since full-precision and fixed-point DNNs require DSP for both multiplication and addition, while LightNNs and
FLightNNs only need DSP for addition, full-precision and fixed-point DNNs have larger DSP resource utilization. Compared to full-precision DNNs which use 32-bit floating point operations, fixed-point DNNs only use 4-bit weights and 8-bit activations, and therefore consume fewer DSP units. LightNNs and FLightNNs use LUT to implement the multipliers, and have a higher utilization of LUT than full-precision and fixed-point DNNs. However, the performance of (F)LightNNs is not bounded by LUT resources since the maximum usage of LUT by LightNN-2 is only 42% and 17% for networks 7 and 8, respectively. Instead, the memory resource (BRAM) bounds the performance for (F)LightNNs, while for full-precision and fixed-point DNNs, the performance is bounded by both BRAM and DSP.

Table 4.2: Accuracy and FPGA throughput for CIFAR-10. In the “Model” column, “Full”, “L-2”, “L-1”, “FP”, “FL” indicate full-precision DNN, LightNN-2, LightNN-1, Fixed-point DNN, and FLightNN, respectively. The subscript “xWyA” indicates x bits for weights and y bits for activations. The FLightNN results are shown in bold face. We use subscript a and b to denote the two trained FLightNNs for each network. These notations also apply for Table 4.3, 4.4 and 4.5.

<table>
<thead>
<tr>
<th>ID</th>
<th>Model</th>
<th>Accuracy (%)</th>
<th>Storage (MB)</th>
<th>Throughput (images/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Full</td>
<td>86.36</td>
<td>0.31</td>
<td>3.2e2</td>
<td>1×</td>
</tr>
<tr>
<td></td>
<td>L-2W8A</td>
<td>86.17</td>
<td>0.08</td>
<td>2.2e3</td>
<td>7.0×</td>
</tr>
<tr>
<td></td>
<td>L-1W8A</td>
<td>84.82</td>
<td>0.04</td>
<td>4.5e3</td>
<td>14.4×</td>
</tr>
<tr>
<td></td>
<td>FPW8A</td>
<td>85.09</td>
<td>0.04</td>
<td>3.3e3</td>
<td>10.5×</td>
</tr>
<tr>
<td></td>
<td>FL1a</td>
<td><strong>85.70</strong></td>
<td><strong>0.04</strong></td>
<td><strong>4.8e3</strong></td>
<td><strong>15.0×</strong></td>
</tr>
<tr>
<td></td>
<td>FL1b</td>
<td><strong>85.91</strong></td>
<td><strong>0.06</strong></td>
<td><strong>4.0e3</strong></td>
<td><strong>12.6×</strong></td>
</tr>
<tr>
<td>2</td>
<td>Full</td>
<td>91.70</td>
<td>2.8</td>
<td>1.4e2</td>
<td>1×</td>
</tr>
<tr>
<td></td>
<td>L-2W8A</td>
<td>91.64</td>
<td>0.7</td>
<td>1.6e3</td>
<td>11.5×</td>
</tr>
<tr>
<td></td>
<td>L-1W8A</td>
<td>91.15</td>
<td>0.4</td>
<td>2.7e3</td>
<td>19.0×</td>
</tr>
<tr>
<td></td>
<td>FPW8A</td>
<td>91.17</td>
<td>0.4</td>
<td>1.5e3</td>
<td>10.7×</td>
</tr>
<tr>
<td></td>
<td>FL2a</td>
<td><strong>91.36</strong></td>
<td><strong>0.4</strong></td>
<td><strong>2.8e3</strong></td>
<td><strong>18.9×</strong></td>
</tr>
<tr>
<td></td>
<td>FL2b</td>
<td><strong>91.48</strong></td>
<td><strong>0.7</strong></td>
<td><strong>1.9e3</strong></td>
<td><strong>13.0×</strong></td>
</tr>
<tr>
<td>3</td>
<td>Full</td>
<td>92.85</td>
<td>18.5</td>
<td>1.3</td>
<td>1×</td>
</tr>
<tr>
<td></td>
<td>L-2W8A</td>
<td>92.72</td>
<td>4.6</td>
<td>10.2</td>
<td>7.8×</td>
</tr>
<tr>
<td></td>
<td>L-1W8A</td>
<td>91.93</td>
<td>2.3</td>
<td>39.2</td>
<td>30.2×</td>
</tr>
<tr>
<td></td>
<td>FPW8A</td>
<td>92.23</td>
<td>2.3</td>
<td>19.8</td>
<td>15.2×</td>
</tr>
<tr>
<td></td>
<td>FL3a</td>
<td><strong>92.59</strong></td>
<td><strong>2.3</strong></td>
<td><strong>39.2</strong></td>
<td><strong>30.2×</strong></td>
</tr>
<tr>
<td></td>
<td>FL3b</td>
<td><strong>92.62</strong></td>
<td><strong>3.3</strong></td>
<td><strong>27.2</strong></td>
<td><strong>21.0×</strong></td>
</tr>
</tbody>
</table>
Table 4.3: Accuracy and FPGA throughput for SVHN.

<table>
<thead>
<tr>
<th>ID</th>
<th>Model</th>
<th>Accuracy (%)</th>
<th>Storage (MB)</th>
<th>Throughput (images/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Full</td>
<td>94.96</td>
<td>0.12</td>
<td>2.2e3</td>
<td>1×</td>
</tr>
<tr>
<td></td>
<td>L-2_{8W8A}</td>
<td>94.90</td>
<td>0.03</td>
<td>4.5e3</td>
<td>2.09×</td>
</tr>
<tr>
<td></td>
<td>L-1_{4W8A}</td>
<td>94.16</td>
<td>0.02</td>
<td>8.3e3</td>
<td>3.63×</td>
</tr>
<tr>
<td></td>
<td>FP_{4W8A}</td>
<td>93.70</td>
<td>0.02</td>
<td>3.7e3</td>
<td>1.70×</td>
</tr>
<tr>
<td></td>
<td>FL_{4a}</td>
<td>94.67</td>
<td>0.02</td>
<td>6.7e3</td>
<td>3.11×</td>
</tr>
<tr>
<td></td>
<td>FL_{4b}</td>
<td>94.88</td>
<td>0.03</td>
<td>5.3e3</td>
<td>2.37×</td>
</tr>
<tr>
<td>5</td>
<td>Full</td>
<td>96.44</td>
<td>0.4</td>
<td>1.1e3</td>
<td>1×</td>
</tr>
<tr>
<td></td>
<td>L-2_{8W8A}</td>
<td>96.38</td>
<td>0.1</td>
<td>2.1e3</td>
<td>2.00×</td>
</tr>
<tr>
<td></td>
<td>L-1_{4W8A}</td>
<td>95.93</td>
<td>0.05</td>
<td>3.7e3</td>
<td>3.53×</td>
</tr>
<tr>
<td></td>
<td>FP_{4W8A}</td>
<td>96.02</td>
<td>0.05</td>
<td>1.8e3</td>
<td>1.71×</td>
</tr>
<tr>
<td></td>
<td>FL_{5a}</td>
<td>96.21</td>
<td>0.06</td>
<td>3.2e3</td>
<td>3.06×</td>
</tr>
<tr>
<td></td>
<td>FL_{5b}</td>
<td>96.24</td>
<td>0.08</td>
<td>3.0e3</td>
<td>2.84×</td>
</tr>
</tbody>
</table>

Table 4.4: Accuracy and FPGA throughput for CIFAR-100.

<table>
<thead>
<tr>
<th>ID</th>
<th>Model</th>
<th>Accuracy (%)</th>
<th>Storage (MB)</th>
<th>Throughput (images/s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Full</td>
<td>69.16</td>
<td>2.8</td>
<td>2.5e2</td>
<td>1×</td>
</tr>
<tr>
<td></td>
<td>L-2_{8W8A}</td>
<td>68.84</td>
<td>0.7</td>
<td>1.6e3</td>
<td>6.4×</td>
</tr>
<tr>
<td></td>
<td>L-1_{4W8A}</td>
<td>67.32</td>
<td>0.4</td>
<td>2.7e3</td>
<td>10.6×</td>
</tr>
<tr>
<td></td>
<td>FP_{4W8A}</td>
<td>67.67</td>
<td>0.4</td>
<td>1.5e3</td>
<td>5.98×</td>
</tr>
<tr>
<td></td>
<td>FL_{6a}</td>
<td>68.59</td>
<td>0.4</td>
<td>2.7e3</td>
<td>10.6×</td>
</tr>
<tr>
<td></td>
<td>FL_{6b}</td>
<td>68.76</td>
<td>0.6</td>
<td>1.8e3</td>
<td>6.88×</td>
</tr>
<tr>
<td>7</td>
<td>Full</td>
<td>71.22</td>
<td>11.2</td>
<td>7.4e1</td>
<td>1×</td>
</tr>
<tr>
<td></td>
<td>L-2_{8W8A}</td>
<td>70.96</td>
<td>2.8</td>
<td>6.0e2</td>
<td>8.11×</td>
</tr>
<tr>
<td></td>
<td>L-1_{4W8A}</td>
<td>69.71</td>
<td>1.4</td>
<td>1.1e3</td>
<td>15.2×</td>
</tr>
<tr>
<td></td>
<td>FP_{4W8A}</td>
<td>69.34</td>
<td>1.4</td>
<td>6.9e2</td>
<td>9.26×</td>
</tr>
<tr>
<td></td>
<td>FL_{7a}</td>
<td>70.85</td>
<td>1.4</td>
<td>1.1e3</td>
<td>15.2×</td>
</tr>
<tr>
<td></td>
<td>FL_{7b}</td>
<td>70.87</td>
<td>2.4</td>
<td>7.4e2</td>
<td>9.98×</td>
</tr>
</tbody>
</table>

4.4.3 Accuracy-energy trade-off on ASIC

For all quantized DNNs, we designed pipelined implementations with one stage per neuron, where the computation unit is reused for each neuron. A 65nm commercial standard library is adopted. The Synopsys Design Compiler [91] is used to generate the gate-level netlist of the computation units. The power consumption of all computation operations within one layer is calculated using Synopsys Primetime. We keep all the DNN architectures implemented in an unoptimized fashion
because our main objective is to compare how different quantized DNNs impact computational energy.

The accuracy and computational energy trade-offs for the quantized DNN models are shown in Fig. 4.5. The energy shown in Fig. 4.5 only includes the computational energy consumption for the largest layer of each network. We can clearly observe that FLightNNs provide a more continuous Pareto front for LightNN-2 and LightNN-1, regardless of the network type (i.e., VGG or ResNet), size and the datasets. Similar to the observation in Sec. 4.4.2, in some networks FLightNNs can achieve higher accuracy than LightNN-1 with lower computational energy cost.
Figure 4.5: Accuracy and computational energy consumption in ASIC for different quantized models on CIFAR-10, SVHN, CIFAR-100 and ImageNet datasets. FLightNNs are marked as red triangles, while the other models are shown as blue dots.
Figure 4.6: Accuracy-storage front for LightNN-2 LightNN-1 and FLightNN. The Pareto front of FLightNN is the upper bound of LightNNs.

4.5 Discussion

Pareto fronts for LightNNs and FLightNNs. Since FLightNNs customize the $k_i$ for each filter, LightNN-1 and LightNN-2 can be considered as two special cases for FLightNNs. Therefore, the Pareto front created by the searched FLightNN solutions should be the upper bound for the front of LightNN-1 and LightNN-2 with varied parameter numbers. We test this hypothesis on CIFAR-100 dataset using networks with varied number of convolutional filters. As shown in Fig. 4.6, the accuracy-storage Pareto-front created by FLightNNs is consistently higher than the LightNNs. This indicates that instead of only filling in the Pareto front of LightNNs, FLightNNs can push forward the Pareto front, due to their larger design space. The proposed differentiable training algorithm optimizes both $k_i$ and weight values in an end-to-end fashion, and therefore significantly reduces searching effort compared to exhaustive or heuristic methods with multiple rounds of training. Future work will further improve training efficiency by using optimized training loss [29] or proper labels [16].

FLightNNs for a fixed chip. The training algorithm described in this chapter assumes that one first trains a FLightNN, and then designs a chip for it. Thus, the $k$ value for each filter can be determined while training the model. However, if a chip is already fabricated for a specific FLightNN setting before we train the model, then we need to change our training algorithm to guarantee that the trained model will satisfy the pre-determined $k$ value per filter. For example, the
chip may require the first conv layer to have 32 filters, where ten of them are L1 filters and the others are L2, etc. Note that we only need to guarantee the numbers of L1 and L2 filters to be 10 and 22, respectively, and we still have the flexibility of choosing which filters to be L1 and which to be L2 because the filters can be shuffled without changing the function of the model.

We propose fixed and error-based approaches to determine the $k$ value for each filter while guaranteeing the numbers of L1 and L2 filters meet the requirement. Assume that we need to quantize $m$ filters of a layer to be L1, and $M - m$ filters to be L2. Then, the fixed allocation approach always quantizes the filters with indices $\{0, 1, \ldots, m - 1\}$ to L1 and the others to L2 during the training process. On the other hand, the error-based approach first quantizes all the filters to L1, and then, gives higher precision (i.e., $k = 2$) to $M - m$ filters that have the largest quantization error. The experiment is done on CIFAR-10 dataset. A fully convolutional network is used, with 8 convolutional layers. To study the models with high energy efficiency, an under-parameterized model is used, where the widest layer has 64 filters. Table 4.7 shows the testing accuracy for the fixed and error-based approaches. Interestingly, the simple fixed allocation method can achieve slightly higher accuracy than the error-based method, possibly because the precision of each filter is more stable rather than changing across iterations, thereby making the training process more stable.

Table 4.7: Testing accuracy for fixed and error-based approaches on CIFAR-10 dataset. $\alpha$ is the ratio of L1 filters among all the filters per layer.

<table>
<thead>
<tr>
<th>Model</th>
<th>LightNN-2</th>
<th>$\alpha=0.1$</th>
<th>$\alpha=0.5$</th>
<th>$\alpha=0.9$</th>
<th>LightNN-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy for fixed</td>
<td>85.67%</td>
<td>85.43%</td>
<td>84.78%</td>
<td>84.43%</td>
<td>83.96%</td>
</tr>
<tr>
<td>Accuracy for error-based</td>
<td>85.14%</td>
<td>84.46%</td>
<td>84.39%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary In this chapter, we propose FLightNNs which customize the number of shift operations for each filter of LightNNs. Equipped with the proposed differentiable training algorithm, FLightNNs can achieve a flexible trade-off between accuracy and speed/energy. Our experimental results on FPGA and ASIC simulations show that FLightNNs can provide a more continuous Pareto front for LightNN models and consistently outperform fixed-point DNNs w.r.t. both accuracy

\[^{3}\text{Here we use L1 and L2 filters to indicate the LightNN-1 and LightNN-2 filters, i.e., the filters with } k = 1 \text{ and } k = 2, \text{ respectively.}\]
and speed/energy. Moreover, due to the gradual quantization nature of the differentiable training, FLightNNs can achieve higher accuracy than LightNN-1 without sacrificing speed and energy efficiency, and thus, push forward the Pareto-optimal front. These promising results suggest the potentials for FLightNNs to achieve fast and accurate inference on learning-based customized hardware.
Chapter 5

Improving Accuracy for Binarized Networks via Distribution Loss

In this chapter, we focus on the accuracy improvement for the Binarized Neural Networks (BNNs)\(^1\) with ultra-low power consumption. We first introduce the benefit for using BNNs for hardware implementation and the challenges of training accurate BNNs. Then, we propose a distribution loss that can be added to the loss function of BNNs, to tackle the challenges induced by BNN training, and increase the accuracy of the trained BNN models. The effectiveness of the distribution loss is also demonstrated by the experiment on CIFAR-10, CIFAR-100, SVHN and ImageNet datasets.

5.1 Motivation

In this section, we motivate the study on BNN accuracy improvement by first introducing the low power characteristics of BNNs and then describing their training challenges.

5.1.1 Low-Power BNNs

Recent years have witnessed tremendous success of Deep Neural Networks (DNNs) in various applications of image, video, speech, natural language, etc [50, 89]. However, the increased

\(^1\)The BNN mentioned in this chapter refers to the type of BinaryNet described in Chapter 2.
computation workload and memory access count required by DNNs pose a burden on latency-sensitive applications and energy-limited devices. Since latency and energy consumption are highly related to computation cost and memory access count, there has been a lot of research on reducing these two important design metrics [32, 111, 52, 19]. Binarized Neural Networks (BNNs) [62] that constrain the network weights and activations to be ±1 have been proven highly efficient on custom hardware [150]. We also show later in Sec. 5.2.1 that a typical block of a BNN can be implemented in hardware with merely a few logical operators including XNOR gates, counters and comparators, and therefore greatly reduce the energy consumption and circuit area, as shown in Table 5.1.

In addition to the computational benefit brought by making the whole network binarized, another benefit of BNNs is the huge reduction of memory footprint due to their 1-bit weights and activations. Prior work on extremely low-bit DNNs [25, 37, 81, 11, 127, 30] mainly focuses on few-bit weights and uses more bits for activations, while only a few [62, 83] target 1-bit weights and activations. However, reading and writing intermediate results (activations) generate a larger memory footprint than the weights [98]. For example, in the inference phase of a full-precision (32-bit) AlexNet with batch size 32, 92.7% of the memory footprint is caused by activations, while only 7.3% is caused by weights [98]. Therefore, the memory footprint of BNNs is significantly reduced due to their binary activations.

### 5.1.2 Accuracy Loss of BNNs

While BNNs can achieve high energy efficiency, training accurate BNNs requires careful hyper-parameter selection [2], which makes the process more difficult than for their full-precision counterparts. Prior work has shown that this difficulty arises from the bounded activation function and the gradient approximation of the non-differentiable quantization function [11]. Even for full-precision DNNs, bounded activation functions (e.g., $Sigmoid$ or $Tanh$) usually lead to lower accuracy compared to the unbounded ones (e.g., $ReLU$, leaky $ReLU$, or $SELU$) due to the gradient vanishing problem [41, 16]. For binarized networks, a bounded activation (i.e., $Sign$ function) is used to lead to binary activations, and the $HardTanh$ activation function is commonly used for gradient approximation [62, 105, 127]. As shown in Fig. 5.1, these bounded activation functions bring the following
Figure 5.1: The basic Conv-BN-Act structure for BNN (BinConv: binary convolution; BN: batch normalization). The pre-activation distribution may exhibit from degeneration, saturation or gradient mismatch problem that causes difficulty in training.

Challenges (we use a convolutional layer as an example for illustration purposes): (i) Degeneration: If almost all the pre-activations of a channel have the same sign, then this channel will output nearly constant activations. In an extreme case, this channel degenerates to a constant. (ii) Saturation: If most of the pre-activations of a channel have a larger absolute value than the HardTanh threshold (i.e., |a| ≥ 1), then the gradients for these pre-activations will be zero. (iii) Gradient mismatch: If the absolute values of pre-activations are consistently smaller than the threshold (i.e., |a| < 1), then this is equivalent to using a straight-through estimator (STE) for gradient computation [8]. While the STE generally performs well in computing gradients of staircase functions when training fixed-point DNNs, using STE for computing the gradient of Sign function causes larger approximation error than staircase function, and therefore causes worse gradient mismatch [11].

Due to the difficulty of BNN training, prior work along this track has traded the benefit of extremely-low energy consumption for higher accuracy. Hubara et al. largely increase the number of filters/neurons per convolutional/fully-connected layer [62]. Thus, while a portion of filters/neurons are blocked due to degeneration or gradient saturation, there is still a large absolute number of filters/neurons that can work well. Similarly, Mishra et al. also increase the width of the network to
In addition to increasing the number of network parameters, lots of work sacrifices BNNs’ pure-logical advantage by relaxing the precision constraint. Rastegari et al. approximate a full-precision convolution by using a binary convolution followed by a floating-point element-wise multiplication with a scaling matrix. [105]. Tang et al. use multiple-bit binarization for activations, which requires floating-point operators to compute the mean and residual error of activations [127]. Lin et al. approximate each filter and activation map using a weighted sum of multiple binary tensors [83]. All these approaches use scaling factors for weights and activations, making fixed-point multiplication and addition necessary for hardware implementation. Liu et al. added skip connections with floating point computations to the model [86]. While the models resulting from these approaches use XNOR convolution kernels, the extra multiplications and additions are not negligible. As shown in Table. 5.2, the energy cost for a typical convolutional layer of BNN is lower than the other binarized DNNs. The layer setting and the proposed approach for energy cost estimation are introduced in Appendix 5.2.2. Furthermore, since the hardware implementation of BNNs do not require digital signal processing (DSP) units, they greatly save circuit area and thus, can benefit IoT applications that have stringent area constraint [42, 23].

5.1.3 Chapter Contribution

In this chapter, we propose a general framework for activation regularization to tackle the difficulties encountered during BNN training. While prior work on weight initialization [41] and batch
Table 5.2: Computational energy for a convolutional layer with different types of binarizations.

<table>
<thead>
<tr>
<th></th>
<th>Pure-logical</th>
<th>Energy (µJ)</th>
<th>Relative cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNN [62]</td>
<td>Yes</td>
<td>1.42</td>
<td>1 ×</td>
</tr>
<tr>
<td>XNOR-Net [105]</td>
<td>No</td>
<td>4.34</td>
<td>3 ×</td>
</tr>
<tr>
<td>ABC-Net [83]</td>
<td>No</td>
<td>24.6</td>
<td>17 ×</td>
</tr>
</tbody>
</table>

normalization [65] also regularizes activations, it does not address the challenges mentioned earlier for BNNs, as detailed in Sec. 5.3.2. Instead of regularizing the activation distribution in an implicit fashion as done in prior work [41, 65], we shape the distribution explicitly by embedding the regularization in the loss function. This regularization is shown to effectively alleviate the challenges for BNNs, and consistently increase the accuracy. Specifically, adding the distribution loss can improve the Top-1 accuracy of BNN AlexNet [62] on ImageNet from 36.1% to 41.3%, and improve the binarized wide AlexNet [98] from 48.3% to 53.8%. In summary, this chapter has the following key contributions:

(i) To the best of our knowledge, we are the first to propose a framework for explicit activation regularization for binarized networks that consistently improve the accuracy.

(ii) Empirical results show that the proposed distribution loss is robust to the selection of training hyper-parameters.

5.2 BNN Activation Regularization

In this section, we first show that BNN blocks can be implemented with pure-logical operators in hardware while the other binarized networks (including XNOR-Net and ABC-Net) based on scaling factors require additional full-precision operations. Then, we propose a framework to address the problems of activation and gradient flow incurred in the training process of BNNs. Finally, we discuss the effectiveness of this framework.
Figure 5.2: Basic block for convolutional BNN [62]. The activations $I^l$ and weights $W$ are binarized to ±1. The inference of this block can be implemented on hardware with only logical operators.

### 5.2.1 Difference Types of Binarized DNNs

Binarized DNNs constrain the weights and activations to be ±1, making the convolution between weights and activations use only $\text{xnor}$ and $\text{count}$ operators. In this subsection, we introduce the structure of three typical binarized DNNs, and analyze their hardware implication.

**BNN** As shown in Fig 5.2, the basic block for BNN [62] is composed of a binary convolution, a batch normalization (BN) and an optional max pooling layer, followed by a sign activation function. Without changing the input-output mapping of this block, we can reorder the max pooling layer and the sign function, and then, combine the BN layer and sign function to be a comparator of the convolution results $A^{\text{conv}}$ and input-independent variables $\mu + \frac{\sigma \beta}{\gamma}$, where $\mu$ and $\sigma$ are the moving mean and variance of per-channel activations, which are obtained from training data and fixed in the testing phase; $\beta$ and $\gamma$ are trainable parameters in the BN layer. Therefore, the inference of this BNN block can be implemented in hardware with pure-logical operators. This transformation can also be applied to binarized fully-connected layers followed by BN, pooling and sign function.

**XNOR-Net** Different from BNN, XNOR-Net [105] approximates the activations $A^{bn}$ after the BN layer with their signs and scaling factors computed by the average of the absolute values of these activations, as shown in Fig. 5.3 in Appendix. Since the scaling factors are input-dependent, the
full-precision multiplications and additions cannot be eliminated. The basic blocks of XNOR-Net is shown in Fig. 5.3.

\[ I^F,i \xrightarrow{\text{BN}} A^{bn} \xrightarrow{A^\text{sign} = \text{sign}(A^{bn})} A^{\text{conv}} = W^B \ast A^{\text{conv}} \xrightarrow{A^{\text{cl}} = A^{\text{conv}} \circ K} \xrightarrow{\text{Pooling}} I^{F,i+1} \]

\[ A^\text{avg} = \text{average}(|A^{bn}|) \quad K = A^\text{avg} \ast k \]

Figure 5.3: Basic block for XNOR-Net [105].

**ABC-Net** ABC-Net [83], shown in Fig. 5.4 in Appendix, approximates both weights and activations with a linear combination of pre-defined bases, and therefore making the convolution kernel binarized. However, the approximation prior to binary convolution and the scaling operations after the convolution require full-precision multiplications and additions that cannot be eliminated. The basic blocks of ABC-Net is shown in Fig. 5.4.

\[ A_1^\text{conv} = W^B \ast A_1^B \quad A_1^{\text{cl}} = \alpha \beta_1 A_1^\text{conv} \]

\[ A_N^\text{conv} = W^B \ast A_N^B \quad A_N^{\text{cl}} = \alpha \beta_N A_N^\text{conv} \]

\[ A_1^{\text{cl}} = A_1^\text{cl} \circledast K_a \]

Figure 5.4: Basic block for ABC-Net [83].

### 5.2.2 Energy Cost for Different Types of Binarization

We adopt a convolutional layer from VGG-16 for ImageNet to estimate the computation energy cost in Table 5.2. In the chose layer, both input and output channels are 256; both input and output feature maps are 56x56; the kernels are 3x3 with stride 1. To estimate the computational
energy consumption for BNN [62], XNOR-Net [105] and ABC-Net [83], we first compute the number of XNORs, bit-count operations, fixed-point multiplications and additions for each of the binarization methods, and then add the energy consumption for all the operations together. For BNN and XNOR-Net, the number of XNORs is \( C_{in}C_{out}HWK_hK_w \), where \( C_{in}, C_{out}, H, W, K_h, K_w \) are input channels, output channels, output feature map height and width, and kernel height and width, respectively. For ABC-Net the number of XNORs is \( MNC_{out}HWK_hK_w \) where \( M \) and \( N \) are the number of bases for weights and activations, respectively. In addition to the XNOR operations, BNN also needs counts and comparators, where the number of counts is roughly the same as XNORs, and the number of comparators is \( C_{out}HW \). XNOR-Net also needs \( 2C_{out}HW \) fixed-point multiplications and \( C_{out}HWK_hK_w \) additions, as shown in Fig. 5.3, since the multiplications within the convolution between \( A^{avg} \) and \( k \) can be combined with the following scaling operation. ABC-Net needs approximately \( MNC_{out}HW \) multiplications and additions as shown in Fig. 5.4.

5.2.3 Regularizing Activation Distribution

In this section we first introduce some notations and formally define the difficulties encountered when training BNNs. We denote \( A^{b,l,c} \) as the pre-activations (activations prior to the \( \text{Sign} \) function) for the \( c \)-th channel of the \( l \)-th layer for the \( b \)-th batch of data. Thus, \( A^{b,l,c} \) is a 3D tensor with size \( B \times W \times H \) where \( B \) is the batch size, \( W \) and \( H \) are the width and height of the activation map. From this point on, to avoid clutter, we will omit the superscript of \( A \) whenever possible. \( A_{(q)} \) denotes the \( q \) quantile of \( A \)'s elements where \( 0 \leq q \leq 1 \). We define degeneration, saturation, and gradient mismatch as follows:

\[
\begin{align*}
\text{Degeneration: } & \quad A_{(0)} \geq 0 \text{ or } A_{(1)} \leq 0 \\
\text{Saturation: } & \quad |A|_{(0)} \geq 1 \\
\text{Gradient mismatch: } & \quad |A|_{(1)} \leq 1
\end{align*}
\]  

(5.1)

where \( |A|_{(q)} \) is the \( q \) quantile for \( |A| \), and we use 1 because it is the threshold of \( \text{HardTanh} \) activation shown in Fig. 5.1.

To alleviate the aforementioned problems, we propose to add the distribution loss in the objective function to regularize the activation distribution. Using degeneration as an example, an intuitive way
CHAPTER 5. IMPROVING ACCURACY FOR BINARIZED NETWORKS VIA DISTRIBUTION LOSS

0

Underlying distribution of pre-activations

Penalizing degeneration:

\[ L_D = [(A_0 - 0)_+]^2 + [(0 - A_1)_+]^2 > 0 \]

Easily affected by outliers

\[ L_D = [(A_0 - 0)_+]^2 + [(0 - A_1)_+]^2 = 0 \]

Figure 5.5: Motivation for adjusting regularization. The loss function directly formulated from hypothesis (e.g., degeneration) relies on the minimum (or maximum) of the pre-activations, and therefore is sensitive to outliers.

of formulating a loss to avoid the degeneration problem for \( A \) is

\[ L_D = [(A_0 - 0)_+]^2 + [(0 - A_1)_+]^2, \]

where \(( . )_+\) is the ReLU function. However, this may lead to too loose regularization since a small outlier can make this loss zero, as shown in Fig. 5.5. In addition, this formulation of \( L_D \) is not differentiable w.r.t. the pre-activations \( A \).

Therefore, we propose a three-stage framework consisting of hypothesis formulation, adjusting regularization, and enabling differentiability, to systematically formulate an outlier-robust and differentiable regularization, as shown in Fig. 5.6. First, based on the prior hypothesis about the activation distribution, we can formulate a loss function to penalize the unwanted distribution. Then, if this formulation uses large-variance estimators (e.g., maximum or minimum of samples), we can use relaxed estimators (e.g., quantiles) to increase robustness to outliers. Finally, if the formulated loss is not differentiable, we need to approximate it by assuming the type of parametric distribution (e.g., Gaussian), and approximate the non-differentiable estimators with the distribution parameters.

**Degeneration.** We first formulate the degeneration hypothesis in the loss function as

\[ L_D = [(A_0 - 0)_+]^2 + [(0 - A_1)_+]^2. \]

To make the loss function more robust to outliers, we adjust the regularization by using relaxed quantiles \( \varepsilon \) and \( 1 - \varepsilon \), with

\[ L_D = [(A_{(\varepsilon)} - 0)_+]^2 + [(0 - A_{(1-\varepsilon)})_+]^2. \]

Then, to make \( L_D \) differentiable so that it can fit in the backpropagation training, we first assume a parameterized distribution for the pre-activations \( A \) and then use its parameters to formulate a differentiable \( L_D \).
Based on the heuristics from prior art [81, 11, 101], we assume that the values of $A$ follow a Gaussian distribution $\mathcal{N}(\mu, (\sigma)^2)$, where the $\mu$ and $\sigma$ can be estimated by the sample mean and standard deviation over the 3D tensor. Thus, we can formulate the $\epsilon$ quantile by $\mu - k_\epsilon \sigma$ where $k_\epsilon$ is a constant determined by $\epsilon$. Therefore, $L_D = [(\mu - k_\epsilon \sigma - 0)_+]^2 + [(0 - (\mu + k_\epsilon \sigma))_+]^2 = [(|\mu| - k_\epsilon \sigma)_+]^2$.

**Saturation.** The saturation problem can be penalized by $L_S = [(|A|_0 - 1)_+]^2$, where $|A|_0$ is the minimum value of $|A|$. By adjusting the regularization, we have $L_S = [(|A|_\epsilon - 1)_+]^2$. Since $L_D$ already eliminates the degeneration problem, we find that simply assuming $A$ has a zero mean (i.e., $\mathcal{N}(0, (\sigma)^2)$) works well empirically. Thus, the loss function is formulated as $L_S = [(k_\epsilon \sigma - 1)_+]^2$.

**Gradient mismatch.** When most of the activations lie in the range of [-1,1], the backward pass is simply using a STE for the gradient computation of the sign function, causing the gradient mismatch problem. Therefore, we can formulate the loss as $L_M = [\min(1 - A_1, A_0 + 1)_+]^2$. Similarly, relaxing the regularization leads to $L_M = [\min(1 - A_{1-\epsilon}, A_\epsilon + 1)_+]^2$. With a Gaussian assumption, we have $L_M = [\min(1 - \mu - k_\epsilon \sigma, \mu - k_\epsilon \sigma + 1)_+]^2 = [(1 - |\mu| - k_\epsilon \sigma)_+]^2$.

Then, in the training phase, we add the distribution loss for the $b$-th batch of input data:

\[ L_{DL}^b = \sum_{l,c} L_{DL}^{b,l,c} = \sum_{l,c} L_D^{b,l,c} + L_S^{b,l,c} + L_M^{b,l,c}, \]  

and the total loss for $b$-th batch is:

\[ L_{total}^b = L_{CE}^b + \lambda L_{DL}^b \]  

where $L_{CE}^b$ is the cross-entropy loss, and $\lambda$ is a coefficient to balance the losses.

### 5.2.4 Intuition for the Proposed Distribution Loss

The distribution loss is proposed to alleviate training problems for pure-logical binarized networks. In contrast with full-precision networks, BNNs use a bounded activation function and therefore exhibit the gradient saturation and mismatch problems. By regularizing the activations, the distribution loss maintains the effectiveness of the back-propagation algorithm, and thus, can speedup training and improve the accuracy.
CHAPTER 5. IMPROVING ACCURACY FOR BINARIZED NETWORKS VIA DISTRIBUTION LOSS

Formulating hypothesis

Adjusting regularization

Enabling differentiability

Degeneration

$$L_D = [(A(0) - 0)_+]^2 + [(0 - A(1)_1)_+]^2$$

Saturation

$$L_S = [(|A|_{(0)} - 1)_+]^2$$

Gradient Mismatch

$$L_M = \text{min}(1 - A(1), A(0) + 1)_1^2$$

Figure 5.6: Proposed framework for formulating the differentiable loss function to regularize activation distribution. Starting from the three hypotheses (“degeneration”, “saturation” and “gradient mismatch”), we can formulate the loss function $L_D$, $L_S$ and $L_M$ for them, respectively. We omit the superscript for $A$ and $L$ for better representation.

Since the distribution loss changes the optimization objective, one concern may be that it will lead to a configuration far from the global optimal of the cross-entropy loss function. However, prior theoretical [117, 69, 22] and empirical [63] work has shown that a deep neural network can have many high-quality local optima. Kawaguchi proved that under certain conditions, every local minimum is a global minimum [69]. Through experiments, Im et al. show that using different optimizers, the achieved local optima are very different [63]. These insights show that adding the distribution loss may deviate the training away from the original optimal, but can still lead to a new optimal with high accuracy. Moreover, the distribution loss diagnoses the poor conditions of the activation flow, and therefore may achieve higher accuracy. Our experiment results confirm this hypothesis.
5.3 Experimental Results

In this section, we first evaluate the accuracy improvement by the proposed distribution loss on CIFAR-10, SVHN, CIFAR-100 and ImageNet. Then, we visualize the histograms of the regularized activation distribution. Finally, we analyze the robustness of our approach to the hyper-parameter selection.

5.3.1 Accuracy Improvement

Training configuration. We use fully convolutional VGG-style networks for CIFAR-10 and SVHN, and ResNet for CIFAR-100. All of them use the ADAM optimizer [72] as suggested by Hubara et al. [62]. For the BNN trained with distribution loss (BNN-DL), we compute the loss with the activations prior to each binarized activation function (i.e., Sign function that uses HardTanh for gradient computation). Unless noted otherwise, we set the coefficient $k_e$ to be 1, 0.25 and 0.25 for $L_D$, $L_S$ and $L_M$, respectively, and set $\lambda$ to be 2. To show the statistical significance, all the experiments for CIFAR-10 and SVHN are averaged over five experiments with different parameter initialization seeds. The details of the network structure and training scheme for each dataset is as follows:

**CIFAR-10.** The network structure can be formulated as: $x\text{C-}x\text{C-MP-}2x\text{C-}2x\text{C-MP-}4x\text{C-}4x\text{C-10C-GP}$, where $x\text{C}$ indicates a convolutional layer with $x$ filters, MP and GP indicate max pooling and global pooling layers, respectively. $3 \times 3$ filter size is used for all the convolutional layers. We vary the $x$ to different values ($\{128, 179, 256, 384\}$) to explore the trade-off between accuracy and energy cost, which are shown in Table 5.3 as networks 2-5. We also train a small BNN without the two $4x\text{C}$ layers for CIFAR-10, which is network 1 in Table 5.3. Each convolutional layer has binarized weights and is followed by a batch normalization layer and a sign activation function. The learning rate schedule follows the code from BNN authors [60]. All networks are trained for 200 epochs.

**SVHN.** The network structure for SVHN is the same as CIFAR-10, except that the $x$ is varied from $\{51, 64, 96, 128\}$, shown by networks 6-9 in Table 5.3. The initial learning rate value is 1e-2,
and decays by a factor of 10 at epochs 20, 40 and 45. We train 50 epochs in total.

CIFAR-100. We use the full pre-activation variant of ResNet [51] with 20 layers for CIFAR-100. Prior work has shown the difficulty of training ResNet-based BNNs without scaling layers [140]. Since in ResNet-based BNNs the main path activations and residual path activations do not have matching scales, directly adding the two activations will cause difficulty in training. Therefore, we add two batch normalization layers after these two activations to maintain stable activation scales. Similar to CIFAR-10 and SVHN, we also vary the number of filters per layer. The basic block for ResNet-based BNN is shown in Fig. 5.7. Compared to the full-precision resnet, we add two convolutional layers, BN3 and BN4, to maintain a stable activation flow. For BNN-DL, the distribution loss is still applied to the activations prior to two sign activation functions. To train the ResNet, the initial learning rate is set to 1e-4, and reduced by a factor of 3 every 80 epochs. The networks are trained for 300 epochs.

The network structure for CIFAR-100 is: $x\text{C}-x\text{B}-x\text{B}-2x\text{B}-2x\text{B}-4x\text{B}-4x\text{B}-8x\text{B}-8x\text{B}-\text{GP}-100\text{L}$, where $x\text{C}$ indicates a convolutional layer with $x$ filters, $x\text{B}$ indicates a basic block with $x$ filters for each convolutional layers, GP means global pooling, and $x\text{L}$ means a linear layer with $x$ output neurons. All the convolutional layers use $3 \times 3$ filter sizes. The first convolutional layers within the 2nd, 3rd and 4th blocks use stride 2 to reduce the feature map sizes, while the other convolutional layers use stride 1. We vary $x$ from $\{128, 192, 256\}$.

Results on CIFAR-10, SVHN and CIFAR-100. As shown in Table 5.3, the accuracy for BNN-DL is consistently higher than the baseline BNN. The accuracy gap between BNN and BNN-DL is generally larger than their standard deviations. Using t-test, the p-values for all the network 1-9 are smaller than 0.005, which demonstrates the statistical significance of our improvements. In addition to accuracy results, we also show the computational energy cost for each network, obtained
Table 5.3: Accuracy improvement with distribution loss. Network depth is defined as the number of convolutional layers, while the network width is defined as the number of filters in the largest layer. The best results are shown in bold face. All the accuracy for CIFAR-10 and SVHN is averaged over five experiments with different weight initialization.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Network ID</th>
<th>Depth/width</th>
<th>Params storage</th>
<th>Energy cost (µJ)</th>
<th>Accuracy (mean ± std) (%)</th>
<th>Delta accuracy (%)</th>
<th>BNN</th>
<th>BNN-DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>1</td>
<td>5/256</td>
<td>0.4 MB</td>
<td>0.30</td>
<td>80.61 ± 0.49</td>
<td>2.72</td>
<td>83.33 ± 0.32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>7/512</td>
<td>0.6 MB</td>
<td>0.47</td>
<td>87.54 ± 0.38</td>
<td>1.59</td>
<td>89.13 ± 0.23</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7/716</td>
<td>1.1 MB</td>
<td>0.93</td>
<td>88.99 ± 0.13</td>
<td>1.29</td>
<td>90.28 ± 0.28</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7/1024</td>
<td>2.3 MB</td>
<td>1.89</td>
<td>90.09 ± 0.10</td>
<td>0.92</td>
<td>91.01 ± 0.09</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7/1536</td>
<td>3.8 MB</td>
<td>4.23</td>
<td>90.68 ± 0.11</td>
<td>0.88</td>
<td>91.56 ± 0.16</td>
<td></td>
</tr>
<tr>
<td>SVHN</td>
<td>6</td>
<td>7/204</td>
<td>0.09 MB</td>
<td>0.08</td>
<td>96.23 ± 0.15</td>
<td>0.34</td>
<td>96.57 ± 0.12</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7/256</td>
<td>0.15 MB</td>
<td>0.12</td>
<td>96.53 ± 0.11</td>
<td>0.42</td>
<td>96.95 ± 0.10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7/384</td>
<td>0.3 MB</td>
<td>0.27</td>
<td>97.15 ± 0.15</td>
<td>0.19</td>
<td>97.34 ± 0.05</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>7/512</td>
<td>0.6 MB</td>
<td>0.47</td>
<td>97.34 ± 0.07</td>
<td>0.17</td>
<td>97.51 ± 0.03</td>
<td></td>
</tr>
<tr>
<td>CIFAR-100</td>
<td>10</td>
<td>20/1024</td>
<td>5.6 MB</td>
<td>53.7</td>
<td>60.40</td>
<td>7.77</td>
<td>68.17</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>20/1536</td>
<td>12.6 MB</td>
<td>120.9</td>
<td>64.57</td>
<td>6.96</td>
<td>71.53</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>20/2048</td>
<td>22.3 MB</td>
<td>215.0</td>
<td>66.07</td>
<td>7.35</td>
<td>73.42</td>
<td></td>
</tr>
</tbody>
</table>
by summing up the energy of each operation for the inference of a single input image. Note that this cost excludes the energy of memory accesses, which is the same for BNN and BNN-DL in the inference phase. We also visualize the trade-off between accuracy and energy cost in Fig. 5.8. In most cases, the BNN-DL with a smaller model size can achieve the same or higher accuracy than the BNN with a larger size.

The use of the distribution loss improves the testing accuracy mostly because it regularizes the activation and gradient flow in the training phase, so that the networks can better fit the dataset. As shown in Fig. 5.9, 5.10 and 5.11, the training loss for BNN-DL is consistently lower than the BNN baseline after a few epochs. For most of the experiments, distribution loss is found to converge to a very small number (e.g., 1/10000 of the initial value) in the first few epochs. This indicates that the network can be easily regularized by the distribution loss, which then improves the rest of the training process.

Comparison with prior art. We also compare our results with prior work on binarized networks as shown in Table 5.4. For CIFAR-10 and SVHN, we follow the same network configuration by Hou et al., and also split the dataset into training, validation and testing sets as they do [53]. Table 5.4 shows that by just applying the distribution loss when training BNNs can achieve higher accuracy than the baseline BNN [62], XNOR-Net [105] and LAB [53]. We also show the normalized energy cost for the models. Since XNOR-Net and LAB use scaling factors for the weights and activations, which introduces the need for full-precision operations, XNOR-Net and LAB require 4.5× energy cost than BNN and BNN-DL. We use 16-bit fixed-point multipliers and adders instead of 32-bit floating-point operators to estimate the energy cost of these full-precision operations because prior
CHAPTER 5. IMPROVING ACCURACY FOR BINARIZED NETWORKS VIA DISTRIBUTION LOSS

Figure 5.9: Training loss and testing accuracy curves for different networks with or without distribution loss for CIFAR-10 dataset. The widths of the curves are 2 standard deviation ranges.

Table 5.4: Comparison with prior art using 1-bit weights and activations, in terms of accuracy and computation energy on different datasets. The best results are shown in bold face.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Model</th>
<th>Pure-logical</th>
<th>Energy cost</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>CIFAR-10</td>
<td>BNN [62]</td>
<td>Yes</td>
<td>1×</td>
<td>87.13%</td>
</tr>
<tr>
<td></td>
<td>XNOR-Net [105]</td>
<td>No</td>
<td>4.5×</td>
<td>87.38%</td>
</tr>
<tr>
<td></td>
<td>LAB [53]</td>
<td>No</td>
<td>4.5×</td>
<td>87.72%</td>
</tr>
<tr>
<td></td>
<td>BNN-DL</td>
<td>Yes</td>
<td>1×</td>
<td><strong>89.90%</strong></td>
</tr>
<tr>
<td>SVHN</td>
<td>BNN [62]</td>
<td>Yes</td>
<td>1×</td>
<td>96.50%</td>
</tr>
<tr>
<td></td>
<td>XNOR-Net [105]</td>
<td>No</td>
<td>4.5×</td>
<td>96.57%</td>
</tr>
<tr>
<td></td>
<td>LAB [53]</td>
<td>No</td>
<td>4.5×</td>
<td>96.64%</td>
</tr>
<tr>
<td></td>
<td>BNN-DL</td>
<td>Yes</td>
<td>1×</td>
<td><strong>97.23%</strong></td>
</tr>
<tr>
<td>CIFAR-100</td>
<td>DQ-2bit [102]</td>
<td>No</td>
<td>-</td>
<td>49.32%</td>
</tr>
<tr>
<td></td>
<td>BNN-DL</td>
<td>No</td>
<td>1×</td>
<td><strong>68.17%</strong></td>
</tr>
</tbody>
</table>

Quantization work shows that 16-bit fixed-point operation is generally enough for maintaining accuracy [45]. For CIFAR-100, the closest work that reports 1-bit weights and low-bit activations is by Polino et al. [102], where they use a 7.9MB ResNet with 2-bit activations for CIFAR-100, which presumably has larger energy cost than our 5.6MB model with 1-bit activations, and our results on
CHAPTER 5. IMPROVING ACCURACY FOR BINARIZED NETWORKS VIA DISTRIBUTION LOSS

Figure 5.10: Results for SVHN. The widths of the curves are 2 standard deviation ranges.

accuracy surpass theirs by a large margin.

Results on ImageNet. Having shown the effectiveness of the distribution loss on small datasets, we extend our analysis to a larger image dataset - ImageNet ILSVRC-2012 [109]. We consider AlexNet, which is the most commonly adopted network in prior art on binarized DNNs [62, 105, 151, 127, 98]. We compare our BNN-DL with the baseline BNN [62], XNOR-Net [105], DoReFa-Net [151], Compact Net [127], and WRPN [98]. The BNN uses binarized weights for the whole network [62], while XNOR-Net and DoReFa-Net keep the first convolutional layer and last fully-connected layer with full-precision weights [105, 151]. Compact Net uses full-precision weights for the first layer but binarizes the last layer, and uses 2 bits for the activations [127]. WRPN doubles the filter number of XNOR-Net, and uses full-precision weights for both the first and last layers [98]. Also, BNN uses 64 and 192 filters while the other networks use 96 and 256 filters (or doubling these numbers as WRPN does) for the first two convolutional layers. We train our BNN-DL using the same settings as prior work, except that we use 1-bit activations instead of 2-bit when comparing with Compact Net. The learning rate policy follows prior implementations [60], but starts from 0.01.
Table 5.5: Comparison with prior art on ImageNet with AlexNet-based topology. We use the same model structure as prior work, except that Compact Net uses 2 bits for activations while we only use 1 bit. Training with distribution loss outperforms prior work consistently.

<table>
<thead>
<tr>
<th>Model</th>
<th>Baseline</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Top-1</td>
<td>Top-5</td>
</tr>
<tr>
<td>BNN [62]</td>
<td>36.1%</td>
<td>60.1%</td>
</tr>
<tr>
<td>XNOR-Net [105]</td>
<td>44.2%</td>
<td>69.2%</td>
</tr>
<tr>
<td>DoReFa-Net [151]</td>
<td>43.5%</td>
<td>-</td>
</tr>
<tr>
<td>Compact Net [127]</td>
<td>46.6%</td>
<td>71.1%</td>
</tr>
<tr>
<td>WRPN [98]</td>
<td>48.3%</td>
<td>-</td>
</tr>
</tbody>
</table>

As shown in Table 5.5, BNN-DL consistently outperforms the accuracy of the baseline models. All baseline models except BNN use scaling factors to approximate activations while we keep them binarized. Therefore, our model also has lower energy cost than the prior models. In addition, we highlight that our BNN-DL can outperform Compact Net though we use fewer bits for activations.
5.3.2 Regularized Activation Distribution

To show the regularization effect of the distribution loss, we plot the distribution of the pre-activations for the baseline BNN and for our proposed BNN-DL. More specifically, we conduct inference for network 2 on CIFAR-10, and extract the (floating-point) activations right after the batch normalization layer prior to the binarized activation function of the fourth convolutional layer with 256 filters. Therefore, for each of the 256 output channels, we get its values across the whole dataset. Then, for illustration purposes, we select four channels from baseline BNN and our proposed BNN-DL, respectively, and plot the histogram of these per-channel values, as shown in Fig. 5.12. The four channels’ activation distributions for the baseline BNN are picked to show the degeneration, gradient mismatch, and saturation problems, while the distributions for BNN-DL are randomly selected. From Fig. 5.12a we can see that the good weight initialization strategy [41] and batch normalization [65] adopted for BNNs do not solve the distribution problems.

To show that BNN-DL alleviates these challenges, we compute the standard deviation of activations for each of these 256 channels, as well as their positive ratio, which is the proportion of positive values. As shown in Fig. 5.13, the standard deviation of BNN-DL is more regularized and centralized than that of BNN. The channel with very small standard deviation like the middle two histograms in Fig. 5.12a is rarely seen in BNN-DL, while BNN has a long tail in the area of small standard deviations. This indicates that without explicit regularization, the scale factors of batch normalization layer could shrink to very small values, causing the gradient mismatch problem. From Fig. 5.13, we can also observe that the positive ratio of BNN has more extreme values (i.e., those close to 0 or 1) than BNN-DL. This indicates that the degeneration problem is reduced by distribution loss. Interestingly, we can see that the positive ratio of BNN-DL also deviates away from 0.5. We conjecture that this is because the activations centered at 0 are more prone to gradient mismatch, and thus, be penalized by $L_M$.

We also conduct experiments on ImageNet. A small MobileNet-v1 architecture is adopted which includes 19 convolutional layers and one fully-connected layer. The accuracy for the baseline BNN and the BNN trained with the distribution loss is shown in Table 5.6. Fig. 5.14 provides quantile analysis for the small MobileNet-v1. We extract the activation values for each channel
Table 5.6: Results on a small MobileNet-v1 for the ImageNet dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>Top-1 Accuracy (%)</th>
<th>Top-5 Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline BNN</td>
<td>45.3%</td>
<td>70.4%</td>
</tr>
<tr>
<td>BNN-DL</td>
<td>49.3%</td>
<td>70.6%</td>
</tr>
</tbody>
</table>

Figure 5.12: Activation distribution for BNN trained (a) without or (b) with distribution loss. Each histogram refers to the activations of one channel. In (a), the channel in the left histogram shows a generation problem, the middle two show gradient mismatch, and the right one shows saturation problem. $\sigma$ is standard deviation, and “positive” refers to the ratio of positive activations.

5.3.3 Robustness to Hyper-parameter Selection

Another benefit of the distribution loss is its robustness to the selection of the training hyper-parameters [119]. Prior work [2] has shown that the accuracy of BNNs is sensitive to the training
CHAPTER 5. IMPROVING ACCURACY FOR BINARIZED NETWORKS VIA DISTRIBUTION LOSS

Figure 5.13: Histogram of standard deviation and positive ratio of per-channel activations.

optimizer. We observe the same phenomenon by training BNNs with different optimizers including SGD with momentum, SGD with Nesterov [125], Adam [72] and RMSprop [130]. However, when training BNN with distribution loss, these optimizers can be consistently improved, as shown in Table 5.7. We use CIFAR-10 for the experiments in this subsection. We use the same weight decay and learning rate schedule as Zagoruyko et al. [144] for Momentum and Nesterov, and change the initial learning rate to 1e-4 for RMSprop. We use the same setting as Hubara et al. [60] for Adam. Each model is trained for 200 epochs, and the best testing accuracy is reported. Then,
Table 5.7: Robustness to the selection of optimizer, learning rate, and network structure. CIFAR-10 is used for illustrating the results.

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>BNN</th>
<th>BNN-DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Momentum</td>
<td>66.02%</td>
<td>89.37%</td>
</tr>
<tr>
<td>Nesterov</td>
<td>68.66%</td>
<td>89.22%</td>
</tr>
<tr>
<td>Adam</td>
<td>88.12%</td>
<td>89.62%</td>
</tr>
<tr>
<td>RMSprop</td>
<td>87.39%</td>
<td>90.24%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Learning rate</th>
<th>BNN</th>
<th>BNN-DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(lr_{init}=1e-1)</td>
<td>82.19%</td>
<td>89.60%</td>
</tr>
<tr>
<td>(lr_{init}=5e-3)</td>
<td>88.12%</td>
<td>89.62%</td>
</tr>
<tr>
<td>(lr_{init}=2e-4)</td>
<td>85.62%</td>
<td>88.73%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Network</th>
<th>BNN</th>
<th>BNN-DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG</td>
<td>88.12%</td>
<td>89.62%</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>85.71%</td>
<td>90.47%</td>
</tr>
</tbody>
</table>

We vary the learning rate schedule of Hubara et al.’s implementation [60] by scaling the learning rate at each epoch by a constant. Table 5.7 shows that BNN-DL is more robust to the selection of learning rate values. Furthermore, we show that the BNN-DL can work well for both VGG-style networks with stacked convolutional layers and ResNet-18 which includes skip connections. The VGG-style network uses the network 2 in Table 5.3. The ResNet-18 structure uses the pre-activation variant [51] with added batch normalization layers as described in Sec. 5.3.1. Since BNN has non-regularized activations, maintaining the activation flow in the training process requires more careful picking of hyper-parameter values. However, the distribution loss applies regularization to the activations, making the network easier to train, and therefore reduces the sensitivity to hyper-parameter selection.

We also show that the distribution loss is robust to the selection of the introduced hyper-parameter, \(\lambda\) coefficient, which indicates the regularization level of the distribution loss. As shown in Table 5.8, by varying \(\lambda\) from 0.2 to 2000, the accuracy for BNN-DL is consistently higher than the baseline BNN. As mentioned in Sec. 5.3.1, the distribution loss quickly decays to a small magnitude in the first few epochs, and we find that this holds for a wide range of \(\lambda\). The robustness analysis indicates that the distribution loss is a handy tool to regularize activations, without the need of much hyper-parameter tuning.
Table 5.8: Accuracy for BNN-DL on CIFAR-10 with varied regularization levels. $\lambda = 0$ indicates the baseline BNN.

<table>
<thead>
<tr>
<th>$\lambda$</th>
<th>0</th>
<th>0.2</th>
<th>2</th>
<th>20</th>
<th>200</th>
<th>2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acc. (%)</td>
<td>87.39</td>
<td>90.12</td>
<td>90.16</td>
<td>90.18</td>
<td>90.61</td>
<td>90.20</td>
</tr>
</tbody>
</table>

5.4 Discussion

In this chapter, we tackle the difficulty of training BNNs with 1-bit weights and 1-bit activations. The difficulty arises from the unregularized activation flow that may cause degeneration, saturation and gradient mismatch problems. We propose a framework to embed this insight into the loss function by formulating our hypothesis, adjusting regularization and enabling differentiability, and thus, explicitly penalizing the activation distributions that may lead to the training problems. Our experiments show that BNNs trained with the proposed distribution loss have regularized activation distribution, and consistently outperform the baseline BNNs. The proposed approach can significantly improve the accuracy of the state-of-the-art networks using 1-bit weights and activations for AlexNet on ImageNet dataset. In addition, this approach is robust to the selection of training hyper-parameters including learning rate and optimizer. These results show that distribution loss can generally benefit the training of binarized networks which enable latency and energy efficient inference on mobile devices.
Chapter 6

Related Work

6.1 Efficient DNN Inference

In addition to DNN quantization introduced in Chapter 2, there are many other approaches that aim for increasing the inference speed and energy efficiency for DNNs. Some of them change the DNN model itself, while some focus on the hardware implementation.

6.1.1 Algorithmic Improvement

Some prior work focuses on the algorithmic side to improve the DNN inference speed. In this section, we introduce network pruning, neural architecture search, data transformation, and neural block design, which can reduce both the computation and the memory footprint.

Network pruning  Pruning a DNN refers to reducing the number of convolutional filters for CNNs, or reducing the number of neurons for the fully-connected layers of Multi-Layer Perceptrons (MLPs) or CNNs. Because network pruning can reduce both FLOPs and memory footprint of a DNN, and can easily fit into the current CPU or GPU framework, there has been lots of work on neural network pruning [141, 87, 133, 80, 20]. The key question in network pruning is to determine which neurons or convolutional filters should be pruned out, to meet a resource constraint. Ye et al. use a stochastic training method to force some of the channels to be constant values, and then
prune out these channels and adjust the bias terms to recover accuracy [141]. Liu et al. first train a network with an inflated number of channels, and prune the unimportant convolutional filters from the large network by imposing regularization to the scaling factors [87]. Wen et al. add grouped lasso terms to the loss function to enable structural sparsity of the trained CNNs [133]. Li et al. prune the convolutional filters with the smallest $l_1$ norms [80]. Chin et al. propose a way to globally rank the convolutional filters across all layers of a CNN, and prune the bottom-ranked filters [20]. The evolutionary algorithm or reinforcement learning can be used for learning this global ranking.

**Neural architecture search** Neural Architecture Search (NAS) refers to the process of searching for a neural network architecture that has high accuracy and meets the resource constraint (e.g., FLOPs, latency, etc.). Thus, NAS is a multi-objective optimization problem. Prior work on NAS varies due to the different approaches to parameterize the network architecture, determine the objective function, and solve the optimization problem [126, 106, 68, 120]. Tan et al. set the objective function to be the model accuracy multiplied by a power of the normalized latency [126]. They use a reinforcement learning approach to search for a network architecture that maximizes the objective function. Real et al. define the normal cells and reduction cells and use evolutionary algorithm to search for the cell architectures that maximize the accuracy [106]. Kandasamy et al. use a Gaussian process based Bayesian optimization method to search for the most accurate architecture [68]. Stamoulis et al. formulate the NAS in a differentiable way and use an end-to-end training method to quickly search for an accurate and lightweight architecture [120].

**Data transformation** Some work focuses on the reduction of computation and memory footprint by conducting transformations for the input images [19, 44]. Chin et al. propose AdaScale that adaptively changes the scale of the input images to improve the accuracy of the object detection application [19]. Since the images will be downsampled for most of the time and the inference latency is roughly proportional to the square of the input image size, the expected inference speed is improved. Gueguen et al. train the DNNs that directly use the JPEG-format images to be the input, instead of decompressing the JPEG images first [44]. Also, because the first layer of a regular
DNN maps the image from spatial domain to frequency domain, their new DNN directly receives the frequency information and thus, can reduce the number of layers of the architecture.

**Neural block design** Some other work designs the neural block that is more lightweight than the conventional ones [55, 111, 54, 149, 18]. Different from NAS, research in this thread focuses on employing human intuitions for the neural block design rather than using automated search. Howard *et al.* propose MobileNet, which uses a depthwise conv layer followed by a point-wise conv layer to replace the conventional conv layer [55]. By decomposing the feature extraction and the feature combination functionalities, the new neural block reduces FLOPs and thereby can increase the inference speed. The ShuffleNet proposed by Zhang *et al.* utilizes a point-wise group convolution and a channel shuffle layer to reduce the FLOPs [149]. Sandler *et al.* propose MobileNet-v2 that adds a point-wise convolutional layer at the beginning of each MobileNet block, and adds a residual connection to the block, to improve the accuracy of MobileNet while preserving the low-latency benefit [111]. Then, Howard *et al.* combine NAS and human-designed neural blocks, and further push the accuracy-latency Pareto front by MobileNet-v3 [54]. Chen *et al.* propose a virtual pooling layer that can reduce the convolution operations by interpolating values of a feature map [18]

In addition to the improvement of the general-purpose network backbone, there has also been work that targets on specific applications and designs lightweight heads of the DNNs. In the regime of object detection, there is a thread of networks that keeps pushing the Pareto front of the detection results [40, 39, 108, 85, 107, 82]. Girshick *et al.* propose the first DNN-based object detector, R-CNN, that extracts lots of proposal regions from the input image and applies a CNN classifier to each of the region [40]. Then, Girshick proposes fast R-CNN to improve the speed of R-CNN by several network modifications and training techniques [39]. After that, Ren *et al.* proposes faster R-CNN, which feeds the whole image into the CNN, and generates proposal regions for the high-level feature maps [108]. A head is applied to the high-level features, to classify the objects in each region. Faster R-CNN further improves the speed of inferring an image because of the shared back-bone computation for different proposals. In addition, the Single Shot MultiBox Detector
(SSD) proposed by Liu et al. [85], the YOLO network proposed by Redmon et al. [107], and the RetinaNet proposed by Lin et al. [82] improve the object detection speed from different aspects.

6.1.2 Hardware Implementation

Some prior work focuses on the hardware implementation of the DNNs, and tries to reduce the area and energy cost by designing better hardware architectures for specific deep learning models. In this section, we introduce the Diannao family [12, 15, 34, 84], Google’s TPUs [67, 57], and Eyeriss [13, 14]. In addition to these, IBM’s TrueNorth chip [94] and Microsoft’s Brainwave platform [38] are also examples of DNN accelerators.

**Diannao family** The Diannao family is a series of DNN accelerators aiming for smaller area, higher throughput and lower energy consumption. The first architecture in this thread is Diannao, proposed by Chen et al. [12], which tackles the big memory footprint of DNNs by exploiting the locality properties of their layers. Chen et al. decompose the computation of a DNN into basic linear algebra units, and design custom hardware for the computation of one neuron, which includes the multiplication between weights and activations, the addition of the multiplication results, and the sigmoid function. Then, Chen et al. propose Dadiannao [15] that scales up the supported DNN size by interconnecting multiple chips that share the same large RAM which is sufficient for containing the whole DNN. Thus, the main memory is not needed, which solves the problem of high bandwidth requirement. Du et al. propose another approach to tackling the bandwidth problem [84]. Their proposed ShiDianNao directly connects the accelerator and the vision sensors, and improves the energy efficiency due to the absence of DRAM accesses and a careful exploitation of the data path patterns. After that, Liu et al. extend the approaches to more general machine learning algorithms and design PuDianNao that uses vector inner product for the basic computation unit and can serve for k-means, k-nearest neighbors, naive Bayes, Support Vector Machine, linear regression, and neural networks [84].
TPU  Jouppi et al. from Google design the Tensor Processing Units (TPUs) and deploy them in the data centers to accelerate the DNN inference required by speech, vision, language, translation, search ranking services [67]. Matrix computation units are used for the basic block for TPU, and the TPU microarchitecture is designed to maximize the utilization of the matrix units. Systolic execution is utilized to reduce reads and writes of the buffer because reading a large SRAM uses much more power than arithmetic. They use 8-bit fixed point to replace the 32-bit floating point representation to save area and energy cost. In addition, they developed the TPU software stack that is compatible with those developed for CPUs and GPUs, and thus, makes the TPU easier to use from the perspective of algorithm engineers.

To deploy the TPUs in edge devices, Google has also developed the Edge TPU that supports energy-efficient DNN inferences [57]. Rather than supporting various machine learning frameworks including Tensorflow, scikit-learn, XGBoost and Keras as the Cloud TPU does, Edge TPUs only support the TensorFlow Lite (a lightweight version of Tensorflow) and NN APIs. While the flexibility is reduced, Edge TPUs can save much power compared to the Cloud TPUs deployed in the data centers.

Eyeriss  Different from TPUs that are mainly designed for Multi-Layer Perceptrons (MLP) and Recurrent Neural Networks (RNNs), the Eyeriss designed by Chen et al. focuses on CNNs [13]. The authors present a Row Stationary dataflow, that reuses the convolutional filter weights and activations, and minimizes the data movement. This is achieved by having each Processing Engine (PE) to compute a 1D convolution, and properly passing the rows of the convolutional filter and activation maps between the PEs. Thus, the data movement between the PE arrays and the global buffer is reduced. This mechanism can also be extended to the fully connected layers and the pooling layers by rearranging the tensor shapes and replacing the MAC computation by a MAX operation, respectively. Then, Chen et al. develop Eyeriss v2 to support the pruned DNNs that have irregular shapes and the compressed DNNs that have sparse weights [14].
6.2 Efficient DNN Training

There has also been recent work that trades the parameter precision for the training efficiency (e.g., speed, energy efficiency) [134, 95, 26, 132]. In general, when training a DNN, the parameters that need to be quantized for reducing energy cost are the weights, activations, and gradients [134]. The parameter precision required by DNN training is also generally higher than DNN inference because of two reasons. First, lots of work shows that the ranking of precision requirements for the parameters is: gradients, activations, weights (from high to low). Thus, DNN inference does not involve gradient quantization, and can maintain accuracy with lower precision. Second, at the backward phase of each training iteration, the gradients are computed using the activations produced by the forward phase. Therefore, the precision loss of the activations can accumulate to the gradient computation.

Due to the different precision requirement for weights, activations and gradients, prior work has adopted mixed precision for these parameters. Wu et al. propose to use different number of bits for weights, activations, the gradients of the loss function \( \text{w.r.t.} \) weights, and the gradients of the loss function \( \text{w.r.t.} \) activations [134]. Micikevicius et al. from Nvidia use half precision for the computation in the forward and the backward phase, but use single precision for updating the gradients [95]. To increase the accuracy for the half-precision computation, they still use single precision for the accumulation operation in the convolutions and the matrix multiplications. Das et al. from Intel propose to use a shared exponent representation for tensors, which addresses the distribution distinction of the weights, activations and gradients [26]. In addition, they use integer Fused-Multiply-and-Accumulate operations where the two operands are INT16 operands and the output is INT32. The three works all use fixed-point representations for the DNN training, while Wang et al. from IBM propose to train DNNs with low-precision floating point numbers [132]. They customize the precision in the operation level. The operations of the multiplications between weights and activations are using the 8-bit floating point representation, while the accumulation of the multiplication results uses 16-bit floating point additions. The gradients used for weight updates are also 16-bit floating point numbers. The stochastic rounding for the weight updates is used to reduce accuracy loss. In addition to these algorithmic advances, the Cloud TPU by Google
also supports efficient training by using the Brain Floating Point (BFloat16) representation [56]. In addition to increasing training efficiency by quantization, Zhuo et al. reduce the training data amount required for a high accuracy by using fine-grain labels [16].

### 6.3 BNN Improvements

Chapter 5 introduces a distribution loss to regularize the activation distribution for BNNs, and therefore can improve the BNN accuracy. In this section, we first introduce prior work that regularizes the activation distributions for full-precision DNNs, and prior work on the accuracy improvement for BNNs. Then, prior work on the hardware implementation for BNNs is introduced.

#### 6.3.1 Activation Flow Regularization

Prior work has proposed various approaches to regularize the activation flow of full-precision DNNs, mainly to address the gradient vanishing or exploding problem. Ioffe et al. propose batch normalization to centralize the activation distribution, accelerate training, and achieve higher accuracy [65]. More specifically, the activations generated by convolutional and fully-connected layers are first centralized at each channel, and go through a linear transformation where the weight and bias are trainable and initialized as 1 and 0, respectively. The batch normalization layer is equivalent to a linear layer and can be fused to the convolutional or fully-connected layer ahead of it without changing the function, but the initial values of the weight and bias help the activation distribution to start from a near-Gaussian state. Similarly, Huang et al. normalize the weights with zero mean and unit norm followed by scaling factors [58]. Shang et al. extend the normalization idea to residual networks using normalized propagation [113], while Ba et al. and Salimans et al. normalize the activations of Recurrent Neural Network (RNN) by layer-wise normalization and weight reparameterization, respectively [6, 110]. In addition, some prior work develops good initialization strategy to regularize the activations in the initial state [96, 135]. By assuming the output activations of each layers to be standard Gaussian, one can compute the weight variance required to maintain the non-exploding and non-diminishing Gaussian activation distribution. In
addition, some work proposes new activation functions to maintain stable activation distribution across layers \[73, 90\]. While these approaches have been shown effective on full-precision DNN training, they do not address the difficulty of training a BNN due to its bounded activation functions and its Bernoulli-distributed weights.

### 6.3.2 Accuracy Improvement for BNNs

Prior work on binarized DNNs alleviates the problem of activation distribution mainly by approximating the full-precision activations with multiple-bit representations and floating-point scaling factors \[127, 11, 98, 102, 97, 37, 83, 53\]. Tang et al. introduce scaling layers that relax the activation values to floating point numbers rather than only 1 or -1 \[127\], and they use 2 bits for activations which allow for 4 quantization levels. Cai et al. also use multi-level activation function for inference and variants of ReLU for gradient computation to reduce gradient mismatch \[11\]. Polino et al. leverage knowledge distillation to guide training and improve the accuracy with multiple bits for activations \[102\]. Lin et al. approximate both weights and activations with multiple binary bases associated with floating-point coefficients \[83\], so the convolutional layer can be implemented by a few binary convolutions followed by a weighted average operation where the weights are floating point numbers. While these approaches improve the accuracy for binarized networks, they sacrifice the energy efficiency due to the increased bits and the required DSP units for the floating point additions and multiplications.

### 6.3.3 BNN Hardware Implementation

Due to BNNs’ potentials of high throughput and energy efficiency, there has been lots of work on BNN hardware implementation \[7\]. Yonekawa et al. implement BNNs on FPGAs, and store all the binarized weights and activations in the BRAM \[142\]. They eliminate the floating point operations of batch normalization layers via the method described in Sec. 5.2.1, Chapter 5. Similarly, some other work also designs BNNs using the in-memory computation \[123, 21, 4\], due to the small memory footprint required by BNNs. In addition, recent research has focused on mixing the digital and analog technologies applied on BNNs. Due to the small area and large storage of resistive
random access memory (RRAM), Yu et al. built custom hardware for BNNs using the binary RRAM devices [143]. Similarly, Sun et al. propose a RRAM synaptic architecture and implement the XNOR and bit-counting operations in a parallel fashion [124]. Tang et al. design a RRAM accelerator for the BNN inference, and propose approaches to address the matrix splitting problem caused by a large layer size. Zhou et al. use the non-linear synaptic cells to train BNNs [152]. They use RRAM and design a 2T2R-based synaptic cell to realize the functions of reading binary weights and updating the weights with the gradients. Bankman et al. build a mixed-signal BNN processor for always-on inference applications [7]. To address the energy bottleneck caused by wide vector summation, they design an energy-efficient switched-capacitor neuron and employ a large capacitive digital-to-analog converter section for summing up the multiplication results.
Chapter 7

Conclusion

Deep learning models have been adopted by many vision, speech and language applications, but most of the models are deployed on the cloud due to the energy constraint of the edge devices. In this thesis, we aim to build the deep learning models on the edge devices because of several benefits of on-edge inference. First, the elimination of transferring users’ speech, text or image data to the cloud can protect user privacy. Second, on-edge model inference can be done without an internet connection. Third, the transmission delay is eliminated, and thus can potentially reduce the latency of the service. However, the deep learning models that are highly accurate for these applications generally have many layers and lots of neurons or convolutional filters in each layer, and therefore need powerful computation units and large memory to enable fast inference. Since edge devices usually are small in area and use batteries that provide limited energy, one can hardly use powerful hardware (e.g., Nivida TitanX GPUs) for edge devices.

In this thesis, we try to tackle this challenge by using the DNN quantization approach that can reduce the requirement for hardware resources. We propose LightNNs, which constrain the weight values of DNNs to be a power of 2 or the sum of two powers of 2. LightNNs modify the computation logic of conventional DNNs by making reasonable approximations, and replace the multipliers with more energy-efficient operators involving only one shift or limited shift-and-add operations. In addition, LightNNs also reduce weight storage, thereby decreasing the memory access energy. Rather than directly quantizing the trained full-precision model, we also propose an
CHAPTER 7. CONCLUSION

associated training algorithm for LightNNs to compensate for the quantization error while training. Experiment results on various image classification datasets show that LightNNs can achieve similar accuracy as the full-precision model in the over-parameterized cases, and achieve consistently higher accuracy than BNNs. Experiment results of gate-level hardware simulation show that LightNNs fill the gap between conventional DNNs and BNNs in terms of accuracy, storage, energy and area, and provide more options for hardware designers to select DNN architectures based on their accuracy and resource constraints. We also show that the FPGA implementation for LightNNs requires fewer resources and increases the throughput compared to the conventional DNNs.

To increase the flexibility of LightNNs, we propose FLightNNs which customize the number of shift operations for each convolutional filter of LightNNs. Instead of using a search-based method to determine this shift number for each filter, we propose an approach to formulate the selection of shift numbers to be a differentiable function, and thus, enable an end-to-end training using the back-propagation algorithm. In addition, the proposed training scheme starts from a high-precision and gradually increases the network sparsity by using a grouped lasso loss function. This gradual quantization helps the accuracy of the trained FLightNN to be higher than a LightNN at the same sparsity level. We conduct experiments on vision datasets including CIFAR-10, SVHN, CIFAR-100 and ImageNet, and our FPGA and ASIC simulations show that FLightNNs can provide a more continuous Pareto front for LightNN models and consistently outperform fixed-point DNNs w.r.t. both accuracy and speed/energy. In addition, by varying the number of convolutional filters per layer, experimental results show that FLightNNs can push forward the accuracy-storage Pareto front of LightNNs. These promising results suggest the potentials for FLightNNs to achieve fast and accurate inference on learning-based customized hardware.

Compared to LightNNs and FLightNNs, Binarized Neural Networks can reduce even more energy consumption and improve the inference speed, because BNNs quantize the weights and activations to be either +1 or -1. Thus, the multiplications between the weights and activations can be replaced by XNOR operations. Memory accesses are also reduced because of the 1-bit weights and activations. However, BNNs have big accuracy loss due to the aggressively reduced precision and several training-induced issues. More specifically, training BNNs is difficult since the
activation flow encounters degeneration, saturation, and gradient mismatch problems. Instead of increasing more bits for activations or adding scaling layers which may sacrifice BNNs’ benefits of pure-logical inference, we propose to use distribution loss to explicitly regularize the activation flow, and develop a framework to systematically formulate the loss. Our experiments on CIFAR-10, SVHN, CIFAR-100 and ImageNet datasets show that the distribution loss can consistently improve the accuracy of BNNs without losing their energy benefits. Moreover, equipped with the proposed regularization, BNN training is shown to be robust to the selection of hyper-parameters including optimizer and learning rate.

Future work in hardware-aware machine learning can explore efficient training and application-oriented model compression. Since user privacy has been more and more important, on-edge training and federated learning will be of interest for machine learning practitioners. On-edge training directly trains the model on the edge devices using the user data, making both the accuracy and energy efficiency important for the deployment of the model. Federated learning still trains the model in the cloud, but instead of transferring user data to the cloud, it transfers the gradients of the weights obtained by training a few steps using a user’s data. Most prior work on model compression targets on image classification, and thus, just needs to compress the DNN backbone. Due to the fast emergence of 5G technology and Internet of Things (IoTs), many interesting applications that use machine learning approaches will be built on edge devices. Application-oriented model compression will be useful for designing efficient models that are tailored for the target application and the target device.
Bibliography


[57] https://cloud.google.com/edge-tpu/. Google’s purpose-built asic designed to run inference at the edge. 88, 89


[59] Itay Hubara, 2017. 28


[61] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio, 2016. 24


[64] Synopsys inc, 2016. 30


[100] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. 2011. 10


