UNIVERSITY OF CALIFORNIA SAN DIEGO

Resource Efficient and Error Resilient Neural Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Jeng-Hau Lin

Committee in charge:

Professor Rajesh K. Gupta, Co-Chair
Professor Zhuowen Tu, Co-Chair
Professor Gert Cauwenberghs
Professor Garrison W. Cottrell
Professor Julian J. McAuley

2019

The Dissertation of Jeng-Hau Lin is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California San Diego

2019

DEDICATION

To my dearest family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to express my heartfelt thanks to my Ph.D. advisor Professor Rajesh K. Gupta, who generously provided me a position in his research group and discerningly pointed a bright direction for me when I was in a quandary. Were it not for his organization and support, I cannot devote myself to the exciting research projects included in this dissertation. Further, his modesty and positive thinking habits have affected me beyond my study and research to my daily life and equip me with confidence and a lenient attitude to embrace and tackle the unseen challenges or setbacks in my career.

I would also like to dedicate my gratitude to my erudite and creative co-advisor, Professor Zhuowen Tu, who kept shedding light on my research problems uninterruptedly. My horizon in computer vision and supervised learning has been broadened because of his prudential guide and teaching. In the LBPNet research, his profound knowledge helped me more than once to solve algorithmic bottlenecks and dare to build the unprecedentedly simple yet effective idea for hardware.

I thank my committee members, Professor Gert Cauwenberghs, Professor Garrison W. Cottrell, and Professor Julian J. McAuley for their valuable advice and comments in my dissertation. I am also grateful to Professor Lawrence K. Saul, Professor Ryan Kastner, and Professor Tajana imuni Rosing for the suggestions in my research exam.

My gratitude goes to my colleagues from both Prof. Gupta's MESL and from Prof. Tu's mlPC Lab, Xun Jiao, Atieh Lotfi, Dezhi Hong, Vahideh Akhlaghi, Mulong Luo, Zhou Fang, Jason Koh, Omid Assare, Francesco Fraternali, Dhiman Sengupta, Yunfan Yang, Weijian Xu, Justin Lazarow, Yifan Xu, Long Jin, Saining Xie and Zeyu Chen for the discussing and helping me on the implementing of GPU programming and FPGA syntheses. I would like to thank my friends Hsin-Ming Lin, Sung-En Chiu, Yan-Tsung Peng, Chung-Lun Hsu, Chang-Heng Wang, Wei-Ting Chan, Hung-Wei Tseng, and many more for both of the spiritual and physical supports that have enriched my Ph.D. years. My thanks for the supports in my first two years in VLSI Lab to Professor Patrick Mercier, Professor Chung-Kuan Cheng, Hao Zhuang, Jingwei Lu, Ilgweon

Kang, Hao Liu, Chia-Hung Liu, Gung-Yu Pan, Xinan Wang, Xiang Zhang, and Yu-Te Wang. Special thanks to Professor Sorin Lerner, and Julie Conner for the advice during my transferring.

Finally, I thank my parents, Li-Ching Chung and Lien-Fa Lin, parents in law, Po-Fen Yen and Ta-Wei Chih, cousins in law, Jordan Jih and Jonathan Jih, my elder sister and her husband, Yi-Hsin Lin and Hung-Chih Wang, and all other family members for always being encouraging and supportive throughout my life. I owe my sincere gratitude to my wife, Pei-Chain Chih, for her patient companion, delicious dishes, euphonic piano performance, and the most beautiful smile in the world every day.

The material in this dissertation is based on the following publications.

Chapter 1 includes part of the results of Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, Zhiru Zhang. "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs". *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)* as the motivation of the following research in chapter 2 and chapter 3. This dissertation author is the co-author of this paper.

Chapter 2 contains reprints of Jeng-Hau Lin, Tianwei Xing, Ritchie Zhao, Zhiru Zhang, Mani Srivastava, Zhuowen Tu, and Rajesh K. Gupta."Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration." *In Computer Vision and Pattern Recognition Workshop (CVPRW). IEEE, 2017*. This dissertation author is the primary author of this paper.

Chapter 3 contains reprints of Jeng-Hau Lin, Justin Lazarow, Yunfan Yang, Dezhi Hong, Rajesh K. Gupta, Zhuowen Tu. "Local Binary Pattern Networks for Character Recognition". *Submitted to International Conference on Computer Vision 2019 (ICCV)*. This dissertation author is the primary author of this paper.

Chapter 4 contains reprints of Jeng-Hau Lin, Atieh Lotfi, Vahideh Akhlaghi, Zhuowen Tu, and Rajesh K. Gupta, "Accelerating Local Binary Pattern Networks with Software-Programmable FPGAs", *Design, Automation, and Test in Europe (DATE), 2019*. This dissertation author is the

primary author of this paper.

Chapter 5 contains the re-organized reprints of Xun Jiao, Mulong Luo, Jeng-Hau Lin, Rajesh K. Gupta. “An Assessment of Vulnerability of Hardware Neural Networks to Dynamic Voltage and Temperature Variations”. *In Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD). IEEE Press, 2017. p. 945-950*. The dissertation author is the co-author and primary investigator of this paper.

All my co-authors, Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Justin Lazarow, Yunfan Yang, Dezhi Hong, Atieh Lotfi, Vahideh Akhlaghi, Xun Jiao, Mulong Luo, Prof. Mani Srivastava, Prof. Zhiru Zhang, Prof. Zhuowen Tu, and Prof. Rajesh K. Gupta, have kindly approved the inclusion of the aforementioned publications in my thesis. Thanks to many anonymous reviewers for the comments to improve our works. The research projects are supported by NSF IIS-1618477, NSF CCF-1029783, DARPA HR0011-16-C-0037, Samsung Research America, and Qualcomm Technologies.

VITA

2005	B.S., Engineering, Electrical Engineering, National Taiwan University
2007	M.S., Communication Engineering, National Taiwan University
2019	Ph.D., Computer Science (Computer Engineering), University of California, San Diego

PUBLICATIONS

Jeng-Hau Lin, Atieh Lotfi, Vahideh Akhlaghi, Zhuowen Tu, and Rajesh K. Gupta, "Accelerating Local Binary Pattern Networks with Software-Programmable FPGAs", *Design, Automation, and Test in Europe (DATE), 2019*.

Jeng-Hau Lin, Justin Lazarow, Yunfan Yang, Dezhi Hong, Rajesh K. Gupta, Zhuowen Tu. "Local Binary Pattern Networks for Character Recognition". *Submitted to International Conference on Computer Vision 2019 (ICCV)*.

Jeng-Hau Lin, Tianwei Xing, Ritchie Zhao, Zhiru Zhang, Mani Srivastava, Zhuowen Tu, and Rajesh K. Gupta."Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration." *In Computer Vision and Pattern Recognition Workshop (CVPRW). IEEE, 2017*.

Xun Jiao, Mulong Luo, Jeng-Hau Lin, Rajesh K. Gupta. "An Assessment of Vulnerability of Hardware Neural Networks to Dynamic Voltage and Temperature Variations". *In Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD). IEEE Press, 2017. p. 945-950*.

Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, Zhiru Zhang. "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs". *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*.

ABSTRACT OF THE DISSERTATION

Resource Efficient and Error Resilient Neural Networks

by

Jeng-Hau Lin

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2019

Professor Rajesh K. Gupta, Co-Chair
Professor Zhuowen Tu, Co-Chair

Driven by advances in microelectronics, processing and memory systems, neural networks have matured to be useful in a large number of applications that rely upon optimization using stochastic gradient descent. Along with this maturity comes the drive to deploy neural networks in devices that are much closer to the physical systems, the so-called edge devices, that is, devices at the edge of the internet. Whereas these devices are naturally constrained in terms of energy, power, memory, memory bandwidth and computation, the application needs for recognition, optimization tasks has only grown. These trends drive us to seek ultra-efficient implementations of neural networks as algorithms, architecture or directly implemented in hardware.

This thesis specifically explores the algorithmic optimizations of neural networks, specifically convolutional neural networks, that make them suitable for implementation in microelectronic hardware (as ASICs or FPGAs) or in embedded computing systems. As we push the limits of hardware, we naturally run into the growing conservative guardbands that circuit designers use in combating effects of variability in nanometer-scale microelectronic devices.

To address the technical challenges posed, we seek cross-layer solutions to the problems of the high algorithmic demands incurred by deep learning methods and error vulnerability due to hardware variations. This dissertation is organized as follows. We begin with a review of the methods and technologies proposed in the literature including weight encoding, filter decomposition, network pruning, efficient structure design, and precision quantization. We explore in depth binarization of neural network parameters in a binarized neural network (BNN) as a means to improve implementation efficiency on an FPGA target. We introduce optimizations particularly suitable to BNN implementation. Then, we extend BNN on the algorithmic layer with the binarized separable filters and proposed BCNNw/SF. Although the quantization and approximation benefit hardware efficiency to a certain extent, the optimal reduction or compression rate is still limited by the core of the conventional deep learning methods – convolution. To improve further, we introduce the local binary pattern (LBP) to deep learning because of LBP's low complexity yet high effectiveness. We name the new algorithm LBPNet, in which the feature maps are created in a similar fashion to the traditional LBP using comparisons. Our LBPNet can be trained with the forward-backward propagation algorithm to extract useful features for image classification. LBPNet accelerators have been implemented and optimized to verify their classification performance, processing throughput, and energy efficiency. We also demonstrate the error immunity of LBPNet to be the strongest compared with the subject MLP, CNN, and BCNN models since the classification accuracy of the LBPNet is decreased by only 10% and all the other models lose the classification ability when the timing error rate exceeds 0.01.

Chapter 1

Introduction

The increasing algorithmic demands have galvanized the prosperity of hardware computing capabilities on both cloud machines and edge devices. Currently popular deep learning applications, such as robotics, automated driving, and the general purpose artificial intelligence, fusing and digesting large data continue to demand more processing resources, memory and memory bandwidth to meet application needs. For instance, to fulfill the level 4 driving automation [int16] defined by the Society of Automotive Engineers International (SAE), the automobile manufacturers usually coalesce the on-vehicle sensors to not only 360° perspective but also overlapped with each other to enhance the comprehension of the environment as shown in Figure 1.1. Furthermore, multiple subtasks, such as lane detection and traffic sign recognition, may exploit the same sensor reading or video recording and increase the burden of computation. As a result, the workload of the central workstation on a vehicle requires the ADS providers to build more and more powerful single-instruction-multiple-data (SIMD) machines on vehicles, which beget high energy consumption and high hardware cost.

In this dissertation, we shall examine the resource issue in terms of energy, memory, and computation. We realize the gap between the BNN theory [HCS+16a, CHS+16] and physical implementation through implementation of an FPGA accelerator for BNN [ZSZ+17]. Finally, we highlight a new genre of networks utilizing fewer hardware resources while maintaining the near state-of-the-art performance.

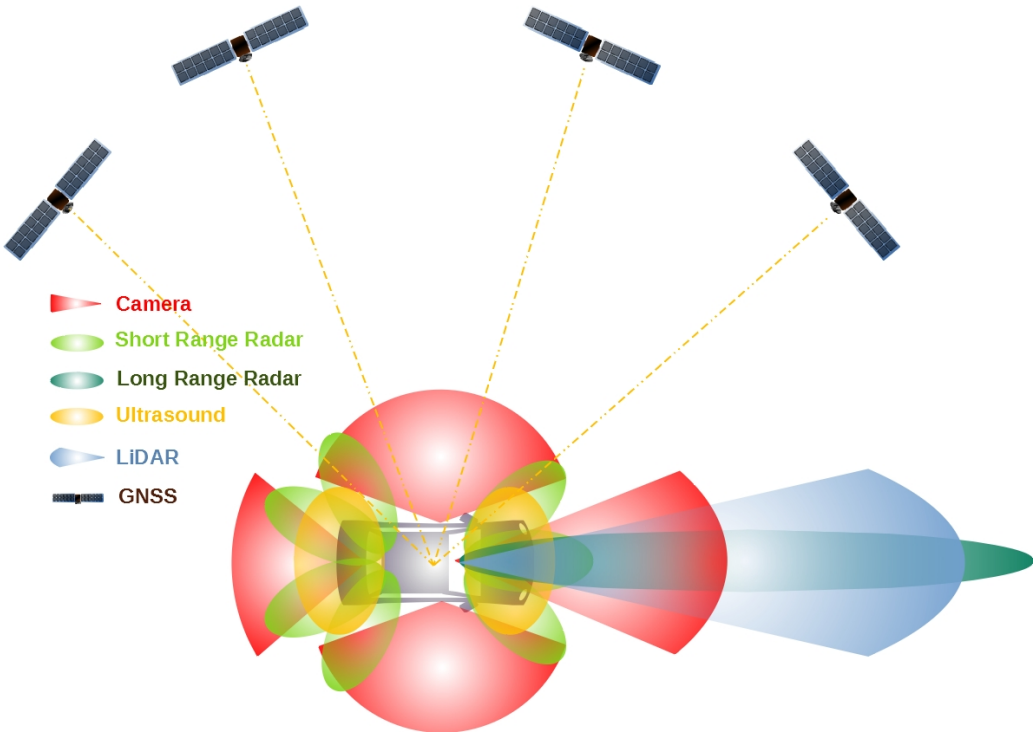

Figure 1.1. The sensor coverage of an automated driving system (ADS).

1.1 Background and Problems

Although the deep learning community has prospered over the past few decades, the inference of neural networks still overburdens resource-limited embedded hardware [RORF16a] for internet of things (IoT) platforms. Table 1.1 lists the performance and hardware demands of five prestige CNN models designed for the ImageNet [DDS$^+$09] dataset. While pursuing a higher classification accuracy on ImageNet, the CNN models have grown deeper and deeper to extract features beneficial to the classification. The core of feature extraction is a convolution, which is composed of a series of multiplication-and-accumulation (MAC) operations. The extracted features, i.e., feature maps or activations, must be stored and fed to the succeeding layer as an input. Billions of the MAC operations mentioned above incur two design challenges for the practical uses of the models.

First, the computation requirements exceed beyond the computation capability of IoT devices powered by embedded CPUs, e.g., from the low-cost Cortex M0™ with 64 Mega

Table 1.1. Model sizes and computational requirements for five well-known CNN architectures for the classification on ImageNet: AlexNet [KSH12], VGG16 [SZ15], GoogLeNet [SLJ+15], ResNext101 [XGD+17], and DenseNet201 [HLvdMW17].

	AlexNet	VGG-16	GoogLeNet	ResNeXt-101	DenseNet-201
Top-1 Acc.	58.2%	71.5%	68.7%	77.7%	70.0%
Top-5 Acc.	80.8%	90.1%	89.0%	94.1%	93.7%
Param. Size	233 MB	528 MB	51 MB	319 MB	77 MB
Feat. Size	3 MB	58 MB	26 MB	273 MB	196 MB
FLOP	0.7 Bn	19.6 Bn	1.5 Bn	18.9 Bn	10.9 Bn

instruction per second to the high-end Cortex A73™ with 2.45 Giga instructions per second, to deliver a real-time frame rate higher than 30 frames per second (fps). Even though the deep models are carried out on embedded GPUs, such as Jetson TX2™ with 1 Tera FLOP/s, to achieve a frame rate higher than 50 fps ideally, the embedded GPUs must be preempted for the concerned single model so that no other tasks can introduce round-robin multitasking to share the throughput. Despite that the NVIDIA multiple-process service™ [Nvi11] (MPS) is scheduled to support embedded GPUs in the future, all the requirements of MPS must be strictly satisfied so that the service can optimize the scheduling overheads, but the 'one fails, all fail' property of MPS still restrains the application and development. As a result, the guardbands in terms of computation units and energy budgets are difficult to estimate or design because of the enormous algorithmic demands.

Second, the memory sizes of the trained parameters and the extracted feature maps in the hidden layers take hundreds of megabytes (MB), which is larger than the on-chip memory in most processors. Adding off-chip DRAM for extra storage space is the conventional solution, but the cost regarding the DRAM size and the speed bottleneck formed by the memory input/output (I/O) bandwidth must be expected. Moreover, the streaming and buffering mechanisms for big data and large network parameters between the off-chip DRAM and on-chip memory further deteriorate the throughput if the timing guardbands are over-designed.

As we optimize design, we encounter limitations caused by conservative guardbands for hardware variations. For example, the gradual voltage decrease due to low battery or the

temporary voltage droop resulted from an ill-regulated power distribution network (PDN) can introduce drifts of the transistor bias points. On the other hand, the rising temperature resulted from the ambient conditions or the internal power dissipation directly decrease the electron/hole mobility. Both of the two common hardware variations weaken the gate charging/discharging of transistors and hence induce timing errors. System designers have passively defined guardbands to prevent the timing errors from causing accuracy degradation or system failure.

As a summary, the resource deficiency problem aroused by the deep models and the timing errors resulted from hardware variations have hampered IoT applications. This dissertation targets to the cross-layer optimization for the challenges from algorithm to hardware implementation and proposes new genres of neural networks that efficiently utilizes computation resources and effectively segregates the timing error propagation.

1.2 Existing Solutions

We have observed that the solutions fell in several streams. In this section, we discuss the methodologies and limitations of existing solutions.

1.2.1 Weight Encoding

Without retraining, the storage size of a trained model can be reduced by encoding. The final stage of Deep compression [HMD15] utilized Huffman coding to take advanced of the profiled weight distribution and achieved the lossless encoding. The succeeding work EIE [HLM$^+$16] encoded the sparse weights into the compressed sparse column (CSC) format as a pair of vectors representing the values and indices in the original weights and then devised a customized hardware architecture for the inference with the CSC format.

Weightless [RGA$^+$17] combined Bloomier filter with retraining to propose a lossy encoding approach. Regardless of the loss introduced by the encoding, Weightless's retraining provided enough error resilience so that the ensemble results showed no significant accuracy loss on both MNIST and ImageNet.

1.2.2 Filter Decomposition

Instead of profiling and encoding the learned parameters, some works [RSLF13, JVZ14, AP16] decomposed the convolutional filters into vectors and used them to approximate the original filters. These works differed in the decomposition methods and whether retraining was involved. As a result, the low-rank approximation guaranteed both the model reduction and speed-up since two successive 1-dimensional (1-D) convolutions, time complexity $O(k)$, can be implemented faster than one 2-D convolution, time complexity $O(k^2)$, given k is the kernel width.

1.2.3 Pruning

We introduce both static and dynamic pruning techniques that can reduce hardware complexity.

Static Pruning

Optimal Brain Damage (OBD) [LDS90] was proposed to prune a fully-connected neural network into a more efficient model. Intuitively, removing the edges with small values [HKPH91, HMD15, HPTD15] should not deteriorate the performance since the pruned weights in the dot-product computation did not contribute as much as other weights do. However, OBD claimed that magnitude did not equal salience, which was the extent of causing training errors. More specifically, the authors defined the salience of a parameter as "the change in the objective function (cost) caused by deleting that parameter."

If a perturbation dU of parameters was presented to the cost l, the change of the cost is shown in Eq. 1.1 in its Taylor series.

$$dl = \sum_i g_i du_i + \frac{1}{2}\sum_i h_{ii} du_i^2 + \frac{1}{2}\sum_{i \neq j} h_{ij} du_i du_j + O(||dU||^3), \tag{1.1}$$

where $g_i = \frac{\partial l}{\partial u_i}$ was the gradient, and the second and third terms $h_{ij} = \frac{\partial^2 l}{\partial u_i \partial u_j}$ made up the

Hessian of l. The first term vanished after a training process since the gradients with respect to (w.r.t.) weights must be zero or approaching zero by the end of the convex optimization. Furthermore, the authors introduced the "diagonal" approximation to negate the effects resulted from cross-coupling terms, the Hessian elements for $i \neq j$.

Optimal Brain Surgeon (OBS) [HSW93] pinpointed, justified, and demonstrated the diagonal approximation was far from the truth in reality, and the wrong approximation led OBD to prune the misjudged weights and thereby even multiple retraining cycles cannot compensate the performance loss of the damaged networks. The Hessian and its inverse matrix of arbitrary network parameters can be calculated without the erroneous diagonal approximation proposed in OBD. With the correct hessian matrix calculated, OBS used a simple for-loop to gradually prune the redundant weights and adjust the remaining until the predefined error tolerance was reached or all less salient weights have been pruned. OBS pruned the weights to achieve a global minimum on the error surface without any retraining process, but the computationally expensive calculation of the Hessian matrices and their corresponding inverses in every update iteration impeded the application of OBS on larger networks or CNNs.

Although named as Dynamic Network Surgery [GYC16] (DNS), the pruning was done off-line. The term "Dynamic" was used because of the heuristic and antagonistic mechanism including magnitude-oriented pruning and recovery splicing. Without the support of error resilience from retraining cycle, DNS was able to reduce AlexNet's model size to 17.7X smaller, while maintaining the classification accuracy.

Dynamic Pruning

The methods in the previous section pruned and fixed the networks before inference, but SnaPEA [AYS+18] performed pruning during inference on a pretrained model. To avoid severe calculation overheads, the pruning in the inference phase can only be greedy and straightforward. Even though OBS has proven that the magnitude pruning trapped the resultant model in a local minimum, the rectified linear unit (ReLU) after the accumulation provided an opportunity to

perform magnitude pruning without any performance degeneration. SnaPEA applied sign-bit speculation to stop accumulating the unnecessary multiplications and delivered the same output activation as that before pruning. SnaPEA differed traditional magnitude pruning from the subject of thresholding: Previous works statically pruned networks according to weight magnitudes, yet SnaPEA dynamically and speculatively pruned by monitoring the magnitudes of activations. The dynamic pruning, therefore, required a customized of the hardware architecture and arithmetic logic unit.

1.2.4 Efficient Structure

More works were exploring efficient network or sub-block structures to push accuracy while reducing computations [SWLS+15, SVI+15, SIV16, HZRS15, XGD+17].

GoogLeNet

As shown in Table 1.1, GoogLeNet comprised the Inception [SWLS+15, SVI+15, SIV16] building blocks that utilized the split-transform-merge strategy to efficiently extract useful features and achieved 68.7% top-1 accuracy on ImageNet. The memory and computation usage were efficient compared with VGG-16, which achieved slightly higher accuracies but used almost 10 times of the memory of GoogLeNet and 13 times more operations.

ResNeXt

Inspired by the highly modulized design in VGG and NIN [LCY14], the group convolution of AlexNet [KSH12] and the branch convolution structure of Inception and ResNet [HZRS15], ResNeXt [XGD+17] proposed a building block composed of identical convolutional transformations and minimized human efforts in the architecture engineering with a new design dimension 'cardinality.' The cardinality is the number of the identical transformation branches in a ResNeXt block. Increasing cardinality benefited the model performance more especially when the effect of increasing of layers started to diminish.

1.2.5 Quantization

Quantization makes NNs hardware-friendly since the resulting fixed-point numbers [ZSZ+17, FHCW16, VB16, SPM+16, HMD15], ternary values [HS14, ZHMD17], or in the extreme case — binary values [HCS+16b, SHM14a, KS16, RORF16a] can be stored in smaller memory space and calculated with cheaper arithmetic logic units (ALUs). Notwithstanding the precision degradation of real numbers inevitably introduced the loss of information, the benefits still motivated researchers to compensate for the performance loss. Here we list several interesting works related to our contributions.

Fix-Point Quantization

Quantizations from 64-bit or 32-bit floating numbers to shorter bit-length fixed-point numbers were common in the implementation of hardware accelerators [ZSZ+17, FHCW16, VB16, SPM+16, HMD15], but most of them stopped at a precision no less than than 16-bit to avoid significant performance loss. However, once the error resilience of retraining was leveraged, the fix-point precision can be further reduced to a length of 2-bit [HS14, KS16, ZHMD17] with minor performance loss.

Expect Backpropagation

Expect Backpropagation (EBP) [SHM14a, CSML15] was proposed as a probabilistic training method without any meta-parameters, i.e., learning rates. The intuition behind EBP was to infer the most likely binarized weights given a training data set $P(W|D_n)$, where W represented the binary weights, and D_n was the entire training dataset. Most likely weights were those weights that produced targets through probabilistic forward propagation. All activations in EBP were binarized with a sign function.

Two approximations made EBP work:

- Mean-field: *Decoupling* the effects caused by the ensemble group of parameters into small independent events with no cross-coupling among the small events. This allowed EBP to

go through the weight optimization one by one instead of trying an exponential number of weight combinations exhaustively.

- Large fan-in: Assuming there were infinite neurons in the upper layer, the normalized input of each layer was a Gaussian distribution according to the Central Limit Theorem (CLT). This approximation provided a mathematic closed-form of the probabilities propagating forward and backward in EBP.

However, as shown in [SHM14a], although the sizes of dataset were as small as $4,000$, EBP's performance was inferior to traditional neural networks. The results indicated that the two approximations of EBP were too aggressive in ignoring the crossing effects between weights and assuming infinite fan-in for every neuron.

Binarized Neural Network and XNOR-Net

BNN [HCS$^+$16b] pushed the quantization to the extreme case — binary numbers. Two important ideas contributed to achieve the near state-of-the-art performance:

- Larger network: As with the well-known exclusive-OR problem [RHW85a], the solution was to promote the model to a deeper one and increase the number of neurons. Expanding networks increased the dimension for projection and added more hyperplanes for better classification.

- Batch normalization [IS15]: The normalization regularized and generalized the network to enforced the immunity against noise. Besides normalization, the batch normalization introduced two extra degrees of freedom, scaling and shifting, for every pixel to further compensate for the additive noises.

Binarization can be considered as an addition of noise (Eq. 1.2) into original real value weights and activation according to either deterministic or stochastic rules, as shown in Eq. 1.3, and Eq. 1.4, respectively.

$$x_b = x + n, \tag{1.2}$$

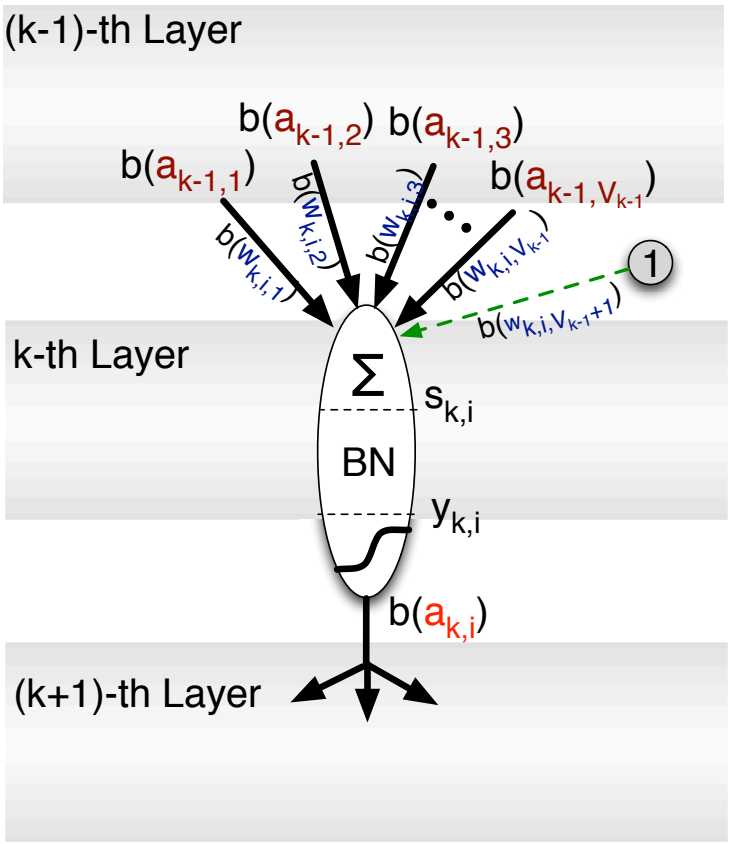

Figure 1.2. Batch normalization, marked as BN, before the nonlinear activation function.

where x_b is the binarized information, x is the real value information, and n is the additive noise. It has been demonstrated in the literature that with regularization, neural networks are robust against additive quantizing noise [CBD15, KS16, DSH13, CDB14, MH12, LCMB15].

$$n = \begin{cases} 1-x, & \text{if } x \geq 0.5 \\ -x, & otherwise \end{cases} \tag{1.3}$$

$$n = \begin{cases} 1-x & \text{with probability } p = \sigma(x) \\ -x, & \text{with probability } p = 1-\sigma(x) \end{cases}, \tag{1.4}$$

where $\sigma(x)$ is the sigmoid probabilistic density function.

Although the original proposed use of the batch normalization was after an activation function, BNN intelligently introduced batch normalization between the dot-product and the

activation function as shown in Figure 1.2. The swapping successfully enable each neuron to learn a linear transformation for better use of the nonlinear part.

Additionally, the two ideas can be applied towards convolutional neural networks to build binarized CNNs (BCNNs). The authors have verified BCNNs on more challenging datasets such as CIFAR-10 and SVHN and achieved the near state-of-the-art results.

Instead of resorting to batch normalization to regularize the additive noise introduced by binarization, XNOR-Net [RORF16a] alternatively adopted an additional scaling layer to generalize the training method for binarized weights further. The scaling factors were calculated from the of the L1-norm errors of weights. Another difference from other neural networks was the sequence of processing layers in a CNN. The authors moved the convolution behind the binary activation function to further decrease information loss due to binarization. While BNN merely achieved a 28.9% on ImageNet top-1 accuracy, XNOR-Net achieved a much higher accuracy of 44.2%.

In summary, weight encoding, filter decomposition, pruning, and quantization approximated the trained models in different granularities and introduced approximation errors increasing the model loss typically. The exploration of efficient structures required the expertise of structural priors and enormous time and efforts in the expedition. Therefore, we chose to start from quantization because the quantization is a common step in hardware implementation and optimization. Through the designing of an accelerator for the extreme-case quantization, BNN, we can realize the gap between theoretical quantization and practical implementation.

1.3 Preliminary Results from Building a BNN Accelerator

We choose to implement BNN [HCS$^+$16a, CHS$^+$16] on the software-programmable FPGA as our starting point of the following research because of the open-sourced availability and architectural simplicity. Table 1.2 lists the architecture of the implemented Binarized CNN and the model size including feature maps and parameters. The network architecture is the

Table 1.2. The architecture of the BinaryNet CIFAR-10 BNN — Output bits refer to the total size of the output feature maps. Conv stands for a convolutional layer, Pool denotes for a max pooling layer, and FC means a fully-connected layer. The weight bits exclude batch normalization parameters, whose total size after optimization is 0.12M bits. This is less than 1% of the size of the weights.

Layer	Input ch. N_{in}	output ch. N_{out}	Output dim d^2	Output bits	Weight bits
Conv1	3	128	32 x 32	131K	4480
Conv2	128	128	32 x 32	131K	148K
Pool	128	128	16 x 16	33K	-
Conv3	128	256	16 x 16	66K	297K
Conv4	256	256	16 x 16	66K	593K
Pool	256	256	8 x 8	16K	-
Conv3	256	512	8 x 8	33K	1.2M
Conv4	512	512	8 x 8	33K	2.4M
Pool	512	512	4 x 4	8192	-
FC1	8192	1024	1	1024	8.4M
FC2	1024	1024	1	1024	1.0M
FC3	1024	10	1	10	10K
Total				519.25K	14.2M
Conv				517.19K	4.59M
FC				2.06K	9.46M

same as the CIFAR-10 model delivering 88.60% accuracy in the BNN paper [HCS+16a]. The weights and bias of fully-connected layers (FC layers) take 9.46M bits and dominate the memory footprint; the binarized convolutional kernels take other 4.59M bits. Even with pooling and binarization, we still need an off-chip DRAM to store the parameters and a buffering mechanism and yield enough block RAMs (BRAMs) to feature maps and activations.

Inside each building block of BNN, every convolutional layer (Conv layer) is followed with a batch normalization layer (BatchNorm layer) [IS15]. The parameter size of a BatchNorm layer can be quantized to take only 0.12M bits, but the linear transformation for all input pixels in a BatchNorm layer cannot be replaced with cheap XNOR operation. Fortunately, FINN [UFG+17] proposed a smart method as shown in Figure 1.3 to drastically improve the resource utilization on FPGAs by avoiding the linear transformation at runtime by calculating the by-pixel thresholds because most of the BatchNorm layers in BNN were followed by a sign

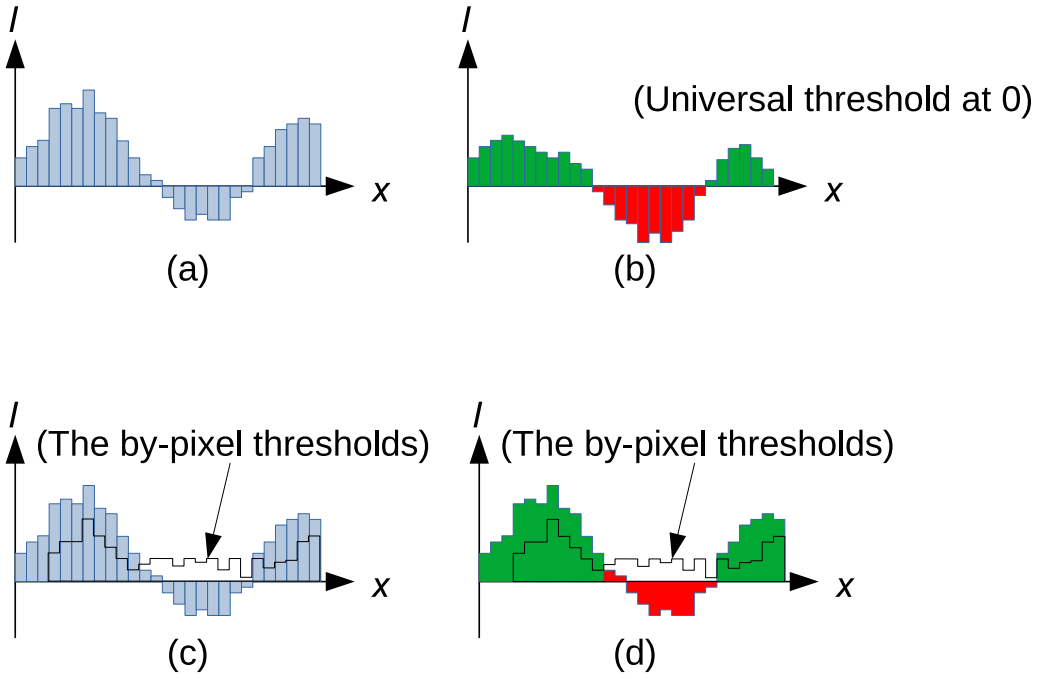

Figure 1.3. The pre-calculated batch normalization thresholding proposed in FINN [UFG+17]: (a) The input pixel strengthes; (b) applying the universal threshold '0' on the scaled and schifted pixels, the green color represents positive output and red represents the negative; (c) the input pixels and the pre-calculated thresholds (yellow); (d) thresholding the input pixels directly without the linear transformation.

function to generate binary outputs. Instead of using a universal '0' as the nonlinear threshold, FINN pre-calculated the thresholds for different pixels and used the new thresholds to decide whether to output a zero or one immediately upon receiving the input. This smart idea can be used in our future designs to offload the calculations in an FPGA's digital signal processing (DSP) blocks to the look-up-table (LUT) blocks.

We applied hardware optimizations, such as loop unrolling, quantization, and customized sub-word buffering, to implement the BNN accelerator on CIFAR-10 without losing any classification accuracy. Table 1.3 lists measured the latency, power, and energy efficiency in img/sec/Watt of our FPGA implementation together with the inference profiles on Intel Xeon Eg-2640 CPU, NVIDIA Tesla K40 GPU and NVIDIA Jetson TK1 embedded GPU. Although inference on a NVIDIA Tesla K40 is 9 times as fast as our BNN accelerator, our energy efficiency

Table 1.3. Energy efficiency comparison between our BNN accelerator on FPGA and software implementations on Intel Xeon Eg-2640 CPU, NVIDIA Tesla K40 GPU, and NVIDIA Jetson TK1 embedded GPU. Time is in milliseconds (ms). Conv1 is the first floating-point Conv layer, Conv2-5 are the binary Conv layers, FC1-3 are the FC layers. A indicates a value we could not measure. Numbers with * are sourced from datasheets. The last row shows power efficiency in throughput per Watt.

	Execution time per image (ms)			
	CPU	GPU	mGPU	FPGA(this work)
Conv1	0.68	0.01	-	0.11
Conv2-5	13.20	0.68	-	4.22
FC1-3	0.92	0.04	-	2.03
Total	14.80	0.73	90	6.36
Speedup	2.3x	0.11x	14.2x	1x
Power(Watt)	95*	235*	3.6	4.7
imgs/sec/Watt	0.71	5.83	3.09	33.5

is 5.7X higher than the GPU inference. Our BNN accelerator beats the inference on CPU and mGPU in both speedup and energy efficiency. For the details of hardware optimization on our BNN accelerator, please refer to our accelerator paper [ZSZ+17]

From the practical implementation of BNN, we have learned that there exists some room for improvements in both BNN's theory and the hardware implementation. For example, the combinations of the binarized convolutional kernels are very limited and have not been exploited. The zero paddings in BNN's theory created an illogical flaw because BNN trained the weights to be either '+1' or '-1', but the padding with '0' introduced the third state of the "binary" values. We have circumvented the flaw with a "do-nothing" mark when the padding is performed, but this must be solved to make the feature maps truely binarized. Moreover, FINN's LUT batch normalization and SIMD architecture can also be employed to speed up the deep learning accelerator in the future.

1.4 Thesis Contribution

Figure 1.4 depicts the organization of this dissertation. We pursue efficient algorithms alleviating the hardware resource deficiency problem. In chapter 2, we first extend BNN with

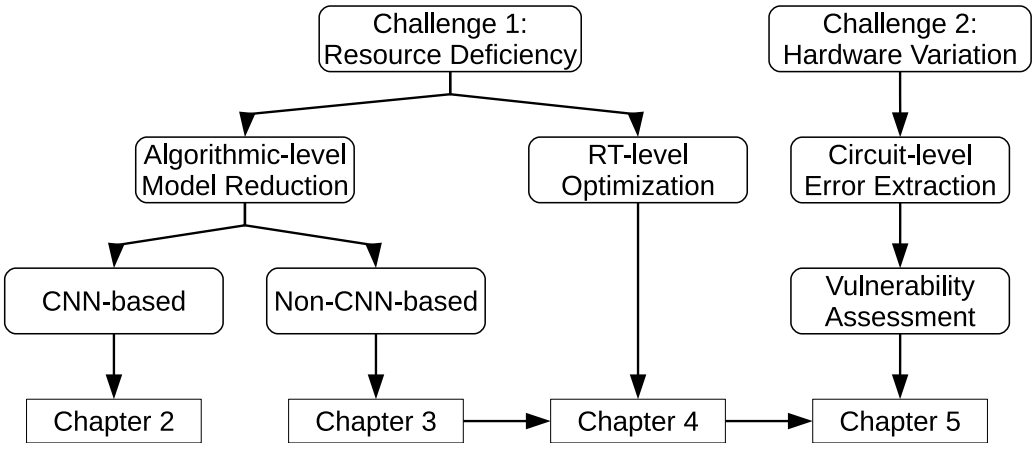

Figure 1.4. The dissertation organization.

our binary separable filters to further reduce the model sizes and replace 2-D convolutions with 1-D convolutions to gain speed-ups. We name the first work BCNNw/SF. A hardware accelerator for BCNNw/SF has been implemented and measured to demonstrate the hardware benefits in terms of memory footprint and computations.

Then, we set off to rethink the core of the concerned deficiency, which was resulted from convolution. To extract useful features, there existed numerous streams in computer vision other than CNN, such as SIFT [Low04], HOG [DT05, WHY09], ShapeContext [BMP02], and the local binary pattern (LBP) [OPH96]. Among these methods, we re-visite the idea of LBP and explore possibilities bringing LBP to deep learning so that we can extract the common features of images with chaep hardware primitives (chapter 3). We implement FPGA accelerators for LBPNets in chapter 4.

Last but not least, we assesse the vulnerability of four genres of NNs including MLP, CNN, BNN, and LBPNet, to understand the extent of performance degeneration resulted by physical variations, i.e., as voltage and temperature fluctuations (chapter 5). LBPNet surprisingly provides high immunity to the variation errors owing that the high parallelism characteristic of LBPNet separates and mitigates the error propagation during the inference stage.

Chapter 1 includes part of the results of Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, Zhiru Zhang. “Accelerating Binarized

Convolutional Neural Networks with Software-Programmable FPGAs". *I*n Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA) as the motivation of the following research in chapter 2 and chapter 3. This dissertation author is the co-author of this paper.

Chapter 2

Binarized Convolutional Neural Network with Separable Filters

State-of-the-art convolutional neural networks are enormously costly in both compute and memory, demanding massively parallel GPUs for execution. Such networks strain the computational capabilities and energy available to embedded and mobile processing platforms, restricting their use in many important applications. In this chapter, we push the boundaries of hardware-effective CNN design by proposing BCNN with Separable Filters (BCNNw/SF), which applies Singular Value Decomposition (SVD) on BCNN kernels to further reduce computational and storage complexity. To enable its implementation, we provide a closed-form of the gradient over SVD to calculate the exact gradient with respect to every binarized weight in backward propagation. We verify BCNNw/SF on the MNIST, CIFAR-10, and SVHN datasets, and implement an accelerator for CIFAR-10 on FPGA hardware. Our BCNNw/SF accelerator realizes memory savings of 17% and execution time reduction of 31.3% compared to BCNN with only minor accuracy sacrifices.

2.1 Introduction

Convolutional neural networks (CNN) [LBD$^+$89a] was first shown to be a promising technique for vision classification since 1989, and Deep Neural Network (DNN) [HOT06] surmounted the supremacy of the support vector machine in the ImageNet challenge in 2006.

Since then, the versatility of CNNs has been demonstrated in many fields, including web search models [HHG+13a], aerial image classification [ME12], and biomedical analysis [SLF14a]. The number of CNN applications and network architectures continues to grow year after year.

Although the community of neural networks has been prospering for decades, state-of-the-art CNNs still demand significant computing resources (i.e., high-performance GPUs), and are eminently unsuited for resource and power-limited embedded hardware or Internet-of-Things (IoT) platforms [RORF16b]. Reasons for high resource needs include the complexity of connections among layers, the sheer number of fixed-point multiplication and accumulation (MAC) operations, and the storage requirements for weights and biases. Even if network training is done off-line, only a few high-end IoT devices can realistically carry out the forward propagation of even a simple CNN for image classification.

Binarized convolutional neural networks (BCNNs) [HCS+16a, CBD15, SHM14b, KS16, RORF16b] have been proposed as a more hardware-friendly model with extremely degenerated precision of weights and activations. BCNN replaces floating or fixed-point multiplies with XNOR operations (which can be implemented exceptionally efficiently on ASIC or FPGA devices) and achieved near state-of-the-art accuracy on many real-world image datasets at time of publication. Unfortunately, this hardware efficiency is offset by the fact that number of parameters in a BCNN is typically tens or hundreds more than that in a CNN of equal accuracy. To make BCNNs practical, an effective way to further reduce the model size is required.

We introduce separable filters (SF) on binarized filters, as shown in Figure 2.1(c), to further reduce the hardware complexity in two aspects:

- SF reduces the number of possible unique d-by-d filters from 2^{d^2} to just 2^{2d-1}, enabling the use of a small look-up table during the forward propagation. This directly results in the $\frac{(d-1)^2}{d^2}$ reduction of memory footprint.

- SF replaces each d-by-d 2D convolution with two d-length 1D convolutions, which reduces the number of MAC operations by d/2. This translates to either speedup or the

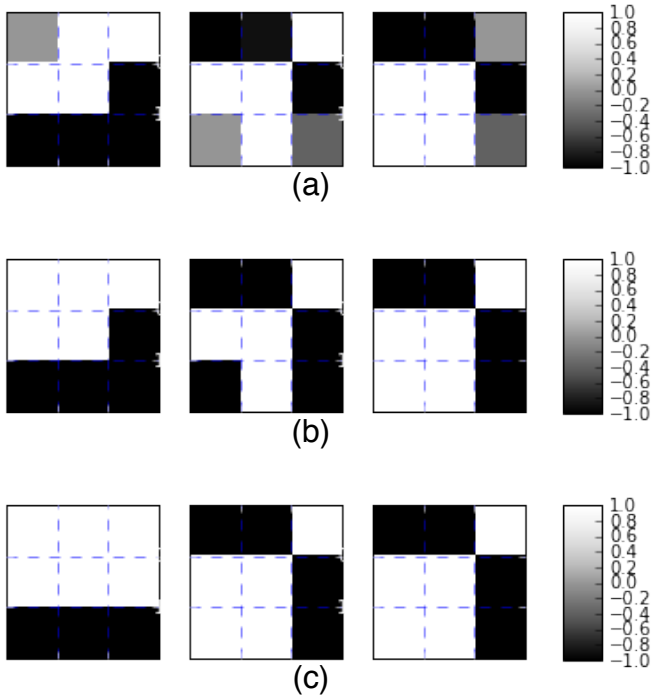

Figure 2.1. Comparison of filters: (a) the original floating-point filters; (b) the same filters after been binarized; (c) the approximated separable binary filters.

same throughput with fewer resources.

In addition, we propose two methods to train BCNNw/SF:

Method 1 - Extended Straight-through Estimator (eSTE):

We take the rank-1 approximation for SFs as a process adding noise into the model and rely on batch normalization to regularize the noise. During backward propagation, we extend the straight-through estimator (STE) to propagate gradient across the decomposition.

Method 2 - Gradient over SVD:

Now that SVD is a linear algebraic technique, we can go through the analytic closed-form of the gradient over SVD to push the chain rule in backward propagation to the binarized filters, which is the filter before SVD.

The rest of the paper is organized as follows: Section 2.2 provides a brief survey of previous works, Section 2.3 presents the design of BCNNw/SF and some implementation details,

Section 2.4 presents two methods for the training of BCNNw/SF., Section 2.5 shows experimental results, Section 2.6 describes the implementation of BCNNw/SF on an FPGA platform, and Section 2.7 concludes the paper.

2.2 Related Works

We leverage the lightweight method for training a BCNN [HCS+16a, CHS+16], which achievedthe state-of-the-art classification results on MNIST, CIFAR-10, and SVHN. Two essential ideas contributed to the effectiveness of their BCNN:

Batch normalization with scaling and shifting [IS15]: A BatchNorm layer regularized the training process by shifting the mean to zero, making binarization more discriminative. It also introduced two extra degrees of freedom in every neuron to further compensate for additive noises.

A larger Model: As with the well-known exclusive-OR problem [RHW85b], using a larger network increased the power of the model by increasing the number of dimensions for projection and making the decision boundary more complex.

XNOR-Net [RORF16b], an alternative BCNN formulation, relied on a multiplicative scaling layer instead batch normalization to regularize the additive noise introduced by binarization. The scaling factors are calculated to minimize the L1-norm error between real-valued and binary filters. While BNN did not perform well on ImageNet [DDS+09] with a top-1 error rate of 72.1%, XNOR-Net improves this error rate to 55.8%.

Learning with Separate Filters [RSLF13] proposed a rank-1 approximate method to replace the 2-D convolution in a CNN with two successive 1-D convolutions. Every filter was approximated by the outer product of a column vector and a row vector which were from the resultant of Singular Value Decomposition (SVD). The authors proposed two schemes of the learning of separable filters: (1) retained only the largest singular value and corresponding vectors to reconstruct a filter; (2) linearly combined the outer products to lower the error rate.

However, the first scheme sacrificed too much performance because the other singular values can be comparable with the largest one in terms of magnitude. The second scheme was designed to compensate for the loss of performance, but more singular values used to recover a filter meant a lesser compression benefit from the approximation. Although learning with separable filters was computationally expensive, the low-rank approximation was an important idea to alleviate hardware complexity.

Inspired by Learning with Separable Filters [RSLF13], more research projects were conducted to explore a more economical model, i.e., networks with smaller memory requirements for the kernels. Low-Rank Approximation [JVZ14] proposed a filter compression method that analyzed the redundancy in a pre-trained model, decomposed the filters into single-channel separable filters, and then linearly combined separable filters to recover original filters. The decomposition was optimized to minimize the L2 reconstruction error of original filters. DecomposeMe [AP16] further reduced the redundancy by sharing the separated filters in the same layer. To alleviate the computational congestion of GoogLeNet [SWLS$^+$15], the Inception building block [SVI$^+$15, SIV16] was proposed as a multi-channel asymmetric convolutional structure, which had the same architecture as the second scheme of Low-Rank Approximation [JVZ14] but in different purposes: Inception used the asymmetric convolutional structure to avoid the expensive 2D convolutions and train the filter directly, while Low-Rank Approximation decomposed pre-trained filters to exploit both input and output redundancies. However, both Low-Rank Approximation and DecomposeMe required a pre-trained model, and both Low-Rank Approximation and Inception's multi-channel asymmetric convolution brought additional channels requiring a larger memory footprint.

Our proposed method differs from the three methods above because we maintain the network structure during training phase, train rank-1 separable filters directly, and then decompose the rank-1 filters into pairs of vector filters for hardware implementation. Last but not least, to the best of our knowledge no existing work provides an analytic closed form of the gradient of filter-decomposition process for backward propagation.

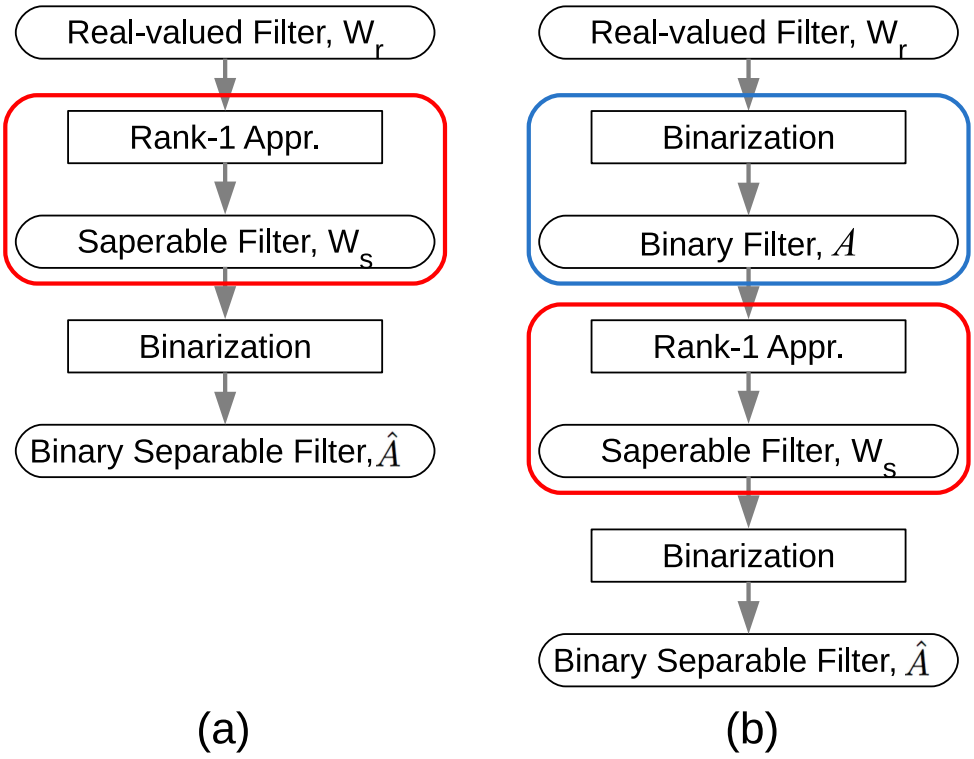

Figure 2.2. Comparison of the two SVD flows; (a) Choice 1: binarize the resulting floating-point filters of SVD; (b) Choice 2: decompose the binarized filters.

2.3 Binarized CNN with Separable Filters

Here we describe the theory of BCNN with Separable filter in details. Our main idea is to apply SVD on binarized filters to further reduce the memory requirement and computation complexity for hardware implementation. We present the details of forward propagation in this section and two methods of backward propagation in the next section.

2.3.1 The Subject of Decomposition

For BCNN, there are two approaches to binary filter decomposition. Figure 2.2 depicts the two choices. If we adopt flow 1 and apply the rank-1 approximation (the red box) directly on the real-valued filters, we cannot avoid real-time decomposition during training because the input filter has an infinite number of possible combinations of pixel strengths. Therefore, we introduce an extra binarization (the blue box) on the real-valued filters and apply the rank-1 approximation

on the binarized filters. The number of possible input filters of rank-1 approximation, with this, is limited to 2^{d^2}, where d is the width or height of a filter. Using flow 2, we can build a look-up table beforehand and avoid real-time SVD during training.

Naturally, the rank-1 approximation and the extra binarization will limit the size of the basis to recover the original filters and equivalently introduce more noise into the model, as shown in Figure 2.1 from (b) to (c). Instead of introducing an additional linear-combination layer to improve the accuracy, we leave the task to the two aforementioned reasons that make BNN work.

2.3.2 Binarized Separable Filters

Here we provide the detailed steps from binarized filters to binarized separable filters. The result of SVD on a matrix A includes three matrices as shown in Eq. 2.1.

$$A = UDV^T, \tag{2.1}$$

where U and V are the left and right singular matrices containing row singular vectors, and D is the singular value matrix with the sorted singular values on the diagonal trace from the largest to the smallest. Similar to real value rank-1 approximation for the separable filters, the binarized separable filters are obtained with an extra binarization process on the dominant singular vectors as shown in Eq. 2.2.

$$\hat{A} = b(U[:,1])b(V[:,1]^T), \tag{2.2}$$

where $U[:,1]$ and $V[:,1]$ stand for the left and right singular vectors corresponding to the largest singular value, respectively, and the function $b(.)$ denotes the binarization and can be implemented in either a deterministic function or a stochastic process [HCS+16a]. Please note the largest singular value is dropped because all singular values are always positive and have no effect on binarization.

Figure 2.3(a) and (c) illustrates a kernel with three filters before and after binarized

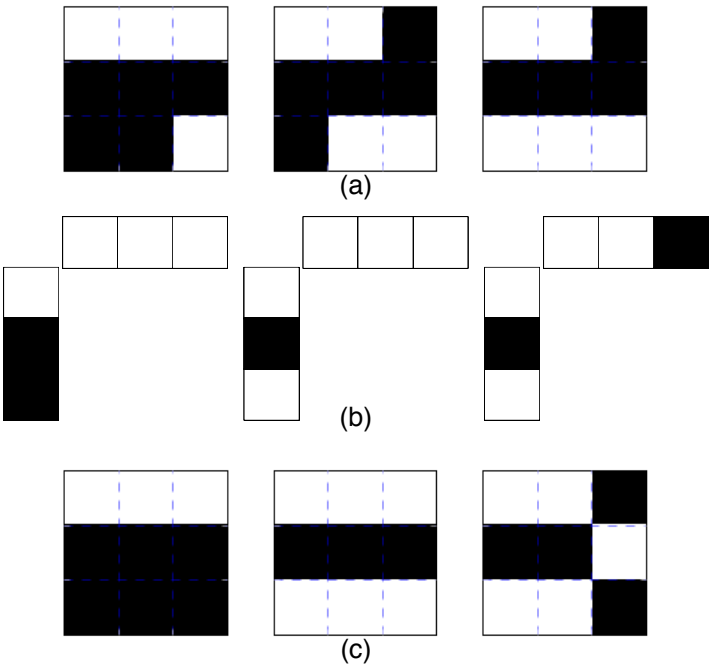

Figure 2.3. Example inputs and outputs of the binarized SVD and the recovered binary filters. (a) A kernel before approximation; (b) Pairs of vectors (u, v^T) of SVD; (c) A kernel after rank-1 approximation in which every filter is an outer product of u and v. The white and black colors stand for $+1$ and -1, respectively.

rank-1 approximation. Analogous with [HCS+16a] we keep a copy of the real-valued filters during each training iteration and accumulate the weight gradients on them since SGD and other convex optimization methods presume a continuous hypothesis space. This also allows us to train the kernels as if the model is real-valued without the need for penalty rules [RSLF13] during the backward propagation. For the test phase, all filters are binarized and approximated to be binarized separable.

In our FPGA implementation, we use the pairs of vectors in Figure 2.3(b) to replace 2-D filters and perform separable convolution, which involves a row-wise 1D convolution followed by a column-wise convolution in back-to-back fashion before accumulating across different channels. More details on the FPGA implementation are presented in Section 2.6.

2.3.3 Details of the Implementation

As mentioned in Section 2.3.1, the benefit of flow 2 is to leverage a finite-sized look-up table (LUT) to replace the costly SVD computation during the forward propagation of the training phase. Although the training takes place on a highly-optimized parallel computing machine, the LUT access is still a potential bottleneck if searching for an entry in the mapping is not efficient enough.

We build two tables to avoid real-time SVD. The first table is composed of all binarized separable filters. The number of entries in the first table can be calculated with Eq. 2.3.

$$K = 2^{2d-1}, \tag{2.3}$$

where d is the width or height of a filter, and K is the number of entries in the first table.

The second table is the mapping relationship between all possible binary filters to their corresponding binarized separable filters on the fist table. We design an estimation function to make the tables content-addressable. The key to index the first table can be obtained with Eq. 2.4.

$$key = \Lambda \cdot A, \tag{2.4}$$

where Λ is a vector or a matrix in the same size of A, and all elements in Λ are the weightings to convert a matrix A into a number. The most straightforward choice of Λ is the binary-to-integer conversion method. We take the first element in A as the least significant bit (LSB), so the Λ is designed as Eq. 2.5.

$$\Lambda = \begin{bmatrix} 2^0 & 2^1 & 2^2 & \dots & 2^N \end{bmatrix}, \tag{2.5}$$

where N is the number of elements of A, and $N = d^2$. With this simple hash function and the efficient broadcasting technique in Theano [The16], we are able to efficiently obtain the keys for all filters in a convolutional layer.

2.4 Backward Propagation of Separable Filters

Besides the extra degrees of freedom regarding filter approximation introduced to BCNNw/SF's forward propagation, two essential techniques make binarized separable filters work. In this section, we describe the two techniques for the training of BCNNw/SF in details.

2.4.1 Method 1: Extended STE (eSTE)

As shown in Figure 2.2(b), during the forward propagation, all filters must be degraded thrice. Since binarization can be considered as noise addition into the model and be regularized with batch normalization, the rank-1 approximation error, which is just another process adding extra noise, can be regularized as well. In details, we extend the straight-through estimator across the three degradation processes in Figure 2.2 to update the real-valued filters with the rank-1 approximated filters. Eq. 2.6 shows the backward propagation of the gradient w.r.t. the binarized rank-1 approximated filter,$g_{\hat{A}}$, to the gradient of real-valued filter, g_r.

$$g_r = g_{\hat{A}} 1_{|r|\leq 1} \tag{2.6}$$

This simple method relies on batch normalization to regularize the noise introduced by two binarization and one rank-1 approximation.

2.4.2 Method 2: Gradient over SVD

Whereas binarization is not a continuous function, BNN [HCS$^+$16a] resorted to the STE to update the real-valued weights with the gradient of loss w.r.t binarized weights. Howbeit, owing to the continuity of singular value decomposition, we are allowed to calculate the gradient w.r.t. the resultant of the first binarization, W_b. More specifically, the rank-1 approximation is differentiable because all of the three resultant matrices, i.e., U,D, and V, of SVD in Eq. 2.1 are differentiable w.r.t. every element of the original input matrix, A. From the approximation we adopt for separable filters as shown in Eq. 2.2, one can quickly obtain the derivative of $\hat{A}$

w.r.t. the elements of the original matrix before the approximation as Eq. 2.7, if the STE for binarization is applied.

$$\frac{\partial \hat{A}}{\partial a_{ij}} = \frac{\partial U[:,1]}{\partial a_{ij}} b(V[:,1]^T) + b(U[:1]) \frac{\partial V[:,1]^T}{\partial a_{ij}} \tag{2.7}$$

Jacobian of Singular Value Decomposition [PL00] provided the mathematical closed-form of the gradient of the three resultant matrices, as shown in Eq. 2.8, and 2.9.

$$\frac{\partial U}{\partial a_{ij}} = U\Omega_U^{ij} \tag{2.8}$$

$$\frac{\partial V}{\partial a_{ij}} = -V\Omega_V^{ij}, \tag{2.9}$$

where $\Omega_U^i j$ and $\Omega_V^i j$ are anti-symmetric matrices with zeros on their diagonals, and Eq. 2.10 and 2.11 can give us all off-diagonal elements.

$$\Omega_{U_{kl}}^{ij} = \frac{d_l u_{ik} v_{jl} + d_k u_{il} v_{jk}}{d_l^2 - d_k^2} \tag{2.10}$$

$$\Omega_{V_{kl}}^{ij} = \frac{d_k u_{ik} v_{jl} + d_l u_{il} v_{jk}}{d_k^2 - d_l^2} \tag{2.11}$$

Eq. 2.12 shows the general form of the differential equation.

$$\frac{\partial \hat{A}}{\partial a_{ij}} = \begin{bmatrix} \frac{\partial \hat{a}_{11}}{\partial a_{ij}} & \frac{\partial \hat{a}_{12}}{\partial a_{ij}} & \cdots & \frac{\partial \hat{a}_{1N}}{\partial a_{ij}} \\ \frac{\partial \hat{a}_{21}}{\partial a_{ij}} & \frac{\partial \hat{a}_{22}}{\partial a_{ij}} & \cdots & \frac{\partial \hat{a}_{2N}}{\partial a_{ij}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \hat{a}_{M1}}{\partial a_{ij}} & \frac{\partial \hat{a}_{M2}}{\partial a_{ij}} & \cdots & \frac{\partial \hat{A}_{MN}}{\partial a_{ij}} \end{bmatrix} \tag{2.12}$$

$$\frac{\partial \hat{a}_{kl}}{\partial a_{ij}} = b(U_{k1}) \sum_{n=2}^{N} V_{kn} \Omega_{V_{1n}}^{ij} - b(V_{l1}) \sum_{n=2}^{N} U_{ln} \Omega_{U_{1n}}^{ij} \tag{2.13}$$

From equations 2.8 to 2.11, we can derive every element in Eq. 2.12 as shown in Eq. 2.13 and

see there exist cross-terms between elements. The gradient of an SVD resultant matrix w.r.t. one element in the original input matrix is also a matrix of the same dimension, M-by-N, i.e., a single element's change in the input matrix can affect all other elements in the resultant of SVD. The intuition behind is that the rank-1 approximation is a matrix-wise filter-level mapping relationship rather than an element-wise operation, and multiple elements contribute to the mapping result of a filter.

To recap Eq. 2.12 with the chain rule calculation of backward propagation, we follow a similar fashion to how neurons on a preceding layer collect errors from the lower layer. Eq. 2.14 shows the dot product for collecting errors from a succeeding layer and propagate the errors to every element in binarized filters A. For method 2, we also build a table of the derivatives together with the binarized rank-1 approximation to avoid real-time calculation of Eq. 2.12.

$$\frac{\partial loss}{\partial a_{ij}} \equiv \frac{dloss}{d\hat{A}} \cdot \frac{\partial \hat{A}}{\partial a_{ij}} \tag{2.14}$$

2.5 Experiments

We conduct experiments on the Theano [The16] based on the Courbariaux's framework [Cou16], using two GPUs: NVIDIA GeForce GTX Titan X and GTX 970 to finish the training/testing process. In most of the experiments, we obtain the near state-of-the-art results using BCNNw/SF.

In this section, we describe the network structures we use and list the classification result on three datasets. We compare our result with related works and then analyze different perspectives, including the binarized separable filters and learning ripples.

2.5.1 Datasets and Models

We evaluate our methods on three benchmark image classification datasets: MNIST, CIFAR-10, and SVHN. MNIST is a dataset for 28x28 gray-scale handwritten digits, which

Table 2.1. Network architecture for different datasets. The dimension of a convolutional layer's kernel stands for number of kernels on the concerned layer M, number of channels C, the width of a filter W, and the height of a filter H; the dimension of a fully-connected layer's weights means the number of preceding layer's neurons and the number of the concerned layers' neurons.

Name	MNIST(CNN)	CIFAR-10	SVHN
Input	1x28	3x32x32	3x32x32
Conv-1	64x3x3	128x3x3	64x3x3
Conv-2	64x3x3	128x3x3	64x3x3
Pooling	2 x 2 Max Pooling		
Conv-3	128x3x3	256x3x3	128x3x3
Conv-4	128x3x3	256x3x3	128x3x3
Pooling	2 x 2 Max Pooling		
Conv-5	256x3x3	512x3x3	256x3x3
Conv-6	256x3x3	512x3x3	256x3x3
Pooling	2 x 2 Max Pooling		
FC-1	1024	1024	1024
FC-2	1024	1024	1024
FC-3	10	10	10

has a training set of 60K examples, and a testing set of 10K examples. SVHN is a real-world image dataset for street view house numbers, cropped to 32x32 color images, with 604K digits for training, 26K digits for testing. Both of these datasets classify digits ranging from 0 to 9. CIFAR-10 dataset consists of 60K 32x32 color images in 10 mutually exclusive classes (airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck), with $6,000$ images per class. There are 50K training images and 10K test images.

The convolutional neural networks we use has almost the same architecture as BNN [HCS$^+$16a]'s except for some small modification. The VGG [SZ14] network inspires us these architectures. It contains three fully-connected layers and six convolutional layers, in which the kernels for convolutional layers is 3 x 3. Please refer to Table 2.1 for the detailed numbers of parameters.

In each experiment, we split the dataset into three parts: 90% of the training set is used for training the network, the remaining 10% is used as a validation set. During the training, we use both the training loss on the training set and inference error-rate on the validation set as

Table 2.2. Classification Accuracy (Error Rate) Comparison on Different Datasets. BCNNw/SF1 stands for our training method 1; BCNNw/SF2 denotes for our training method 2.

Dataset	MNIST(CNN)	CIFAR-10	SVHN
No binarization (standard results)			
Maxout Networks [GWFM$^+$13]	0.94%	11.68%	2.47%
Binarized Network			
BCNN(BinaryNet) [HCS$^+$16a]	0.47%	11.40%	2.80%
Binarized Network with Separable Filters			
BCNNw/SF Method 1 (this work)	0.48%	14.12%	4.60%
BCNNw/SF Method 2 (this work)	0.56%	15.46%	4.18%

performance measurements. To evaluate the different trained models, we use the classification accuracy on the testing set as the evaluation protocol.

In order for all these benchmarks to remain challenging, We did not use any pre-processing, data-augmentation or unsupervised learning. We use a binarized hard tangent [HCS$^+$16a] function as the activation function. The ADAM adaptive learning rate method [KB14] is used while minimizing the square hinge loss with an exponentially decaying learning rate. We also apply batch normalization to our networks, with a mini-batch of size 100, 50 and 50 (separately for MNIST, CIFAR-10, and SVHN), to speed up the learning, and we scale the learning rate for each convolutional layer with a factor from Glorot's batch normalization [IS15]. We train our networks for 300 epochs on MNIST and CIFAR-10 datasets, and 200 epochs on SVHN datasets. The results are given in Section 2.5.2.

2.5.2 Benchmark Result

Figure 2.4 depicts the learning curves on the CIFAR-10 dataset. There exists certain accuracy degradation if we compare BCNN with our methods due to a more aggressive noise. By the end of the training phase, our method 1 yields an accuracy less than that of BCNN by roughly 2.72%, and method 2 reaches an even more inferior accuracy. For the sake of CIFAR-10's higher difficulty, the loss of accuracy meets our expectation. We will discuss in detail the benefit of using the exact gradient over the rank-1 approximation in next sub-section.

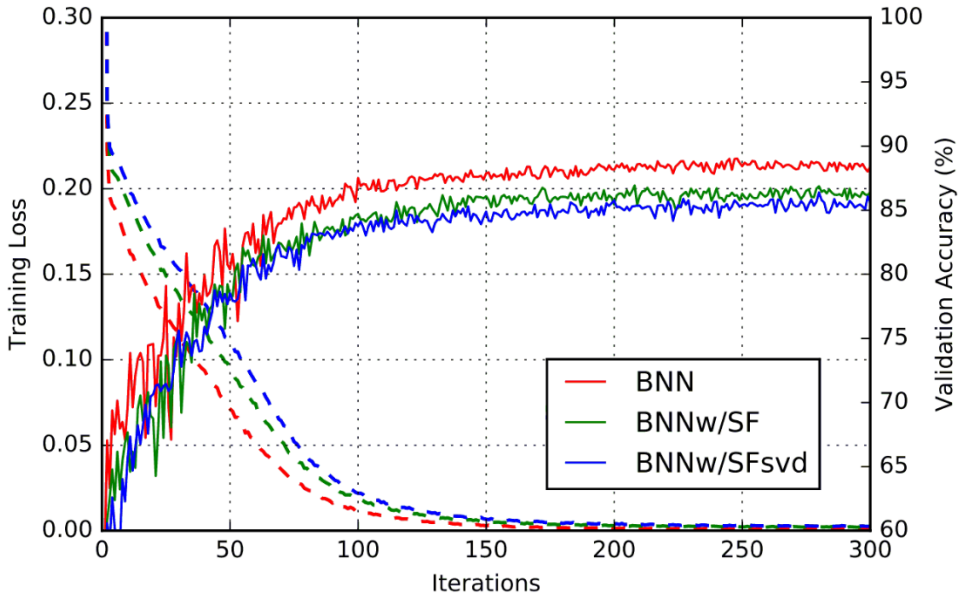

Figure 2.4. Learning Curves of BNN(red), BNNw/SF1(green) and BNNw/SF2(blue) on the CIFAR-10 dataset. The dashed lines represent the training costs(square hinge losses) and the solid lines the corresponding validation error rates.

Table 2.2 summarizes the experimental results in terms of error rate. Compared with BNN [HCS+16a], for the gray-scale manuscript number classification, both of our two training methods achieve accuracy close to that of the binarized convolutional neural networks. The difference is within 0.09%. It is noteworthy that our method 2 outperforms method 1 on SVHN by 0.42% error rate. For CIFAR-10 and SVHN, our methods are inferior to BCNN by a difference less than 2.72% because we limit choices of filters from 512 to 32, where the filter size is 3x3. Since the performance degradation on CIFAR-10 is the largest, we implement a hardware accelerator in FPGA to inspect at what extent of hardware complexity can be improved with the sacrifice of the 2.72% accuracy loss. Section 2.6 provides the details and a comparison with a BCNN accelerator to demonstrate the benefits of BCNNw/SF.

2.5.3 Scalability

We also explore different sizes of networks to improve the accuracy and exam the scalability of BCNNw/SF. Table 2.3 lists two additional larger models and an AlexNet-like

Table 2.3. The 1st column shows a deeper model with two extra convolutional layers, and the 2nd column shows a widened network with all numbers of kernels doubled. The 3rd column is inspired by AlexNet to include 3 sizes of filters.

Name	Deeper	Wider	AlexNet-like
Input	3x32x32	3x32x32	3x32x32
Conv-1	128x3x3	256x3x3	96x5x5
Conv-2	128x3x3	256x3x3	256x5x5
Pooling	2 x 2 Max Pooling		
Conv-3	256x3x3	512x3x3	512x3x3
Conv-4	256x3x3	512x3x3	512x3x3
Pooling	2 x 2 Max Pooling		
Conv-5	512x3x3	1024x3x3	256x3x3
Conv-6	512x3x3	1024x3x3	512x1x1
Pooling	2 x 2 Max Pooling		
Conv-7	512x3x3	-	-
Conv-8	512x3x3	-	-
Pooling	2x2 Max Pooling	-	-
FC-1	1024	1024	1024
FC-2	1024	1024	128
FC-3	10	10	10

model for CIFAR-10. The wider one stands for a model with all numbers of kernels doubled, and the deeper one is a network including two extra convolutional layers. Different from the models above, the AlexNet-like model includes three sizes of filters: 5-by-5, 3-by-3, and 1-by-1. Applying our rank-1 approximation on 5-by-5 filter, we can get 64% memory reduction. We train the three bigger networks with our method 1, and Figure 2.5 shows the learning curves of the two enlarged models for CIFAR-10. Since the number of trainable parameters has increased, it requires more epochs to travel in the hypothesis space and reach a local minimum. Therefore, we train these two bigger networks with 500 epochs and compare the results with BCNN(BinaryNet). As shown in Figure 2.5 the wider one (blue) starts with largest ripples yet catch up the same performance as BCNN(black) does around the 175th epoch.

Table 2.4 lists the results on CIFAR-10 of the three bigger models as well as the CIFAR-10 results in Table 2.2. The performance improvement of the deeper networks is very scarce since the feature maps experience the extra destructive max pooling layer as shown in Table 2.3,

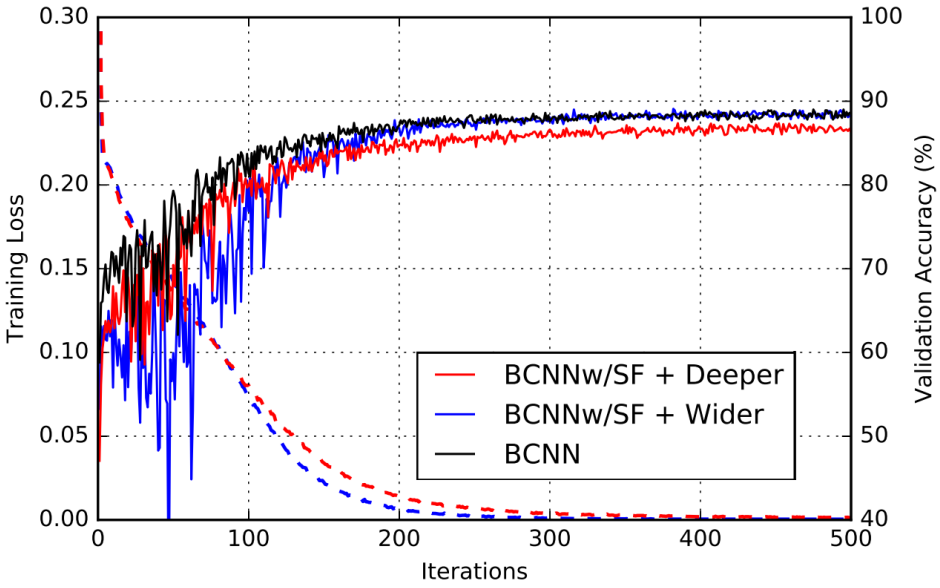

Figure 2.5. Learning Curves of the larger BCNNw/SF models: deeper Network(red), BCNNw/SF with wider Network(blue) and the original BCNN on the CIFAR-10 dataset. The dotted lines represent the training costs(square hinge losses) and the solid lines the corresponding validation error rates.

which reduces the size of the first fully-connected layer, FC-1, and hence suppresses the improvement. The wider network achieves 11.68%, which is very close to the performance of BCNN(BinaryNet). The AlexNet-like model demonstrates that a model with 5-by-5 filters sacrifices more accuracy to provide higher memory reduction. In summary, the accuracy degradation of BCNNw/SF has been compensated by enlarging the sizes of the networks.

2.5.4 Discussion

In this section, we use the experimental results on CIFAR-10 as an example of a detailed analysis. We unpack the trained rank-1 filters and learning curves to gain a better understanding of the mechanism of BCNNw/SF.

Figure 2.6 lists all the 32 rank-1 filters and their frequency on CIFAR-10. Although certain filters are rarely used, there is no filter forsaken. In Figure 2.6 we can learn that the all-positive and all-negative filters are trained most frequently, and these two filters render the

Table 2.4. Classification Accuracy (Error Rate) of the three larger models.

Dataset	CIFAR-10
BCNN(BinaryNet) [HCS$^+$16a]	11.40%
Binarized Network with Separable Filters (this work)	
BCNNw/SF Method 1	14.12%
BCNNw/SF Method 1 depper	14.11%
BCNNw/SF Method 1 wider	**11.68%**
BCNNw/SF Method 1 AlexNet-like	15.1%

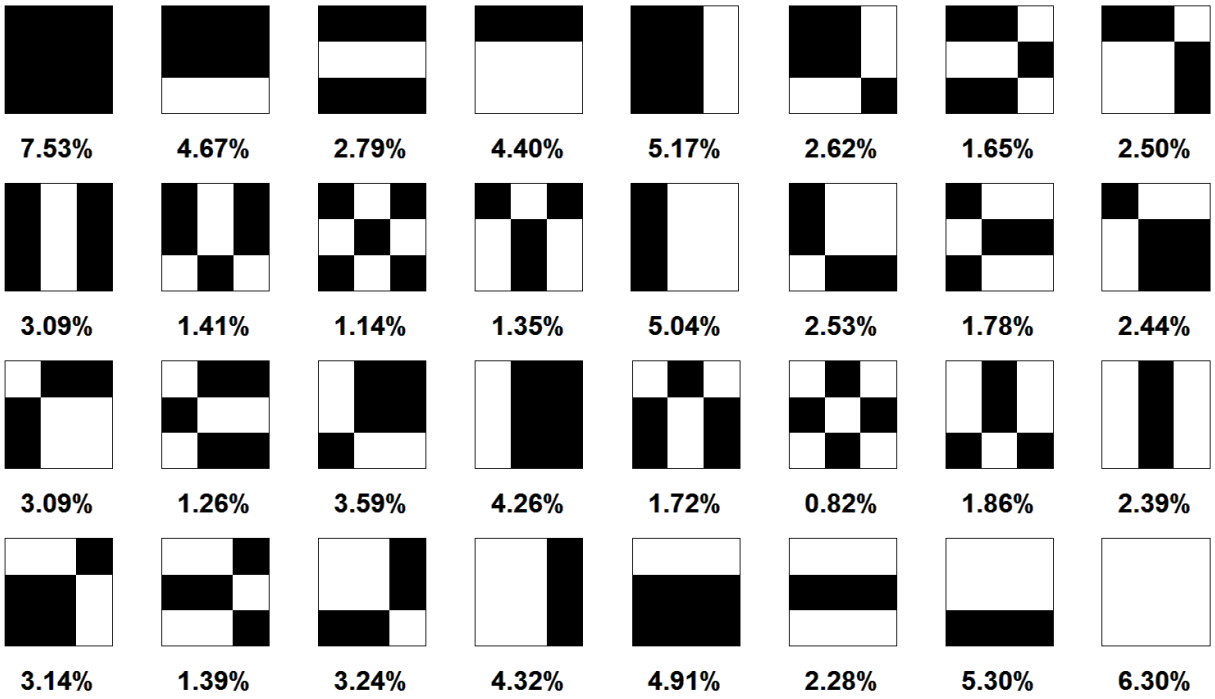

Figure 2.6. Separable Filters and Frequencies used in CIFAR Model

convolution to running-sum calculation with a sliding window. As mentioned in Section 2.3.1, through the summation of separated convolution from a preceding layer, we can achieve the tangled linear combinations, which are essential to BCNNw/SF.

Unknowing the spectrum of the ripple, we apply Savitsky-Golay filter [SG64] to obtain the baseline of validation accuracy and, thereby, subtract the original accuracy with the baseline to get the ripple. The window width of the Savitsky-Golay filter is 51, and we use a quadratic equation to fit the original learning curve. All ripples are quantized into 100 categories for the statistic analysis.

Table 2.5. The statistics of the ripples in terms of percent of error rate.

Statistics	mean	std	max
BCNN(BinaryNet) [HCS+16a]	0.052	1.213	5.09
BCNNw/SF Method 1	0.055	1.059	4.465
BCNNw/SF Method 2	0.035	0.723	3.622

Table 2.5 compares our method 1 and methods 2 with BCNN. All three statistic advantages of method 2 are reduced. The analytic gradient over the rank-1 approximation stabilizes the descending trajectory with more accurate gradient calculation. Both BCNN and our method 1 rely on the gradient w.r.t. binarized filters to update all parameters due to the lack of analytic gradient w.r.t. real-valued filters. However, it is also the rigorous gradient that limits the possibility to escape a local minimum on the error surface. As we can see in Table 2.2, the results of our method 1 are closer to that of BCNN. We use the trained binarized separable filter from our method 1 to implement an FPGA accelerator for CIFAR-10 in the following section.

2.6 FPGA Accelerator

2.6.1 Platform and Implementation

To quantify the benefits that BCNNw/SF can achieve for hardware BCNN accelerators, we created an FPGA accelerator for the six convolutional layers of the Courbariaux's CIFAR-10 network. Our accelerator is built from the open-source FPGA implementation in [ZSZ+17]. The dense layers were excluded as they are not affected by our technique. As BCNNw/SF is ideal for small, low-power platforms, we targeted a Zedboard with a Xilinx XC7Z020 FPGA and an embedded ARM processor. This is a much smaller FPGA device compared to existing CNN FPGA accelerators [QWY+16, SCD+16]. We write our design in C++ and use Xilinx's SDSoC tool to generate Verilog through high-level synthesis. We implement both BCNN and BCNNw/SF and examine the performance and resource usage of the accelerator with and without separable filters.

Our accelerator is designed to be small and resource-efficient; it classifies a single

image at a time and executes each layer sequentially. The accelerator contains two primary compute complexes: `Conv1` computes the first (non-binary) convolutional layer, and `Conv2-5` is configurable to compute any of the binary convolutional layers. Other elements include hardware to perform pooling and batch normalization, as well as on-chip RAMs to store the feature maps and weights. Computation with the accelerator proceeds as follows. Initially, all input images and layer weights are stored in off-chip memory accessible from both CPU and FPGA. The FPGA loads an image into local RAM, then it loads the layer's weights and performs the computation for each layer. Larger layers require several accelerator calls due to limited on-chip weight storage. Intermediate feature maps are fully stored on-chip. After completing the convolutional layers we write the feature maps back to main memory and the CPU computes the dense layers.

We kept the BCNN and BCNNw/SF implementations as similar as possible, with the main difference being the convolution logic and storage of the weights. For BCNN, each output pixel requires $3 \times 3 = 9$ MAC operations to compute. For BCNNw/SF we can apply a 3x1 vertical followed by a 1x3 horizontal convolution, a total of 6 MACs. As the MACs are implemented by XORs and an adder tree, BCNNw/SF can potentially save resource.

In terms of storage, BCNN requires the 9 bits to store each filter. Naively, BCNNw/SF requires 6 bits, as each filter is represented as two 3-bit vectors. However, recall we only use rank-1 filters — Eq. 2.3 shows that the number of unique 3×3 is 32, meaning we can encode them losslessly with only 5 bits. A small decoder in the design is used to map the 5-bit encodings into 6-bit filters.

2.6.2 Results and Discussion

Table 2.6 compares the execution time and resource usage of the two FPGA implementations. Resource numbers are reported post place and route, and runtime is wall clock measured on a real Zedboard. We exclude the time taken to transfer the final feature maps from FPGA to main memory, as it is equal between the two networks; transfer time for the initial image and weights are included.

Table 2.6. Comparison of performance and resource usage between BCNN and BCNNw/SF FPGA implementations. Runtime is for a single image, averaged over 10000 samples.

	BCNN	BCNNw/SF (this work)	δ
Conv layer runtime (ms)	0.949	0.652	-31.3%
LUT	35255	36384	+3.2%
FF	41418	41054	-1.0%
Block RAM	94	78	-17.0%
DSP	8	8	0.0%

Our experimental results show that BCNNw/SF achieves runtime reduction of 31% over BCNN, which equates to a 1.46X speedup. This is due mostly to the reduction of memory transfer time of the compressed weight filters. For similar reasons, BCNNw/SF is able to save 17% of the total block RAM (RAMs are used for both features and weights). Look-up table (LUT) counts have increased slightly, due most likely to the additional logic needed to map the 5-bit encodings to actual filters. Overall, BCNNw/SF realizes significant improvements to performance and memory requirement with minimal logic overhead.

2.7 Conclusion and Future Work

In this chapter, we proposed the binarized convolutional neural network with Separable Filters (BCNNw/SF) to make BCNN more hardware-friendly. Through binarized rank-1 approximation, 2D filters are separated into two vectors, which reduce memory footprint and the number of logic operations. We have implemented two methods to train BCNNw/SF with Theano and verified our methods with various CNN architectures on a suite of realistic image datasets. The first method relies on batch normalization to regularize noise, making it simpler and faster to train, while the second method uses gradient over SVD to make the learning curve more smooth and potentially achieves better accuracy. We also implement an accelerator for the inference of a CIFAR-10 network on an FPGA platform. With separable filters, the total memory footprint is reduced by 17.0% and the performance of the convolution layers is improved by 1.46X compared to baseline BCNN.

Integrating probabilistic methods [SHM14b] to reduce the training time and exploring more elegant structures of networks [SIV16] will be a promising direction for future works.

Chapter 2 contains reprints of Jeng-Hau Lin, Tianwei Xing, Ritchie Zhao, Zhiru Zhang, Mani Srivastava, Zhuowen Tu, and Rajesh K. Gupta."Binarized Convolutional Neural Networks with Separable Filters for Efficient Hardware Acceleration." *I*n Computer Vision and Pattern Recognition Workshop (CVPRW). IEEE, 2017. This dissertation author is the primary author of this paper.

Chapter 3

Local Binary Pattern Networks

Emerging edge devices such as sensor nodes are increasingly being tasked with non-trivial tasks related to sensor data processing and even application-level inferences from this sensor data. These devices are, however, extraordinarily resource constrained in terms of CPU power (often Cortex M0-3 class CPUs), available memory (in few KB to MBytes), and energy. Under these constraints, we explore a novel strategy to carry out character recognition tasks using local binary pattern networks, or LBPNet, that can learn and perform bit-wise operations in an end-to-end fashion. LBPNet has its advantage for characters for which the features are composed of structured strokes and distinct outlines. LBPNet uses local binary comparisons and random projections in place of conventional convolution (or approximation of convolution) operations, providing an important means to improve memory efficiency as well as inference speed. We evaluate LBPNet on a number of character recognition benchmark datasets, as well as several object classification datasets, and demonstrate its effectiveness and efficiency.

3.1 Introduction

Rigid and deformable objects like optical characters are interesting patterns to study in computer vision and machine learning. In particular, instances found in the wild – handwriting, street signs, and house addresses (as shown in Figure 3.1) – are of high importance to the emerging mobile edge systems such as augmented reality glasses or delivery UAVs. The recent

Figure 3.1. Examples from character recognition datasets.

innovations in Convolutional Neural Networks (CNN) [LBD+89b] have achieved state-of-the-art performance on these OCR tasks [YWZL13]. As deep learning models evolve and take on increasingly complex pattern recognition tasks, however, they demand tremendous computational resources with correspondingly higher performance machines and accelerators that continue to be fielded by system designers. This can limit their use to only applications that can afford the energy and/or cost of such systems. By contrast, the universe of embedded devices, especially when used as intelligent edge devices in the emerging distributed systems, presents a higher range of potential applications from augmented reality systems to smart city systems. As a result, seeking for memory and computationally efficient deep learning methods becomes crucial to the continued proliferation of machine learning capabilities to new platforms and systems, especially mobile sensing devices with ultra-small resource footprints.

Various methods have been proposed to perform network pruning [LDS90, GYC16], compression [HMD15, IHM+16], or sparsification [LWF+15], in order to reduce the model complexity. Impressive results have also been achieved lately by binarizing selected operations

in CNNs [CBD15, HCS+16b, RORF16b]. At their core, these efforts seek to approximate the internal computational granularity of CNNs, from network structures to variable precisions, while still keeping the underlying convolution operation exact or approximate. However, the nature of character images has not been fully taken advantage yet.

In this work, we propose a light-weight and compact deep learning approach, LBPNet, that can leverage the nature of character images. Particularly, we focus on the character classification task and explore an alternative to convolutional operations – the local binary patterns (LBP), which employs a number of *predefined* sampling points that are mostly on the perimeter of a circle, compares them with the pixel value at the center using logical operations, and yields an ordered array of logical outputs in order to learn the patterns in an image. This operation makes LBP particularly suitable for recognizing characters that consist of discriminative outlines and structured strokes. We note that our work has roots in research before the current generation of deep learning (DL) methods, namely, the adoption of LBP [OPH96]. While LBP gives rise to a surprisingly rich representation [WHY09] of the underlying image patterns and has shown to be complementary to the SIFT-kind features [Low04], it has been under-explored in the DL research community, where the feature learning primarily refers to the CNN features in a hierarchy [KSH12, HZRS15].

Multiple innovations and important properties within LBPNet distinguish it from previous attempts:

- **Multiplication and accumulation free.** We employ the LBP that involves only logic operations to extract features of images, which is in stark contrast to previous attempts trying to either directly binarize the CNN operations [HCS+16b, RORF16b] or approximate LBP with convolution operations [JXBS17]— convolution requires expensive, power-hungry multipliers and slow accumulation operations.
- **Learnable LBP kernel.** The sampling points in a traditional LBP kernel are at fixed locations upon initialization, due to the lack of a sustainable mechanism to deform the sampling patterns

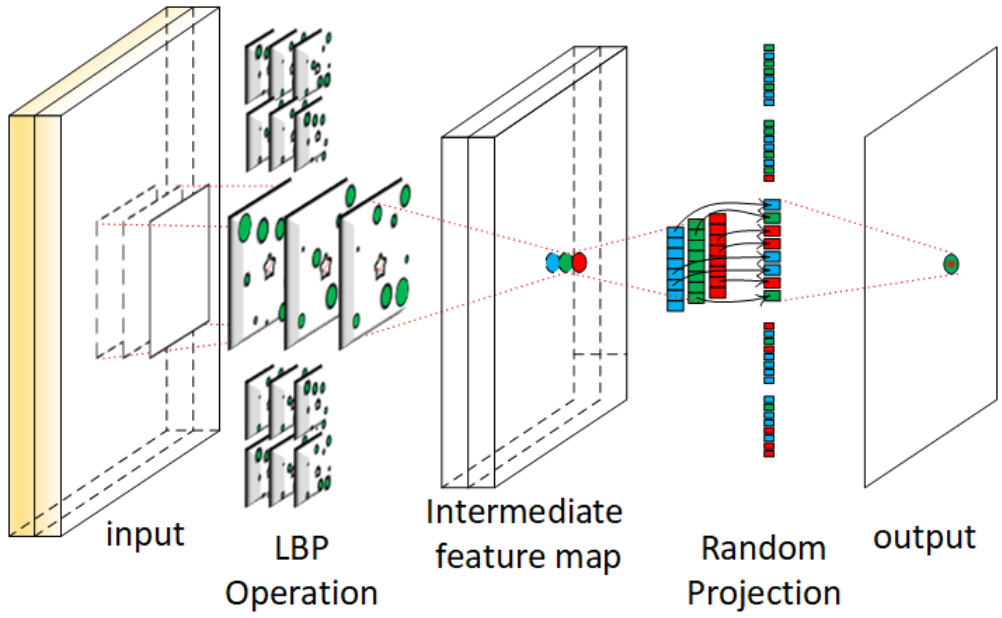

Figure 3.2. The LBPNet micro-architecture: The LBP operation generates feature maps via comparison and bit allocation, and the random projection fuses the intermediate channels to final output.

for feature extraction, and they only learned the linear combination of the resultant generated from the fixed patterns. Instead, we design a differentiable function to learn the sampling patterns and prove the effectiveness of LBPNet's learning via the optical flow theory and gradient descent.

- **Compact model size.** CNN-based models are stored in dense matrices which usually takes mega-byte storage space, while LBPNet learns discrete and sparse patterns. Without further encoding or compression, the typical sizes of the kernels in LBPNets are on the kilo-byte level, yielding $1,000$X model reduction in parameter size.
- **Fast inference speed.** The accumulation in convolution impedes CNN's inference speed. Even though the basic linear algebra subprogram (BLAS) library utilizes techniques such as loop unrolling and tiling, there still exists the accumulation of small accumulating blocks. However, LBPNet's memory indexing, comparison, and bit-allocation have no data-dependency on the neighboring computing elements, and can thus be parallelized. This significantly boosts the

inference speed for LBPNet on common single-instruction-multiple-data (SIMD) architectural systems like GPUs or pipeline-parallel systems like FPGAs or ASICs.

- **Optimized backward propagation and end-to-end learning.** The backprop of LBPNet follows the framework of the state-of-the-art fastest implementation of Conv Layer, SpatialConvolutionMM [CPS06]. Owing to the sparse sampling patterns in LBPNet, we can replace part of the gradient computation with much simpler CUDA-C routines.

3.2 Related Work

Related work regarding model reduction of CNN falls within four primary categories.

Character Recognition. In addition to the CNN-based solutions to character recognition like BNN [HCS$^+$16b], the random forest [YBL14, YBSL14] was prevailing as well. However, it usually required one or more techniques such as feature extraction, clustering, or error correction codes to improve recognition accuracy. Our method, instead, provides a compact end-to-end and computationally efficient solution to character recognition.

Active or Deformable Convolution. Among the notable line of recent work that learned local patterns were active convolution [JK17] and deformable convolution [DQX$^+$17]. While they indeed learned data dependent convolution kernels, which still heavily relied on multiplication and addition operations, they did not explicitly seek to improve the network efficiency. By contrast, our LBP kernels learn the best location for the sampling points in an end-to-end fashion via simple yet effective logic operations, without the need for multiplication and addition operations required in convolutions.

Binarization for CNN. Binarizing CNNs to reduce the model size has been an active research direction [CBD15, HCS$^+$16b, RORF16b]. Through binarizing the weights and/or activations, these works replaced multiplications with logic operations thus reducing the model size. However, non-binary operations such as batch normalization in BNN [HCS$^+$16b] and scaling and shifting in XOR-Net [RORF16b] still required floating-point operations. Both BNN

and XNOR-Net can be considered as the discretization of real-valued CNNs, and thus the two works were still fundamentally based on spatial convolution — we instead leverage the less computationally hungry LBP that employs logic operations.

CNN Approximation for LBP Operation. Recent work on local binary convolutional neural networks (LBCNN) [JXBS17] took an opposite direction to BNN [HCS+16b]. LBCNN utilized the difference between pixel values together with a ReLU layer to simulate the LBP operations. During training, the sparse binarized difference filters were fixed, and only the successive 1-by-1 convolution kernel, serving as a channel fusion mechanism, and the parameters in the batch normalization layer (BatchNorm layer), were learned. However, the feature maps of LBCNN were still made up of floating-point numbers, and this resulted in significantly increased model complexity as we shall show later in Table 3.3 and 3.4.

Although LBCNN have achieved some degree of model compression on OCR tasks, it still relied heavily on using batch normalization layers, which must be performed in floating numbers for the linear transformation. While implementing hardware accelerators, people have found that the four BatchNorm parameters at most can be quantized from 32-bit floating numbers to 16-bit fixed numbers without significant accuracy loss [ZSZ+17]. Because the size and computation of a batch normalization layer were linear in the size of the feature maps, LBCNN was still too cumbersome for IoT devices built with limited memory and compute resources. Even for binarized neural networks, the convolutional kernels and batch normalization layer parameters were still so large that an off-chip DRAM and on-chip buffering mechanism are required [ZSZ+17, UFG+17]. Therefore, we propose LBPNet to directly learn the sparse and discrete LBP kernels, which are typically as tiny as several kilobytes.

3.3 Local Binary Pattern Network

In LBPNet, the forward propagation is composed of two key procedures: the LBP operation and channel fusion. In this section, we elaborate on them, describe the carefully designed

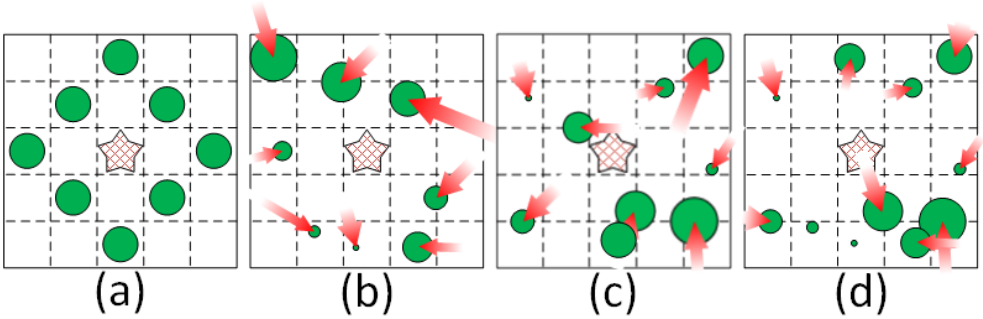

Figure 3.3. The traditional and our local binary patterns: (a) A traditional local binary pattern. (b)-(d) Our learnable local binary patterns. The red arrows denote pushing forces during training.

network structure of LBPNet, and present a back-of-the-envelope calculation of hardware gains of LBPNet.

3.3.1 LBP Kernel and Operation

Figure 3.3 (a) shows a traditional LBP with a fixed structure: there are eight sampling points (the green circles) surrounding a pivot point (the meshed star) at the center of the kernel. The pixel at each of the sampling points will be compared with the one at the center, and if the sampled pixel value is larger than that at the center, we output a bit "1"; otherwise, the output is set to "0". These eight 1-bit comparison outcomes are assigned to a bit array according to predefined order, either clockwise or counter-clockwise. The bit array is interpreted as an integer and can be further used with learning methods such as support vector machine, histogram analysis, multi-layer perceptrons, etc.

In LBPNet, we make the fixed sampling points in a traditional LBP kernel *adaptive* and *learnable*, as shown in Figure 3.3(b)-(d): The learnable patterns are first initialized at random locations within a given area following a uniform distribution and then pushed to better locations to minimize the classification error using our proposed mechanism. The sizes of the sampling points (in green) correspond to the bit positions of the comparison outcomes in the output bit array – a larger circle corresponds to a more significant bit. The red arrows represent the driving forces that can push the sampling points, and we defer the details of the deformation mechanism

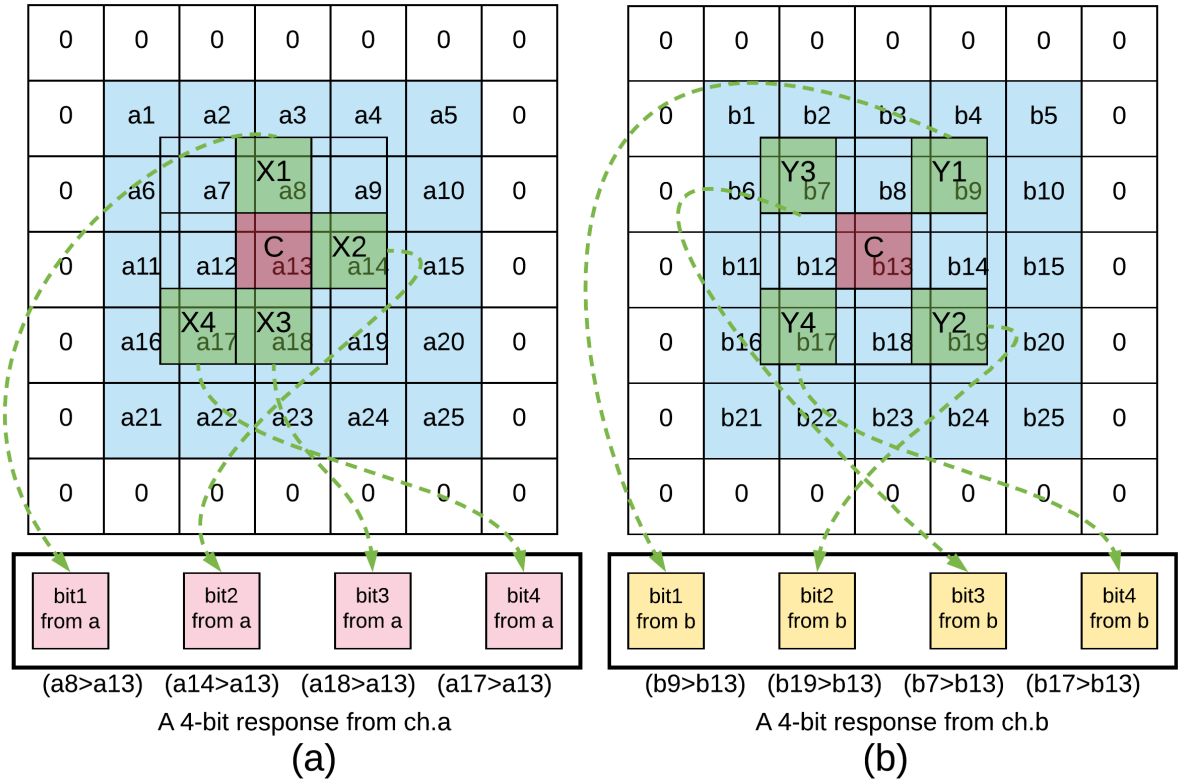

Figure 3.4. An example of the LBP operation on two input channels – ch.a and ch.b: There are four sampling points in each 3-by-3 LBP kernel, and each sampling point produces a logic bit which is assigned to a certain position (marked with arrows) in the output array (shown at the bottom in pink and yellow).

to the next section. The model size of an LBPNet is tiny compared with CNN because the learnable patterns in LBPNet are comprised of the sparse and discrete sampling indices within the window. Finally, multiple patterns in different channels form a kernel of LBPNet.

Figure 3.4 shows a snapshot of the LBP operation. Given two input channels, ch.a and ch.b, we perform the LBP operation on each channel with different 3-by-3 kernel patterns. We only put four sampling points, as an example, in each kernel to avoid cluttered figures, and the two 4-bit binary response arrays are shown at the bottom (in pink and yellow). For clarity, we use green dashed arrows to mark the corresponding pixels for the resulting bits and list the comparison equation under each bit. In LBPNet, we slide the LBP kernel over an entire image, as convolution is done in CNN, to produce a complete feature map, and we perform the LBP

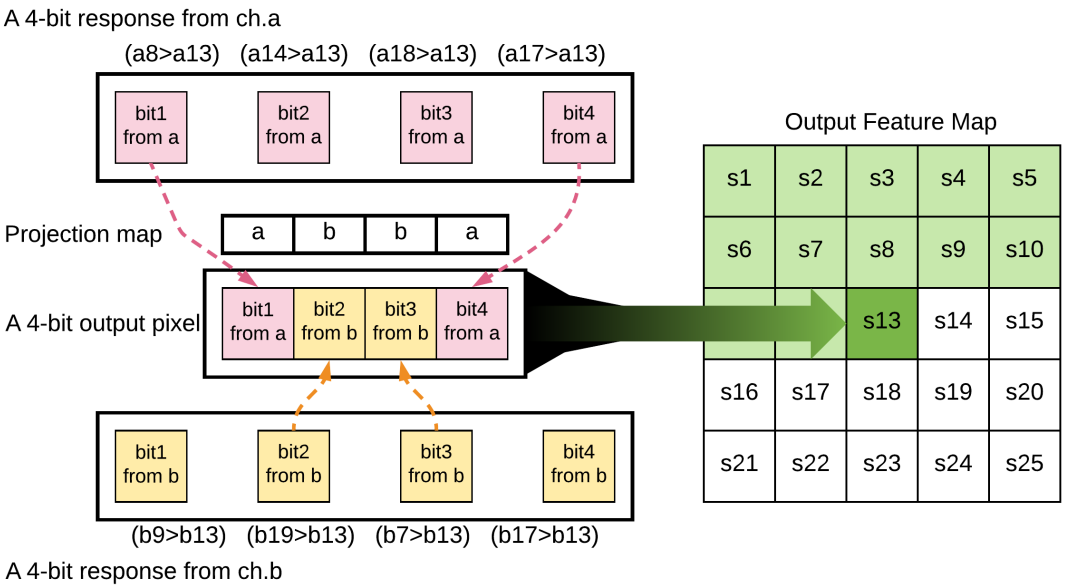

Figure 3.5. An example of LBP channel fusion. The two 4-bit responses in Figure 3.4 are fused and assigned to pixel s13 in the output feature map.

operation on each input channel of the image.

3.3.2 Channel Fusion with Random Projection

With the LBP operation, the number of resulting channels might grow exponentially: suppose we have N LBP layers, and each uses K kernels, the number of the output channels in the last layer will be $O(K^N)$. Akin to channel-wise addition in normal convolutional operation, we need a channel fusion mechanism to avoid the potential explosion. We resort to random projection [BM01] as a dimension-reducing and distance-preserving step to select output bits among intermediate channels for the concerned output channel, as shown in Figure 3.5. The random projection is implemented with a predefined mapping table for each output channel. By way of explanation, the mapping between the bit in the output pixel and the channel of the image is fixed upon initialization, and all output pixels in the same output channel follow the same mapping. For example, in Figure 3.5, the two pink bits in the output pixel come from ch.a while the two yellow bits come from ch.b. As a result, only the most and least significant bits on ch.a

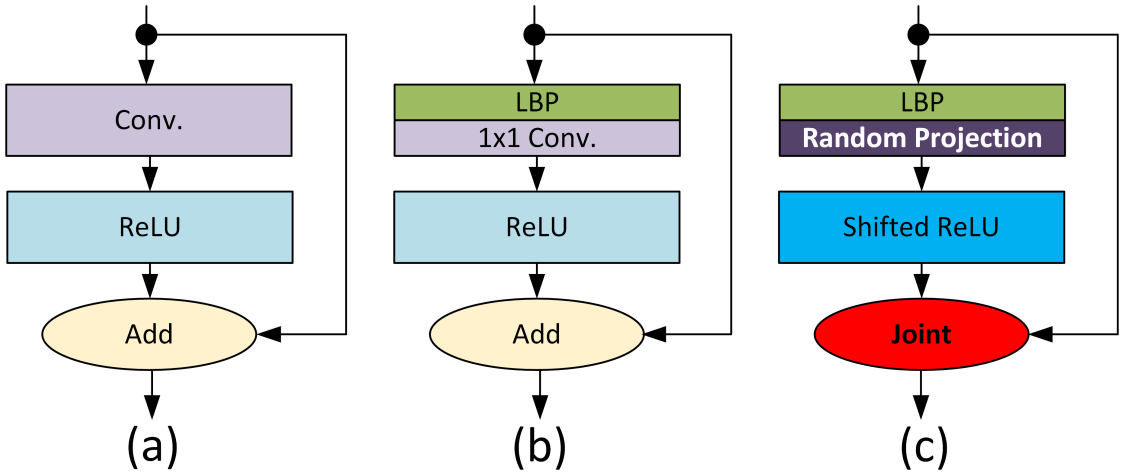

Figure 3.6. Building block architecture: (a) the well-known building block of residual networks. (b) The transition-type building block uses a 1-by-1 convolutional layer as an alternate channel fusion for the preceding LBP layer; this structure is considered as a baseline in evaluation. (c) The multiplication and accumulation (MAC) free building block of our LBPNet.

and the two middle bits on the ch.b need to be computed. In other words, for an n-bit output pixel, the random projection will select only n channels to make n comparisons, eliminating the need of comparing all sampling points with the pivots. The fusion step essentially makes the number of comparisons independent from the number of input channels K_{in} and reduces the memory complexity from $O(K_{in}K_{out})$ to $O(nK_{out})$, where K_{out} is the number of output channels. Although K_{in} is removed from the memory complexity, it still affects the algorithm because a larger K_{in} will result in more variations–there would be $\binom{K_{in}}{n}$ combinations–for the projection.

3.3.3 Network Structures of LBPNet

The network structure of LBPNet must be carefully designed. Owing to the binary nature of the comparison, the outcome of an LBP layer is very similar to the result of difference filtering. In other words, our LBP layer is good at extracting high-frequency components in the spatial domain but relatively weak at understanding low-frequency components. Therefore, we use a residual-like structure to compensate for this weakness of LBPNet. Figure 3.6 shows three kinds of residual-net-like building blocks. Figure 3.6 (a) is the typical building block for

Table 3.1. The number of logic gates for arithmetic units. Energy usage for technology node: 45nm.

Device	#bits	#gates	Energy (J)
Adder	4	20	$\leq$ 3E-14
	32	160	9E-13
Multiplier	32	$\geq$144	3.7E-12
Comparator	4	11	$\leq$ 3E-14

residual networks, where the convolutional kernels learn to obtain the residual of the output after the addition. Similarly, in LBPNet, because the pixels in the LBP output feature maps are always positive, we use a shifted rectified linear layer (shifted-ReLU) accordingly to increase nonlinearities, as shown in Figure 3.6 (c). The shifted-ReLU truncates any magnitudes below the half of the maximum of the LBP output. More specifically, if a pattern has n sampling points, the shifted-ReLU is defined as

$$f(x) = \begin{cases} x, & x > 2^{n-1} - 1 \\ 2^{n-1} - 1, & \text{otherwise.} \end{cases}$$

As mentioned earlier, the low-frequency components evanesce as the information passes through several LBP layers. To preserve the low-frequency components while making the basic block multiplication-and-accumulation free (MAC-free), we introduce a joint operation, which cascades the input tensor of the block and the output tensor of the shifted-ReLU along the channel dimension. The number of channels is under controlled since the increasing trend is linear in the number of input channels.

Throughout the forward propagation, there are no multiplication or addition operations. Only comparison and memory access are used. Therefore, the design of LBPNets is efficient with regard to both software and hardware.

3.3.4 Hardware Benefits

LBPNet avoids the computation-heavy convolution operations and thus saves hardware costs. Table 3.1 lists the reference numbers of logic gates of the concerned arithmetic units. A ripple-carry full-adder requires 5 gates for each bit. A 32-bit multiplier includes a data-path logic and a control logic. Because there are too many feasible implementations of the control logic circuits, we conservatively use an open range to give a sense about the hardware expense. The comparison can be implemented on a pure combinational logic circuit comprised of 11 gates, which also means that only the infinitesimal internal gate delays dominate the computation latency. The comparison is not only cheap regarding its gate count but also fast due to the absence of sequential logic internally. Slight difference in numbers of logic gates may apply if different synthesis tools or manufacturers are chosen. With the capability of an LBP layer as strong as a convolutional layer concerning classification accuracy, replacing the convolution operations with comparison gives us a 27X saving of hardware cost. Another important benefit is energy savings. The energy demand for each arithmetic device has been shown in [Hor14]. If we replace all convolution operations with comparisons, the energy consumption is reduced by 153X. Moreover, the core of LBPNet is composed of bit-shifting and bitwise-OR, and both of them do not have the concurrent accessing issue as in convolution's accumulation process. If we implement an LBPNet hardware accelerator, no matter on FPGA or ASIC flow, the absence of the concurrent issue will guarantee a speedup over CNN hardware accelerator. For more justification, please refer to the forward algorithm in the appendices.

3.4 Backward Propagation of LBPNet

There exist two problems in the backward pass of LBPNet that prevent it from being trained with ordinary gradient descent methods:

1) **Non-differentiability of comparison**. Inherently, LBPNet requires binary comparison of activations; however, this is not differentiable.

2) **The lack of a driving force for parameter updates**. The learnable parameters (excluding those present at the MLP stage) in an LBPNet are the sampling points' locations within each pattern. A gradient (or approximation) between the rest of the network and a given configuration of sampling points was not clearly defined.

3.4.1 Differentiability of comparison

The first problem can be solved if we model the comparison operation with a shifted and scaled hyperbolic tangent function as

$$I_{lbp} > I_{pivot} \overset{modeled}{\rightarrow} \frac{1}{2}\left(\tanh\left(\frac{I_{lbp} - I_{pivot}}{\alpha}\right) + 1\right), \tag{3.1}$$

where α is a scaling parameter to accommodate the number of sampling points from a previous LBP layer, I_{lbp} is the sampled pixel in a learnable LBP kernel, and I_{pivot} is the sampled pixel at the pivot. We provide a sensitivity analysis of α w.r.t. classification accuracy in the appendices. The hyperbolic tangent function is differentiable and has a simple closed-form for the implementation.

3.4.2 Deformation with Optical Flow

The aperture problem within the optical flow theory provides a sustainable mechanism that can push sampling points to extract common features for classification. In this section, we provide a brief proof of the effectiveness of LBPNet regarding feature extraction.

The *optical flow equation* [BFB94] states:

$$\frac{\partial I}{\partial x}V_x + \frac{\partial I}{\partial y}V_y = -\frac{\partial I}{\partial t}, \tag{3.2}$$

where the left-hand side of the optical flow equation can be interpreted as a dot-product of the image gradient $(\frac{\partial I}{\partial x}\hat{x} + \frac{\partial I}{\partial y}\hat{y})$ and optical flow $(V_x\hat{x} + V_y\hat{y})$, and this product equals the negative derivative of luminance versus time across different images, where $\hat{x}$ and $\hat{y}$ denote the two orthogonal unit vectors on the 2-D coordinate, and the infinitesimal time difference ∂t can be

controlled to be a constant.

In the *Lucas-Kanade method* [LK81], the optical flow is constrained to be constant in a neighborhood around each point in the image. Therefore, the optical flow equation can be rewritten as

$$\mathbf{Av} = \mathbf{b}, \tag{3.3}$$

where $\mathbf{A} = \begin{bmatrix} I_{x_1} & I_{y_1} \\ I_{x_2} & I_{y_2} \\ \vdots & \vdots \\ I_{x_m} & I_{y_m} \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} -I_{t_1} \\ -I_{t_2} \\ \vdots \\ -I_{t_m} \end{bmatrix}$, $\mathbf{v} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$, $I_{x_i} = \frac{\partial I[i]}{\partial x}$, $I_{y_i} = \frac{\partial I[i]}{\partial y}$, $I_{t_i} = \frac{\partial I[i]}{\partial t}$, and m is the number of sampled pixels. The unknown optical flow vector $\mathbf{v}$ can, therefore, be solved since the number of equations depends on the number of pixels sampled, which can be designed to make the equation over-determined.

Applying the singular value decomposition (SVD) to the image gradient matrix $\mathbf{A}$ in Eq. (3.3) and move all three decomposition matrices to the right-hand side (RHS), we get the optical flow vector:

$$\mathbf{v} = \mathbf{V}\mathbf{D}^{-1}\mathbf{U}^{\top}\mathbf{b}, \tag{3.4}$$

where $\mathbf{U}$ and $\mathbf{V}$ are the left and right singular matrices which comprise orthonormal column vectors and possess the property of $\mathbf{U}^{\top}\mathbf{U} = I$ and $\mathbf{V}^{\top}\mathbf{V} = I$, and $\mathbf{D}$ is a diagonal matrix containing the singular values on its diagonal trace. $\mathbf{V}\mathbf{D}^{-1}\mathbf{U}^{\top}$ forms a left generalized inverse of $\mathbf{A}$.

We now show how this solution to the optical flow problem can provide useful gradient signal to the sampling points of an LBP pattern. Applying the chain rule within backpropagation to the sampling points (please refer to the appendices for more details of LBPNet's chain rule equations.):

$$\mathbf{g} = \mathbf{k}\mathbf{A}, \tag{3.5}$$

where $\mathbf{k} = \left[g_{o_1}\frac{\partial Fm_1}{\partial I_{lbp_1}}, g_{o_2}\frac{\partial Fm_2}{\partial I_{lbp_2}}, \ldots, g_{o_m}\frac{\partial Fm_m}{\partial I_{lbp_m}} \right]$, $\mathbf{g} = \left[\frac{\partial loss}{\partial x}, \frac{\partial loss}{\partial y} \right]$, and $\mathbf{A}$ is the image gradient matrix in Eq. (3.3), g_o is the error propogated from the succeeding layer, Fm is the output

feature map, $\frac{\partial Fm_i}{\partial I_{lbp_i}} = \frac{1}{\alpha}\left[1 - \tanh^2\left(\frac{I_{lbp_i} - I_{pivot_i}}{\alpha}\right)\right]$, and (I_{lbp_i}, I_{pivot_i}) is a pair of sampled pixels for comparison.

With the gradient of loss and the optical flow vector, we can derive the relation between gradient descent and the minimization of pixel difference as follows.

Multiply Eq. (3.5) to Eq. (3.4) from the left to get Eq. 3.6.

$$\mathbf{gv} = \mathbf{kUU}^\top \mathbf{b} \tag{3.6}$$

Please note that $\mathbf{UU}^\top = I$ only when A is invertible.

Eq. 3.6 can be interpreted as $\mathbf{gv} = \mathbf{k}'\mathbf{b}$, where $\mathbf{k}'$ is a transformed error vector. When the gradient descent minimizes the loss to a local minimum on the error surface, the gradient of loss w.r.t. positions $\mathbf{g}$ will converge be minimized presumably. Thereby the LHS of Eq. 3.6 will reduced, and the inner product of $\mathbf{k}'$ and the temporal difference $\mathbf{b}$ decreases. LBPNet, therefore, senses weaker and weaker differences between images.

3.4.3 Implementation

None of the existing deep learning (DL) libraries can be used to implement LBPNet because the logical operation such as comparison and bit-allocation are radically different from the arithmetic ones, and the deformation of sampling patterns violates the regularity on which conventional DL libraries rely. We, hence, directly use BLAS library to deliver a custom GPU kernel in order to provide a high-level interface for conventional DL libraries to integrate with the fundamental LBPNet operations.

We adopt the framework of convolution in Torch, SpatialConvolutionMM [CPS06], in order to trade memory redundancy via building Toeplitz matrices for speed-ups and leverage the GPU supported primitive functions, e.g., im2col, col2im, GEMM, and GEMV. We refer readers to the appendices for the detailed forward and backward propagation algorithms.

Table 3.2. Details of the datasets used in our experiments.

	#Class	#Example	State-of-the-Art error rate
DHCD	46	46x2,000	1.53% [APG15]
ICDAR-DIGITS	10	988	-
ICDAR-UpperCase	26	5,288	10% [WWCN12]
ICDAR-LowerCase	26	5,453	-
Chars74K-EnglishImg	62	7,705	52.91% [DCBV09]
Chars74K-EnglishHnd	62	3,410	23.33% [KGV17]
Chars74K-EnglishFnt	62	62,992	30.29% [DCBV09]

3.5 Evaluation

We conduct a series of experiments on five datasets – MNIST, SVHN, DHCD, ICDAR2005, and Chars74K – to demonstrate the capability of LBPNet. Some example images in these character datasets are shown in Figure 3.1. To demonstrate its potential in general applicability, we further evaluate LBPNet on a broader set of tasks including face and pedestrian detection as well as affNIST and observe promising results.

3.5.1 Datasets

Images in the MNIST dataset are hand-written numbers from 0 to 9 in 28×28 grayscale bitmap format. The dataset provides a training set of $60,000$ examples and a test set of $10,000$ examples. Both staff and students wrote the manuscripts. Most of the images can be easily recognized and classified, but there is still a portion of sloppy images in MNIST. SVHN is a photo dataset of house numbers. Although cropped, images in SVHN include some distracting numbers around the labeled number in the middle of the image. The distracting parts increase the difficulty of classifying the printed numbers. There are $73,257$ training examples and $26,032$ test examples in SVHN. Table 3.2 summarizes the details of the remaining seven datasets in our experiments. DHCD has handwritten Devanagari characters. ICDAR2005 contains three subsets, which are photos of numbers, lowercase and uppercase English characters. We shall note that the ICDAR2005 dataset was created mainly for text localization and recognition in the wild. We

use the cropped ICDAR because we only focus on the recognition task. Chars74K combines both numbers and English characters together and is considered to be challenging because an alphanumeric dataset that includes some labels is more prone to errors, e.g., classifying character O to number zero or vice versa. The three subsets of Chars74K are cropped photos, handwritten pictures, and printed fonts.

3.5.2 Experimental Setup

In all the experiments, we use all the training examples to train the LBPNet and validate on the provided test sets. There is no data augmentation used in the experiments.

In addition to the LBPNet shown in Figure 3.6 (c), we implement another version of LBPNet as a comparison: we utilize a 1×1 convolution to learn a combination of the LBP feature maps, as illustrated in Figure 3.6 (b). While this convolution still incurs too many multiplication and accumulation operations, especially when the number of LBP kernels increases, we shall demonstrate how this version of LBPNet performs for comparison purposes. In the rest of this section, we call the LBPNet using 1×1 convolution as the channel fusion mechanism **LBPNet (**1×1**)**, and our proposed LBPNet utilizing random projections **LBPNet (RP)** (totally convolution-free). The number of sampling points in a pattern is set to 4, and the size of the window within which the pattern can be deformed is 5×5. A brief sensitivity analysis of the number of sampling points versus classification accuracy on MNIST is provided in the appendices.

LBPNet also has a multilayer perceptron (MLP) block, which consists of two fully-connected layers of 512 neurons and the number of classes in every dataset, respectively. In addition to the nonlinearities, there is one batch normalization layer. The MLP block's performance without any convolutional layers or LBP layers is shown in Table 3.3, and the results on SVHN are in the appendices. The model size and speed of the MLP block are excluded in the comparisons since all the models have an MLP block, and so we focus on the convolutional layers and LBP Layers.

To understand the capability of LBPNet when compared with existing convolution-based methods, we build two feed-forward streamline CNNs as baselines. **CNN-baseline** is designed with the same number of layers and kernels as our LBPNet; the other **CNN-lite** is designed subject to the same memory footprint as the LBPNet (RP). The basic block of the CNNs contains a spatial convolution layer (Conv) followed by a batch normalization layer and a rectified linear layer (ReLU).

In the BNN [HCS$^+$16b] paper, the classification on MNIST was performed with a binarized multilayer perceptron network (MLP). We adopt the binarized convolutional neural network (BCNN) in [HCS$^+$16b] for SVHN to perform the classification and re-produce the same accuracy as shown in [LXZ$^+$17] on MNIST.

3.5.3 Experimental Results

Table 3.3 summarizes the experimental results of LBPNet on MNIST together with the baseline and previous works. We consider three metrics: classification error rate, model size, and the number of operations during inference. As a reference, we also provide the reduction in the number of operations compared with the baseline CNN. The number of operations in giga-operation (GOP) is used for a fair comparison of computation complexity regardless of platforms and implementation optimizations, such as loop tiling or unrolling, pipelining, and memory partitioning.

MNIST. The CNN-baseline and LBPNet (RP) share the same network structure, i.e., 39-40-80, and the CNN-lite is limited to the same memory size, and so its network structure is 2-3. The baseline CNN achieves the lowest classification error rate of 0.44%. The BCNN-6L achieves a decent speedup while maintaining the classification accuracy. Notwithstanding LBCNN-75L claimed its saving in memory footprint, to achieve 0.49% error rate, 75 layers of LBCNN basic blocks are used. As a result, LBCNN-75L loses speedups. Both the 3-layer LBPNet (1x1) with 40 LBP kernels and 40 1-by-1 convolutional kernels and the 3-layer LBPNet (RP) achieve an error rate of 0.50%. Despite the slightly inferior performance, LBPNet (RP) reduces the model

Table 3.3. The performance of LBPNet on MNIST.

	Error ↓	Size ↓ (Bytes)	#Operation ↓ (GOPs)	Reduction ↑
MLP Block	24.22%	-	-	-
CNN-baseline	0.44%	1.41M	0.089	1X
CNN-lite	1.20%	792	0.0004	219X
BCNN-6L	0.47%	153.7K	0.304	0.292X
BCNN-6L-noBN	88.65%	146.5K	0.303	0.293X
BCNN-3L-noBN	89.60%	5.94K	0.087	1.02X
LBCNN-75L	0.49%	12.2M	6.884	0.013X
LBCNN-75L-noBN	90.20%	2.8M	6.882	0.013X
LBCNN-3L-noBN	90.20%	244K	0.276	0.322X
LBPNet (this work)				
LBPNet (1x1)	0.50%	1.27M	0.011	7.80X
LBPNet (RP)	0.50%	715.5	0.0007	136X

size to 715.5 bytes and the number of operations to 0.7MOPs. Even BCNN cannot be on par with such a vast memory and computation reduction. The CNN-lite demonstrates that, if we shrink a CNN model down to the same memory size as the LBPNet (RP), the classification performance of CNN is compromised.

In addition to reproducing the results of BCNN-6L and LBCNN-75L with their open-sourced code, we remove the batch normalization layer inside every basic block (BCNN-6L-noBN and LBCNN-75L-noBN) and reduce the model to 3 layers (BCNN-3L-noBN and LBCNN-3L-noBN) for a fair comparison with LBPNet (RP). As shown in Table 3.3, once the batch normalization layers are removed, interestingly and surprisingly, both BCNN and LBCNN result in high error rates – almost identical to random guess – 90%. In other words, neither BCNN nor LBCNN can learn useful features without BatchNorm Layers. Meanwhile, LBPNet still achieves the state-of-the-art accuracy without the support of batch normalization.

SVHN. For the results on SVHN, we observe a similar pattern to the results on MNIST. Therefore, we defer the results and discussion on SVHN to the appendices.

More OCR Results. Table 3.4 lists the results of LBPNet (RP) on all the character recognition datasets studied in this chapter. The network structures of both the baseline CNNs

Table 3.4. The structures and experimental results of LBPNet on all considered datasets.

	Model	Structure	Error↓	Size↓	Size Red. ↑	GOPs ↓	Op Red. ↑
MNIST	CNN-3L	39-40-80	0.44%	1.41M	-	0.089	-
	LBPNet(RP)	39-40-80	0.50%	715.5	1971X	0.0007	136X
SVHN	CNN-8L	37-40-80-80-160-160-320-320	6.69%	10.11M	-	1.86G	-
	LBPNet(RP)	37-40-80-80-160-160-320-320	7.10%	10.62K	952X	0.010	193X
DHCD	CNN	63-64-128-256	0.72%	4.61M	-	0.637	-
	LBPNet(RP)	63-64-128-256	0.81%	2.30K	2004X	0.002	304X
ICDAR-Digits	CNN	3-4	0.00%	44.47K	-	0.0002	-
	LBPNet(RP)	3-4	0.00%	31.5	1411X	0.00003	7.76X
ICDAR-LowerCase	CNN	3-4	0.00%	44.47K	-	0.0002	-
	LBPNet(RP)	3-4	0.00%	31.5	1411X	0.00003	7.76X
ICDAR-UpperCase	CNN	3-4	0.00%	44.47K	-	0.0002	-
	LBPNet(RP)	3-4	0.00%	31.5	1411X	0.00003	7.76X
Chars74K-EnglishImg	CNN	63-64-128-256-512	40.54%	12.17M	-	2.487	-
	LBPNet(RP)	63-64-128-256-512	41.69%	4.793K	2539X	0.004	152X
Chars74K-EnglishHnd	CNN	63-64-128	28.68%	1.95M	-	0.174	-
	LBPNet(RP)	63-64-128	26.63%	1.15K	1699X	0.001	610X
Chars74K-EnglishFnt	CNN	63-64-128	21.91%	1.95M	-	0.174	-
	LBPNet(RP)	63-64-128	22.74%	1.15K	1699X	0.001M	610X

and LBPNets are designed to be the same for a fair comparison. The model sizes are the actual file sizes (without compression) of the LBP layers, including the discrete LBP kernels and random projection maps. Regarding the model size reduction, it is noteworthy that the wider the model is (i.e., more kernels), the higher the memory reduction rate we can obtain, with the cause explained earlier in section 3.3.2.

LBPNets not only deliver competitive results with the baseline CNNs but also achieve or approach the state-of-the-art error rates listed in Table 3.2. In other words, LBPNet reduces resource demands while maintaining the classification performance on OCR tasks.

3.5.4 Results on Other Objects and Deformable Patterns

We also explore how LBPNet performs on datasets containing general objects. Throughout the following experiments, we built CNNs and LBPNets with structures similar to the one for MNIST, as detailed in the first row of Table 3.4. We observe that LBPNet is able to achieve the same order of reductions in model size and operations.

Pedestrian: We first evaluate LBPNet on the INRIA pedestrian dataset [DT05], which consists of cropped positive and negative images. Note that we did not implement an image-based object detector since this is not the focus of this study. Figure 3.7 shows the trade-off curves of a

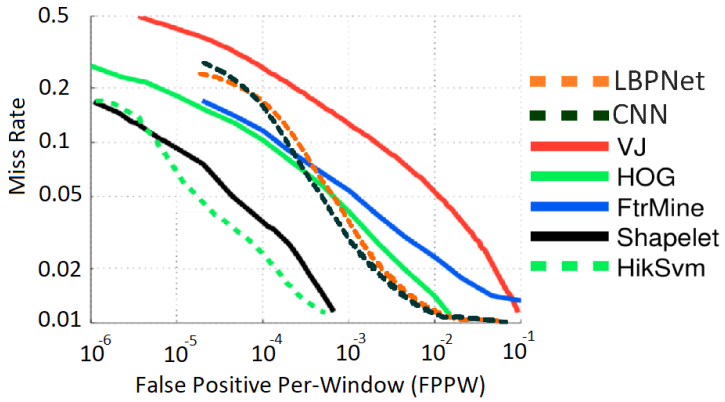

Figure 3.7. Classification error trade-off curves of a 3-layer LBPNet and a 3-layer CNN on the INRIA pedestrian dataset [DT05]. We also plot the results in Figure8(a) of [DWSP09] for comparison with the other five approaches.

3-layer LBPNet (37-40-80) and a 3-layer CNN (37-40-80).

Face: We also examine how well LBPNet performs on the FDDB dataset [JLM10] for face classification. Same as previously, we perform training and testing on a dataset of cropped images; we use the annotated positive face examples with cropped four non-person frames in every training image to create negative face examples, for the purposes of both training and testing. The structures of the LBPNet and CNN are the same as before (37-40-80), and LBPNet achieves 97.78% while the baseline CNN reaches 97.55%.

affNIST: We conduct another experiment on affNIST [Tie13], which contains 32 translation variations of MNIST (including the original MNIST). To accelerate the experiment, we randomly draw three variations of each original example to get training and testing subsets of affNIST. We repeat the same process to draw examples and train the networks ten times to get an averaged result. The network structure of LBPNet and the baseline CNN are the same, 39-40-80. To improve the translation invariance of the networks, we use two max-pooling layers following the first and second LBP layer or the convolutional layer. With the training and testing on the subsets of affNIST, LBPNet achieves 93.1%, and CNN achieves 94.88%.

Traffic Sign: Traffic sign recognition (TSR) is an essential task in autonomous driving systems. Dispatching low-level tasks such as TSR to low-cost/low-power compute nodes to

Table 3.5. The performance of LBPNet on two traffic sign datasets.

	Model	Structure	Error↓
GTSRB	CNN	61-64-128-256-512	1.16%
	LBPNet(RP)	61-64-128-256-512	1.99%
BTSC	CNN	39-40-80	2.30%
	LBPNet(RP)	39-40-80	2.51%

relieve the workload for central SIMD workstation is the modern trend in system designs. The state-of-the-art error rates are 0.29% [AGÁGSM18] and 1.08% [YLW+16] for GTSRB and BTSC, respectively. Table 3.5 lists the classification error rates on the two traffic sign classification datasets. Although the results on the two datasets are slightly weaker than the baseline, the reductions in model size and operations, which are on the order as shown in Table 3.4, hold promise for deploying TSR tasks on low-cost compute nodes.

3.6 Conclusion and Future Work

In this work, we have built a convolution-free, end-to-end LBPNet upon basic bitwise operations and verified its effectiveness on character recognition datasets. Without significant loss in classification accuracy, LBPNet can achieve orders of magnitude reductions in inference operation (100X) and model size (1000X), when compared with the baseline CNNs. The learning of local binary patterns yields unprecedented model efficiency since, to the best of our knowledge, there is no compression/discretization of CNNs that can achieve a kilobyte level model size while still maintaining the state-of-the-art accuracy on the character recognition tasks. We also provide encouraging preliminary results on more general tasks such as pedestrian and face detections. LBPNet points to a promising direction for building a new generation of lightweight, hardware-friendly deep learning algorithms to deploy on resource-constrained edge devices.

Chapter 3 contains reprints of Jeng-Hau Lin, Justin Lazarow, Yunfan Yang, Dezhi Hong, Rajesh K. Gupta, Zhuowen Tu. “Local Binary Pattern Networks for Character Recognition”. *S*ubmitted to International Conference on Computer Vision 2019 (ICCV). This dissertation

author is the primary author of this paper.

Chapter 4

LBPNet Accelerator

Fueled by the success of mobile devices, the computational demands on these platforms have been rising faster than the computational and storage capacities or energy availability to perform tasks ranging from recognizing speech, images to automated reasoning and cognition. While the success of convolutional neural networks (CNNs) have contributed to such a vision, these algorithms stay out of the reach of limited computing and storage capabilities of mobile platforms. It is clear to most researchers that such a transition can only be achieved by using dedicated hardware accelerators on these platforms. However, CNNs with arithmetic-intensive operations remain particularly unsuitable for such acceleration both computationally as well as for the high memory bandwidth needs of highly parallel processing required. In this chapter, we implement and optimize an alternative genre of networks, local binary pattern network (LBPNet) which eliminates arithmetic operations by combinatorial operations thus substantially boosting the efficiency of hardware implementation. LBPNet is built upon a radically different view of the arithmetic operations sought by conventional neural networks to overcome limitations posed by compression and quantization methods used for hardware implementation of CNNs. This paper explores in depth the design and implementation of both an architecture and critical optimizations of LBPNet for realization in accelerator hardware and provides a comparison of results with the state-of-art CNN on multiple datasets.

4.1 Introduction

Convolutional Neural Networks (CNNs) [LBD+89b, KSH12, SZ15] have outperformed other supervised learning methods in computer vision and have been used in many domains, such as web searching [HHG+13b], speech/pattern recognition [DSH13], biomedical analysis [SLF14b]. For most applications, CNN training through convex optimization requires intensive gradient computations. As a result, often the training phase is offline and done separately from the target platforms where recognition tasks may be needed. This is particularly true of devices at or near the edge of the network, the so-called "edge devices." Even with this split, the convolution operation in the inference phase still overburdens the resource-limited embedded hardware [RORF16b] for the Internet of Things (IoT) or real-time edge computing applications. To be specific, the edge devices challenges consist of congested inter-neurons connections, intensive memory accesses, large memory footprint to store parameters and feature maps, and high-latency high-precision multiplication and accumulation (MAC) operations.

There are two main approaches to alleviate the burdens of CNNs for hardware implementations. One approach is to prune the less salient weights to skip the arithmetic operations with less significant numbers [LDS+89, HSW93, GYC16]. The other is to quantize floating numbers either statically [ZHMD17] or dynamically [QWY+16] to degrade the precision for low-bit arithmetic logic units (ALUs). Binarization [HCS+16b, RORF16b, LXZ+17] pushed the static quantization to the limit, and thereby the original floating point multiplication was replaced with a 1-bit exclusive-nor gate (XNOR). There existed more intricate hybrid works of the two trends [HMD15], as well as other explorations of efficient network structures [SIV16, XGD+17] that mainly aimed to reduce the model size from the network structure level.

While carrying out the trained models of the two aforementioned approaches, hardware platforms driven by CPU and GPU clusters inevitably encountered challenges such as the overhead of irregular memory accesses arisen from the pruned irregular matrices, and the limitation of current computer architecture to support sub-word variable storage units and

arithmetic operations. On the other hand, field programmable gate arrays (FPGAs) provide an attractive alternative because they allow a highly customized design to handle the limitations on CPU/GPU machines [QWY+16]. Many hardware accelerators for pruned CNNs or binarized CNNs have been proposed [ZSZ+17, UFG+17]. However, the equivalent compression rate of memory footprint and computation latency are still incremental and continue to be a challenge for effective use of machine learning in edge devices.

LBPNets [LYGT18] fundamentally transforms the arithmetic multiply-and-accumulate (MAC) operations into sampling processes and logic operations. We note that despite the similarity in names, local binary pattern (LBP) and LBPNets are two very different techniques. LBP refers to a known method in the computer vision [OPH96] as a type of visual descriptor used in image classification based on texture maps. Such a descriptor could be used by various classifier algorithms including support vector machines or other machine learning algorithms. LBPNet is a new way to implement neural network algorithms, which obviates the need for computing dot products and sliding windows for convolution operations. Instead, LBPNets sample and compare the input image and records the comparison results to a predefined bit location. In other words, there is no MAC operation in an LBP layer, and only the trained patterns of sampling locations need to be stored. Therefore, the convolution-free LBPNets are hardware-friendly that can achieve significant benefits over the other CNN models.

We implement and optimize an LBPNet for multiple datasets on FPGA targets to characterize its efficiency. Based on these experiments, we propose an efficient architecture for LBPNet and critical optimization strategies. We implement and evaluate the complete system in terms of classification accuracy, latency, resource utilization, and energy efficiency.

4.2 Preliminary

Since the LBPNet [LYGT18] was proposed to be an alternative of the prevailing deep learning method CNN, we start from the preliminary knowledge of CNNs.

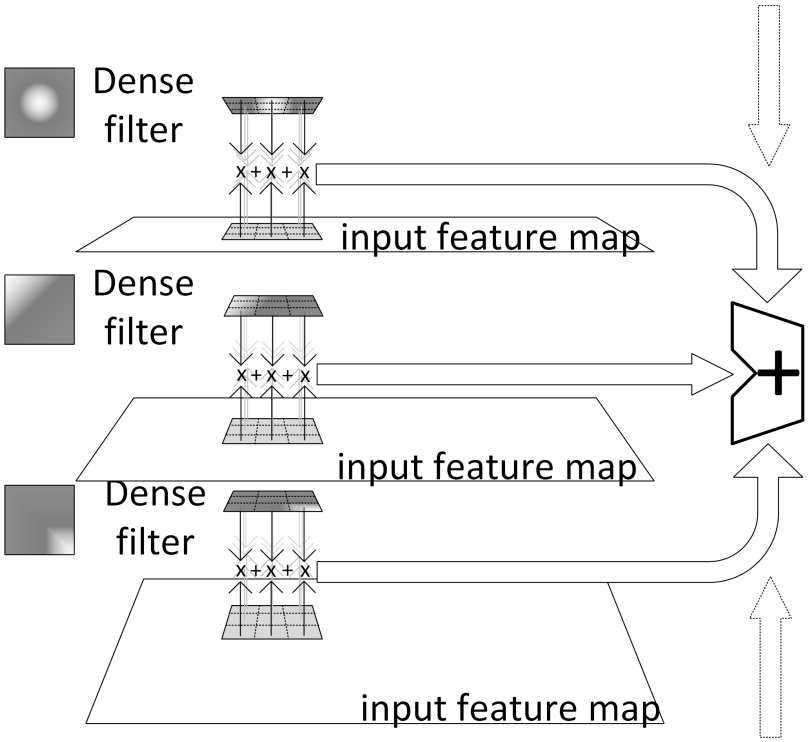

Figure 4.1. The snapshot of a convolution process in a hardware point of view. Dot-products and sliding window operation compose the convolution. The dot-product results from all input channels are fused together by a summation.

4.2.1 Convolutional Neural Networks

Figure 4.1 illustrates the operation in a convolutional layer (Conv layer). A Conv layer performs a 2-D spatial convolution on the input images or feature maps with kernels composed of multi-channel dense filters. Stacking multiple Conv layers up means taking the output feature maps of the previous Conv layer as the input of the current concerned Conv layer. Deepening network structure can extract more abstract representations embedded in the images for classification.

4.2.2 Local Binary Pattern Network

LBPNet [LYGT18] operates based on the optical flow theory, more specifically in addressing the aperture problem. The image gradient of the input image/feature map guides the training of patterns. The local binary patterns are trained to minimize the cross-entropy of a

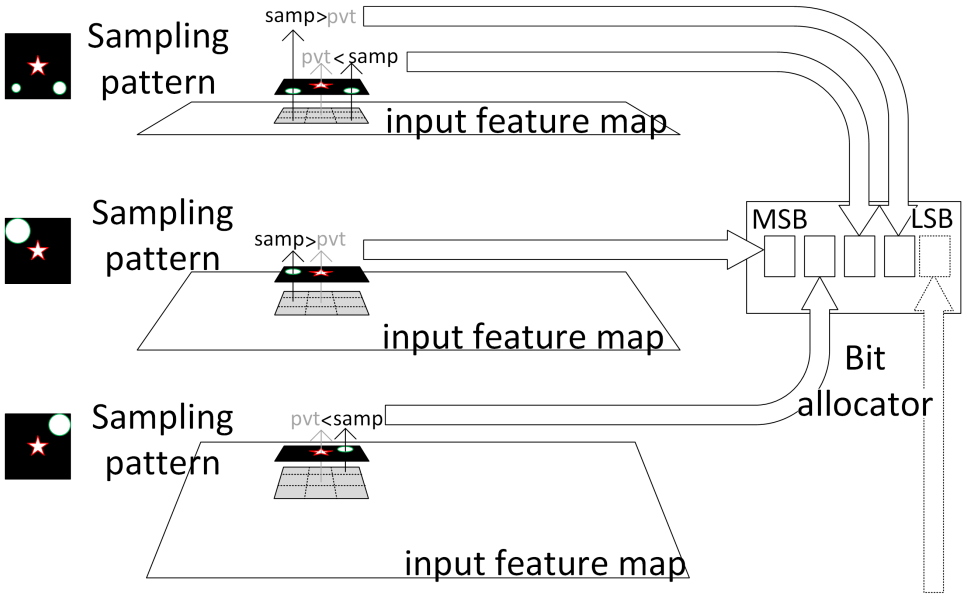

Figure 4.2. The snapshot of the LBP operation from a hardware point of view. Memory indexing and pixel comparison compose the LBP operation. The larger the round aperture is, the higher order bit the comparison result is allocated. Random projection fuses the comparison results from all input channels together by a bit allocator according to a predefined random projection mapping table.

softmax cost function. The learning process deforms the sampling points in a pattern to a better set of locations for discriminative features. The channel fusion process is done with a random projection, which has bee proven to be an effective distance preserving method [BM01].

Figure 4.2 illustrates the operations in an LBP layer; three local binary patterns are visualized as black masks with certain apertures on them. Analogous to a Conv layer, there are multiple local binary patterns, which record the sampling positions for the comparison with a pivot sampling. For each comparison pair, the pivot(a star-shaped aperture) and a sampling point(a round aperture) are used to index two values from the input features. The results of comparisons are allocated to predefined locations in a bit array. If multiple input channels presents, results of LBP operations on all channels are fused together according to a random projection map. There is no MAC operation or convolution in an LBP Layer. The low bit comparison can be implemented in combinational logic, and bit allocations require only a good buffer design.

The benefit of LBPNet is multi-fold. First, the sparse sampling pattern greatly reduces the model size. An LBP pattern contains $N_{sampling}$ sampling points' locations on a window. Assuming the number of input channel is N_{in}, and the number of output channel is N_{out}, the number of sampling locations is $2 * N_{out}N_{in}N_{sampling}$, where the number 2 means the two dimension locations. However, the presence of random projection instructs us to compare only a part of the sampling pairs and drop the unused pairs. Therefore, we only need store $2 * N_{out}N_{sampling}$ sampling positions and a mapping table of size $N_{out} \times N_{Sampling}$. All of them are typically in Kbits.

Second, the convolution-free design of LBP layers unleashes the computation latency from the system pipelined cycles. We can design a customized comparator module for the data parallelism in an LBP Layer. The speedup of an LBP layer over a Conv layer with massive MAC operation is, therefore, guaranteed.

Third, LBPNets reduce hardware cost. In FPGA, it takes 62 LUTs to implement an 8-bit multiplier, and 8 LUTs for 8-bit adder, while a boolean comparator requires only 4 LUTs, which implies we can either use cheaper FPGA with less computation capability or implement more data parallelism within the same FPGA compared with a CNN-based accelerator.

Last but not least, the energy efficiency is expected to be higher than CNN-based accelerator because we only need comparison and buffering to implement LBPNets. For many application, such as unmanned aerial vehicles or hearing aided devices, short battery life is usually a critical issue. The hardware accelerated LBPNets can be used on the energy efficiency concerning applications to boost user experience.

4.3 Analysis and Modifications of LBPNets

The multi-layer perceptron (MLP) classifier in LBPNet [LYGT18] was not the main focus, and hence the floating arithmetics were adopted. To fill the gap between theory and practice, we modify the MLP classifier with two advanced techniques and train the networks

Table 4.1. The architecture of our modified LBPNet on MNIST. The binarized MLP classifier is composed of two binarized FC layers and one modified batch normalization layer [UFG+17]. Although binarized, the FC parameters can only be stored in an off-chip DRAM due to the limitation of FPGAs' typical on-chip BRAM size [ZSZ+17].

Layer	Input ch N_{in}	output ch N_{out}	Fmap dim d	Fmap size (Kbit)	Param (Kbit)
LBP1	1	39	32 x 32	-	2.18
Joint1	40	40	32 x 32	163.84	-
LBP2	40	40	32 x 32	-	2.24
Joint2	80	80	32 x 32	327.68	-
LBP3	80	80	32 x 32	-	4.48
Joint3	160	160	32 x 32	655.36	-
AvgPool	160	160	5 x 5	32.00	-
FC1	4000	512	1	4.10	2,056.19
BatchNorm	512	512	1	0.51	8.19
FC2	512	512	1	0.08	5.28
Total					
LBP				1,178.88	8.90
FC				4.69	2,069.66

from scratch for FPGA implementation. In this section, we start with an overview of the network structure and then dive into the modifications for hardware.

4.3.1 Structures of LBPNets

We implemented multiple LBPNets for different datasets. Despite the numbers of kernels and depths, all network structures share the same characteristics. In this section, we use the network for the MNIST dataset as an example to analyze the structure before describing the FPGA architecture.

The LBPNet structure we adopt is visualized in figure 4.3. We list the model sizes for the MNIST dataset in table 4.1.

There are three LBP layers in a pipeline extracting the feature maps. Each 3-D volume of LBP pattern contains four learnable sampling points and four pivot point in the center. After the LBP operation and channel fusion, there is a shifted ReLU function [LYGT18] to introduce nonlinearity in the LBPNet. A Joint layer after every LBP layer stacks the LBP results on the

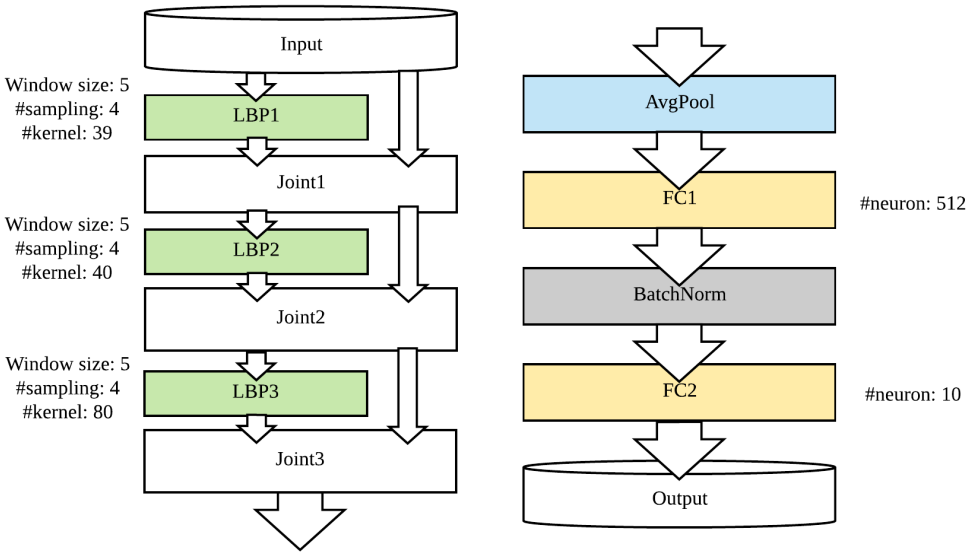

Figure 4.3. The network structure of the LBPNet for MNIST, which includes 3 LBP layers and 2 FC layers. The left sub-figure shows the LBP layers, and the MLP classifier is shown on the right-hand side.

input feature maps brought by the shortcut branch. Once again, there is no multiplication or additions in these operations.

For the MLP classifier part, an average pooling layer is then used to reduce 2-D images. The two binarized FC layers and one modified batch normalization layers are designed to reduce the dimension of data further and extract features for the 10 classes of MNIST dataset.

4.3.2 Quantization of the Average Pooling

All floating numbers in the MLP classifier must be quantized to fixed numbers or integers, or the floating-point calculation will form a bottleneck of the entire system. We remove the division from the AvgPool Layer and then profile all the output pixel values to linearly scaling them into the range of unsigned char by bit shifting.

4.3.3 Binarization of Weights

We binarize the weights of both the two fully connected layers to either -1 or 1. Then, we set all -1 to 0 for digital circuitry. The input of the first layer is the averaged value from the AvgPool Layer, which is in floating or fixed numbers. Although we cannot use an XNOR gate to replace the multiplication between the input and a weight, binarized weights enable us to use a multiplexer to select whether to add or subtract the input from an accumulator. The second binarized fully connected layer takes binarized input from the BatchNorm layer. Therefore, we can replace the multiplication with an XNOR operation in the dot-product. However, binarization has proven to be lossy [CBD15]. We must expect inferior classification accuracy and evaluate the difference for the trade-off between binary and floating arithmetic operations.

4.3.4 Simplification of Batch Normalization Layer

To avoid on-chip floating point arithmetic operations, we also introduce the method for the batch normalization layer mentioned in FINN [UFG+17]. This consists of methods to combine the binarization activation function with the linear transformation and calculating a threshold for each input activation off-line as shown in Eq. 4.1 and Eq. 4.2.

$$\gamma(x-\mu)/\sigma+\beta>0, \tag{4.1}$$

where x is the input, μ is the mean over a mini-batch, σ is the standard deviation over a mini-batch, γ is the learned scaling factor, and β is the learned shifting factor.

$$\begin{cases} x > threshold & , \quad \gamma\sigma > 0 \\ x < threshold & , \quad \texttt{otherwise}, \end{cases} \tag{4.2}$$

where $threshold = \mu - \frac{\beta\sigma}{\gamma}$ is calculated off-line after the modification from Eq. 4.1 to Eq. 4.2 and is lossless due to the mathematical equivalence.

The modified LBPNets are trained on a GPU machine with an NVIDIA Tesla K40, and

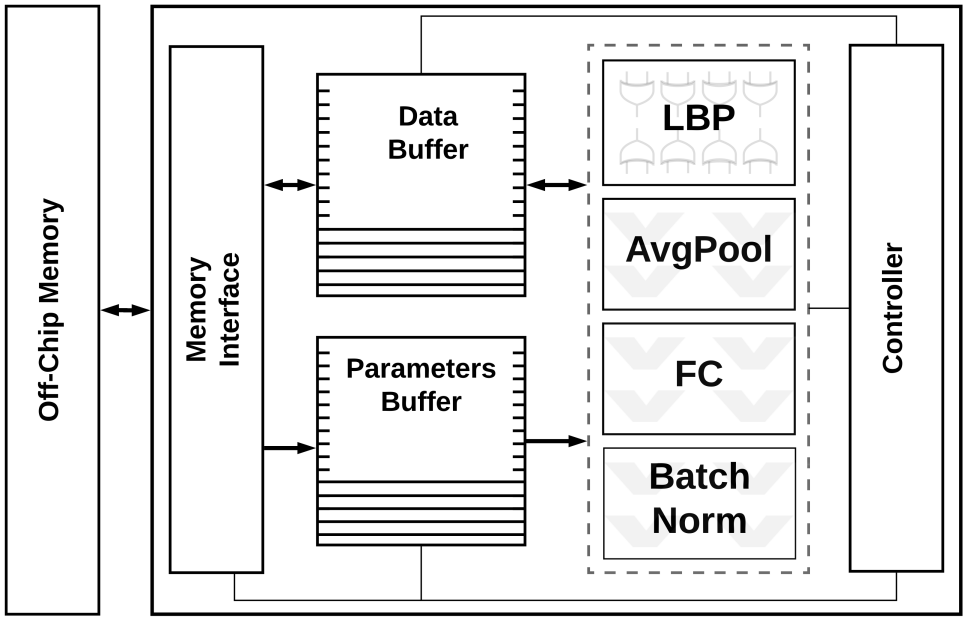

Figure 4.4. System-level architecture for LBPNet accelerator. The data buffer is used for storing both input data and intermediate results. Parameters Buffer stores the parameters of the network. A set of logical binary operations are dedicated to LBP operations. A set of ALUs is dedicated to each AvgPool, FC, and BatchNorm layer.

the training achieves 100.0% accuracy. The test accuracy is 99.34% on MNIST. Compared with the LBPNet paper [LYGT18], as mentioned earlier, we have sacrificed some classification accuracy to make LBPNet hardware-friendly through binarizing the MLP classifier.

4.4 FPGA Accelerator Design

4.4.1 Accelerator Architecture

An overview of the accelerator architecture is shown in Figure 4.4. The accelerator consists of four compute units as shown for each different type of layer, data and weight buffers, a memory access controller for off-chip memory transfers, and a controller. The operations of the LBP, Average Pool, Fully-Connected, and BatchNorm layers are performed through the four compute units *LBP*, *AvgPool*, *FC*, and *BatchNorm*, respectively. The LBP unit — dedicated to performing the compare operations in the LBP layer – consists of a set of logical elements, while the other compute units are made up of arithmetic units to perform operations such as addition

and the simplified multiplication in Section 4.3.3. The input data and the parameters of the layers loaded from the off-chip memory are stored into the on-chip Data buffer and Parameters buffer, respectively.

In addition to storing the input data, the Data buffer can accommodate the intermediate results of the layers which are necessary for the computations of their next layer. Because the size of the intermediate outputs of the layers is in 4-bit fixed numbers, they are small enough to be stored on the available on-chip memories. This eliminates the need to transfer intermediate outputs between the accelerator and the off-chip memory. Thus, off-chip memory transfers are only needed for the input image, loading each layers weights, and sending back the final prediction output. This is one of the benefits of LBPNet compared to most other CNN-based accelerators where the size of intermediate results in floating or longer bit-length fixed numbers typically exceeds the available on-chip storage. On the other hand, the Parameter buffer can store all the weights for all the LBP layers at once. So only one time data and weight load is required for all the LBP layers to compute the input of the AvgPool layer. On the other hand, there is only enough space to store a portion of the FC layers weights. Each time a new set of weights are loaded into the parameter buffer and a new set of intermediate result is generated. This continues until all FC layer outputs are generated and stored in the on-chip Data buffer. To accelerate the communication and parallelize computations, we pack our 8-bit weights and generate to 64-bit words, store these words in the buffers, and unpack them to perform parallel computations.

4.4.2 Execution Flow of the Accelerator

In the beginning, the input image and parameters of the three LBP layers are loaded from the off-chip memory to the Data buffer and Parameters buffer. Afterward, the LBP compute unit performs the corresponding comparison operations starting from the layer LBP1. Accordingly, the output of LBP1 is stored in Data buffer on top of the input data. The process continues until all the LBP layers are performed. Then, the AvgPool unit starts performing a quantized version of average pooling operations on the data to reduce its dimensions. At this point, the parameters

stored in the Parameters buffer are not needed any longer, and the space can be freed to store the parameters of other layers.

The next pass operates on layer FC1. Since the parameters of this layer exceed the size of the Parameters buffer, a portion of the parameters are loaded into the on-chip buffer, and the corresponding MAC operations are performed, and the partial outputs are stored in the Data buffer. Then, the process moves to the computations with the next part of parameters by loading them into Parameters buffer and performing the corresponding FC computations. After all the computations of the layer FC1 are completed, the parameters of the BatchNorm layer are brought into the Parameters buffer and overwrite the parameters of FC1. Then, the compute unit BatchNorm performs the batch normalization on the results of FC1 which are available in the Data buffer. The new outputs generated by the BatchNorm unit are stored in the Data buffer. Finally, the computations of the last layer are performed similarly to executing the layer FC1. The last FC unit generates prediction output values. The final label is computed using *ArgMax* operation on the results of the last FC layer and is written back to off-chip memory.

4.4.3 Compute Units Architecture

LBP Layers: The LBP unit is the most critical component of the accelerator responsible for a number of repeated LBP layers. Each unit in the LBP layer is responsible for reading eight input pixels from the data buffer and performing four comparison operations to generate one output. The position of these eight points are read from the weight buffer; then we can access the corresponding locations in the data buffer, compare every two of them together, and generate the corresponding output pixel by concatenating the four comparison results. As the weights are 8-bits for these layers, and they are stored in 64-bit words, we only need to read two words from the weight buffer which can be done in one cycle. These values indicate the position of points that should be accessed from a tensor in the data buffer. After reading each two-pixel values, a comparison is performed, and 1 bit of the output pixel is generated. This process is performed for every input channel in a pipeline fashion. This process is repeated in a

sliding window pattern for the whole image. To improve the latency of the LBP computations, the operations inside the LBP can be parallelized. In this case, we partition the tensor input horizontally, and each processing element performs the aforementioned operations on one part. In order for the processing elements to access to data buffer at the same time, we partition the data buffer BRAM horizontally. There is clearly a trade-off between resource utilization and performance as we change the level of parallelism.

FC Layer: Each cycle we read in N data words and an equal number of weight words. N here is the input parallelization factor (we used 8 in our implementation). We apply appropriate memory partitioning to be able to access to 8 data words in one cycle. N simplified multiplications implemented with multiplexers are done in parallel, and this process is pipelined until an output is generated. As we perform quantization on FC layers, we only have fixed-point accumulator and multiplexers. After the computations on the available set of weights are done, a new set of weights are loaded from the off-chip memory, and the next set of outputs are generated. Note that the level parallelism in FC layer is typically bound by the memory bandwidth of the off-chip connection, rather than the throughput of the accelerator.

BatchNorm Layer: We implement batch normalization layer using a parallel comparison between the data and weights, and multiplexers [UFG+17] to generate a binarized output for the next fully-connected layer. In each cycle, eight parallel comparisons are made to generate eight outputs, and this process is performed in a fully pipelined fashion.

AvgPool Layer: This layer is relatively simple. It averages over 5-by-5 windows from input channels. The required memory read, computation, and memory write is fully pipelined.

4.5 Experimental Results

4.5.1 Dataset

Table 4.2 lists all the datasets used in this experiment. We convert all colorful images from RGB channels to YUV channel and only use the Y-channel as the input image to train

Table 4.2. The datasets we used in the experiment.

	#Class	#Examples	Description
MNIST	10	60,000+10,000	Handwritten number
SVHN	10	73,257+26,032	Photos of house number
DHCD	46	46x2,000	Handwritten Devanagari characters
ICDAR-DIGITS	10	988	Photos of numbers
ICDAR-UpperCase	26	5,288	Photos of lower case Eng. char.
ICDAR-LowerCase	26	5,453	Photos of upper case Eng. char.
Chars74K-EnglishImg	62	7,705	Photos, Alphanumeric
Chars74K-EnglishHnd	62	3,410	Handwritten, Alphanumeric
Chars74K-EnglishFnt	62	62,992	Printed Fonts, Alphanumeric

LBPNets and verify on FPGA after the hardware implementation.

4.5.2 Experiment Setup

We implemented our design in C++ and used Xilinx Vivado HLS and Vivado Suite 2015.4 as the primary tool for synthesizing the accelerator. We evaluate resulting designs on a low-cost Xilinx Zynq-7000 series (XC7Z020 FPGA) target. We performed HLS design space exploration to select the design options that strike a balance between resource utilization and latency. In our final design, we use 64-bit words, and the LBP compute unit consists of four parallel processing units. The resource utilization and power numbers are reported by Vivado tool after placement and route.

4.5.3 Results

Table 4.3 lists the LBPNet structure and the accuracy for every dataset as well as the CNN baseline. For those baseline results without references, we build CNNs with the same network structures like LBPNets.

The resource utilization for our design is 7954 LUT, 7188 FF, 68 BRAM, and 16 DSP. Our FPGA implementation works at 200 MHz. We evaluate performance of our accelerator for different datasets. The latency break-down for different layers and total execution time is summarized in Table 4.4. The last column in this table (labeled as *Total Runtime*), shows

Table 4.3. LBPNet structure and Accuracy. We use the same binarized MLP classifier throughout the experiment. The CNN baseline results are listed as well.

	layer structure	CNN Baseline	Accuracy
MNIST	39-40-80	99.60% [SSP03]	99.34%
SVHN	39-40-80-160-320	95.10% [SCL12]	93.40%
DHCD	63-64-128-256	98.47%[APG15]	99.16%
ICDAR-Digit	3-4	100.00%	100.00%
ICDAR-Lower	39-40-80	100.00%	100.00%
ICDAR-Upper	39-40-80	100.00%	100.00%
Chars74K-Img	63-64-128-256-512	47.09%[DCBV09]	58.31%
Chars74K-Hnd	63-64-128	71.32%	73.37%
Chars74K-Fnt	63-64-128	78.09%	77.26%

Table 4.4. Latency (number of clock cycles) break-down for different layers and total run time for different datasets. The runtime is in millisecond.

	LBP	AvgPool	FC	BatchNorm	Total Runtime
MNIST	286953	72726	260399	75	3.1
SVHN	1227777	290826	1477931	75	15.6
DHCD	961693	232671	822571	75	10.5
ICDAR-Digit	11820	3619	15147	75	0.16
ICDAR-L/U	286953	72726	260399	75	3.24
Chars74K-Img	1965239	465308	1641771	75	21.2
Chars74K-Hnd	459920	116361	260399	75	4.3
Chars74K-Fnt	459920	116275	260399	75	4.3

the execution time (in millisecond) per image for the optimized design for different datasets. This table also shows the break-down of latencies (number of cycles) per layer in columns 2-5. Column 2 and 4 shows the sum of latencies for all LBP layers and FC layers. Since different LBP and FC layers work on different data sizes, their latency is different. For example, for the MNIST dataset, the latency of three LBP layers are 51745, 78404, 156804 cycles respectively. For the same dataset, the latencies in the two FC layers are 259588 and 811 cycles respectively. Similarly, there are differences in the runtime for different datasets because they use different numbers of LBP kernels.

We compare our design with off-the-shelf CNN FPGA implementations. Table 4.5 compares the resource utilization for different FPGA implementations of LeNet with LBPNet.

Table 4.5. The comparison of resource utilization, throughput, and accuracy in different implementations of LeNet and LBPNet. Numbers for resource utilization is in percentage.

	[VB16]	[FHCW16]	[FHCW16]	[GWL$^+$18]	LBPNet
Backbone Precision	LeNet-5 fixed-16	LeNet-5 floating-32	LeNet-5 fixed-24	LeNet-5 fixed-16	LBPNet-3L fixed-4 & 1-bit
DSP	3.64	80	43	93.18	7.27
BRAM	13.2	83	66	86.43	48.60
LUT	54.64	77	73	71.68	15.20
Flip-Flops	39.02	25	26	40.05	6.75
Clock (MHz)	100	100	166	200	200
Throughput(GOP/s)	0.48	-	-	76.48	61.62
Accuracy(%)	97.92	99.01	99.01	99.10	**99.34**

As shown, our accelerator achieves the highest accuracy among all implementations. It utilizes only 48.6% of BRAMs, 7.3% of DSP units, 15.2% of LUTs, and 6.8% of Flip flops on our target FPGA. Comparing to CNN architectures, we mostly have better resource utilization. We also compare our throughput to other works. Throughput is shown in giga-operations-per-second (GOPS). Our accelerator achieves better throughput that CNN-based accelerators. We have better power consumption when compared to CNN implementations. For example, [FHCW16] utilizes 3.32 W power, while our accelerator consumes only 0.5 W to perform classification. Our accelerator is more energy efficient than CNN due to replacing expensive convolution operations with simple logical operations. In general, LBPNet enables us to achieve a good balance between resource utilization and throughput, while maximizing accuracy.

LBPNet's inference operations on a CUDA-supported GPU presents a latency of 0.7 ms per image for MNIST dataset, the average power consumption of 130, and the memory consumption of 44 MBytes. By comparison, FPGA implementation of LBPNet here is 4.4X slower than a Tesla K40 GPU, but 52X more energy efficient compared to the GPU implementation.

4.6 Related Work

Our work is the first to approach the design of hardware-accelerated LBPNets. While it is hard to conduct a direct comparison with existing hardware accelerators for CNNs because

of the diversity of implementation choices, the effect on computation and memory size can be examined. Here we provide three published works regarding the compression of CNNs as a reference as these use common essential techniques.

Deep Compression [HMD15] utilized multiple techniques to achieve a compression rate of 35X: pruning, quantization, customized weight encoding, Hoffman encoding. However, owing to that the pruning and quantization retraining loops were not combined to minimize the interactive effects, there was no guarantee to the global minimum of the training result.

FINN [UFG+17] fully exploited the critical characteristics of BNN: 1) a popcount module was synthesized to count the number of 1's. 2) redesigning BatchNorm to threshold the popcount results with different values, which can be calculated off-line. Although FINN provided a complete synthesizing flow for trained BNNs, it degraded the classification accuracies because the padding subroutine was not correctly imposed.

BCNN Accelerator [ZSZ+17]. The authors adopted a partially shared streaming architecture and managed to process sub-word buffering and storing efficiently. By adding the 2-bit ignoring elements surrounding the feature maps, BCNN Accelerator circumvented the zero-padding issue as FINN encountered.

All the three works were based on CNN, and the smallest model size they achieved was still a couple of Mbits. The most efforts in those work were the quantization of CNNs because CNNs were not designed to be hardware-friendly. Instead, our accelerated LBPNets are designed for bit-wise operation since the development of its algorithm [LYGT18].

4.7 Conclusion and Future Work

We have presented an efficient accelerator for LBPNet on FPGA with distinct implementation advantages and ability to make hardware related design tradeoffs. The accelerated LBPNets achieve Kbit model size and high throughput while maintaining the state-of-the-art accuracy on all datasets. Our future work includes the combination of binarized FC layers with

LBP layers to further reduce the memory footprints and accesses an exhaustive exploration of systolic and SIMD architectures for the acceleration of LBP layers' indexing operations.

Chapter 4 contains reprints of Jeng-Hau Lin, Atieh Lotfi, Vahideh Akhlaghi, Zhuowen Tu, and Rajesh K. Gupta, "Accelerating Local Binary Pattern Networks with Software-Programmable FPGAs", Design, Automation, and Test in Europe (DATE), 2019. This dissertation author is the primary author of this paper.

Chapter 5

The Error Immunity of LBPNets

As a supervised learning method, neural networks (NNs) have shown broad applicability from medical applications, speech recognition, and natural language processing. This success has even led to the implementation of NN algorithms into hardware. In this chapter, we explore two questions: (a) to what extent microelectronic variations affects the quality of results by neural networks; and (b) if the answer to the first question represents an opportunity to optimize the implementation of neural network algorithms. Regarding first question, variations are now increasingly common in aggressive process nodes and typically manifest as an increased frequency of timing errors. Combating variations – due to process and/or operating conditions – usually results in increased guardbands in a circuit and architectural design, thus reducing the gains from process technology advances. Given the inherent resilience of neural networks due to the adaptation of their learning parameters, one would expect the quality of results produced by neural networks to be relatively insensitive to the rising timing error rates caused by increased variations. On the contrary, our results on multiple-layer perceptron (MLP), convolutional neural network (CNN), binarized convolutional neural network (BCNN), and local binary pattern network (LBPNet) show that physical variations can significantly affect the inference accuracy. This paper outlines our assessment methodology and use of a cross-layer evaluation approach that extracts hardware-level errors from twenty different operating conditions and then injects such errors back to the software layer in an attempt to answer the second question posed above.

5.1 Introduction

Neural network algorithms have found use in a wide range of applications such as medical diagnostics [YJZ+06], image classification [KSH12], speech recognition [HDY+12], and natural language processing [CWB+11]. This versatility has led to their implementation on a variety of hardware platforms: GPU [CLL+14], FPGA [HAM07], and ASIC [CDS+14].

With the continuous scaling of CMOS technology, the underlying transistors in all these implementations are increasingly susceptible to variations in manufacturing and operating conditions. Dynamic variations in microelectronic systems, which is the main focus of this paper, are caused by environmental factors such as supply voltage droops and temperature fluctuations. Voltage droops are caused in response to instantaneous current fluctuations due to activities on the power delivery network. Temperature fluctuation could alter the circuit parameters such as carrier mobility, threshold voltage, etc. Such variations can manifest themselves as timing errors, leading to incorrect computation results and system failures. Such variations have led to increasing use of overdesign and guardbands in a circuit and architectural design to ensure reliability, which reduces the gains from process technology advances.

Due to the ability to adapt their learning parameters and to extract the abstract common features in data, neural networks have an inherent resilience to errors. Thus, one would expect that the quality of results produced by hardware neural networks (HNNs) to be relatively insensitive to the rising timing error rates caused by increased variation, thus opening doors for opportunistic reduction of guardbands to increase the operational efficiency of hardware. There is a need for a quantitative assessment here to explore the extent to which guardbands can be reduced in HNNs. We investigate this question as to whether and how much accuracy of HNNs could be affected by dynamic variations. To do this, we capture and represent variations from low-level hardware, and then expose them to neural networks inferences. Unlike logic errors which can be derived through a mathematical formulation[DLC+15][MSS+16][ZWT+15], variation-induced timing errors can only be obtained using gate-level simulation, making the error injection implementation

time-consuming and not scalable.

Approach and Contributions: We propose a cross-layer approach to assess the vulnerability of HNNs to dynamic voltage and temperature variations, in which we extract the timing errors from hardware layer using gate-level simulations and examine their effects in the software layer using error injections. To evaluate the soundness of this approach, we measure the timing errors using gate-level simulations (GLS) of post-layout circuits in TSMC 45nm technology. We vary the voltage and temperature in a wide range to examine the effects of variations. Then, we represent and inject these timing errors to neural networks during their inference. Finally, we examine the resilience of four types of neural networks, MLP, CNN, BCNN, and LBPNet, by testing them on MNIST dataset[LCB98].

Based on our implementation and evaluation, this paper makes the following contributions:

- We extract the circuit level timing errors caused by voltage and temperature variations from twenty different operating conditions using gate-level simulations.
- We inject such timing errors back into neural network inference and evaluate the accuracy on MNIST dataset at different conditions.
- Our results quantitatively show that variations can significantly affect the inference accuracy on NNs.
- Among the four subject networks, LBPNet provides the strongest error immunity that the other three networks cannot be on par with.

5.2 Hardware Neural Networks

Modeled for neural processing, Figure 5.1 shows a typical neural network, an MLP consisting of an input layer, hidden layers, and an output layer. Except for the input layer, all remaining layers are composed of artificial neurons that represent the basic computation unit.

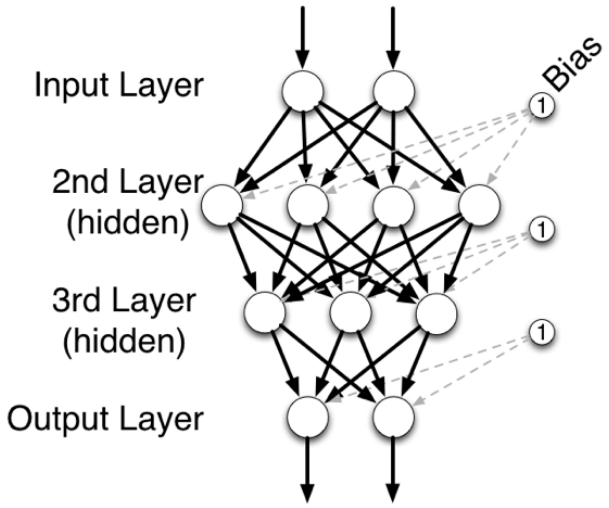

Figure 5.1. An example of a 4-layer multi-layer perceptron neural network.

An artificial neuron consists of a linear processing part followed by a non-linear processing part. The linear part collects the output information, namely the activations, from the previous layer. The collection method is a dot production between weights and activations. The nonlinear part includes regularization like dropout, and activation functions such as logistic sigmoid, hyperbolic tangent, or rectilinear unit. The nonlinear activation function enables a neural network to be a universal function approximator [Gyb89]. [RHW85b] intelligently applies the chain rule of calculus and gradient descent on neural networks to train the weights and hence minimizes the classification errors.

Since proposed in 1989, CNNs [LBD$^+$89a] have pushed the performance of neural networks to a new realm. Figure 5.2 depicts the internal processes in a convolutional layer with 9 kernels, each of which consists of three filters. The convolution operation models the hardwired bonding between the neurons on adjacent layers. It uses a sliding filter to perform dot-products of the filter and uses a portion of the input image to generate an output image, namely the feature map. Since the convolution operations are differentiable, the filters can be trained to capture the features of the input images with backward propagation [RHW85b]. *Pooling* is used to reduce the size of a feature map and increase the reception area by selecting the maximum pixel

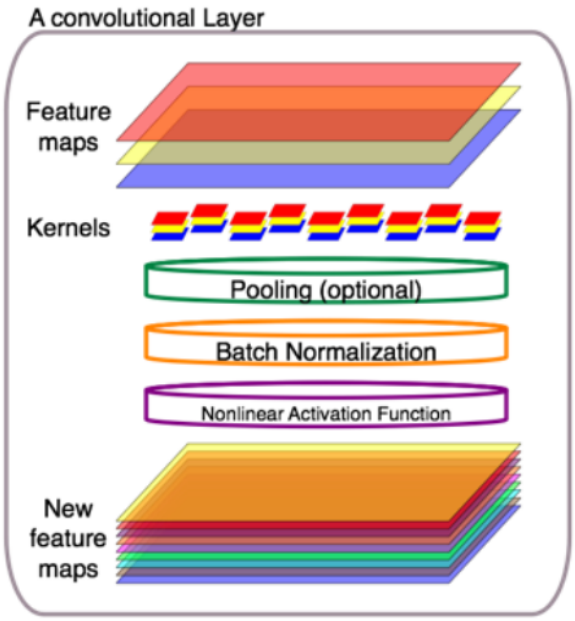

Figure 5.2. The processes among a convolutional layer.

strength or averaging several pixel strengths. It benefits the translation invariance because it drops unnecessary minor information and preserves the most dominant features for the overall classification task.

The robustness of a neural network comes from many aspects. From a higher level point of view, the training process of a neural network model is simply an ensemble of multiple linear or logistic regressions working in parallel. The regression itself ignores minor noises of the data and yields a model for the most likely distribution of the given data. Second, the regularization process inside a neural network also contributes to robustness because no matter how deterministically penalties on weights are added or how stochastically certain partials of the model are dropped, the weights are trained to accommodate the majority of the data with a simplest probable distribution. Moreover, if a retraining process is involved, the convex optimization enforces the learnable parameters in a model to descend on the error surface again. Please note that we only assess the inference performance in this work without performing any re-training.

Hardware variations could impact HNNs through timing errors in both computation

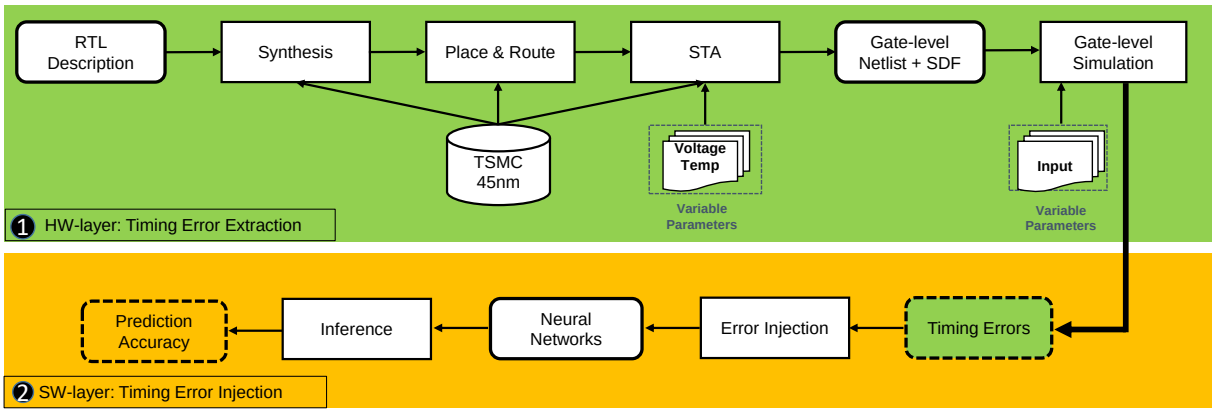

Figure 5.3. Cross-layer assessment flow with two stages: a) HW-layer: Timing Error Extraction to extract the timing errors under different operating conditions; b) SW-layer: Timing Error injection into neural network and perform inference.

logic and control logic. The errors in control logic could lead to catastrophic results, but, fortunately, most critical paths lie in computation logic, which is mainly composed of additions and multiplications, two of the most frequently used operations. Both the forward and backward propagation require intensive additions and multiplications, but most HNNs on ASIC, FPGA, and embedded GPUs do not support on-chip learning. Thus, we mainly focus on the timing errors that occur in addition and multiplication during the inference phase of HNNs.

5.3 Cross-layer Vulnerability Assessment

The cross-layer vulnerability assessment is comprised of two phases as shown in Figure 5.3: *Timing Error Extraction and Timing Error Injection*. a) The *Timing Error Extraction* phase implements the standard ASIC flow and uses gate-level simulation (GLS) to generate timing errors at each operating condition. b) In the *Timing Error Injection* phase, we inject the timing errors into neural networks and then perform inference. We vary the neural network genres and operating conditions to examine the resulted accuracy. More details about the two phases are illustrated as follows.

5.3.1 HW-layer: Timing Error Extraction

We extract the timing errors through *Timing Error Extraction* module as illustrated in Figure 5.3, which is divided into several steps. Note that we focus on dynamic variation-induced timing errors of computation units. We extract timing errors from the adder and the multiplier, which are the two most frequently used computation units in neural networks computation. We use FloPoCo [DDP11] to generate the synthesizable VHDL codes of floating point units. We use *Synopsys Design Compiler* to synthesize the Verilog codes and use *Synopsys IC Compiler* to generate the post place-and-route netlist in TSMC 45nm technology. Next, we use *Synopsys PrimeTime* to perform static timing analysis, generating Standard Delay Format (SDF) files at different operating conditions. To do this, we use the voltage-temperature scaling features of Synopsys PrimeTime for the composite current source approach of modeling cell behavior. We consider twenty operating conditions as shown in Figure 5.7, which could introduce both mild and aggressive timing errors. Then, we use *Mentor Graphics ModelSim* to do SDF back-annotation gate-level simulations under nominal frequency to generate output data at different operating conditions. To extract timing errors, we compare the GLS output $y[t]$ with a pure-RTL simulation result $y\_gold[t]$, which is free from timing errors because there is no delay annotation. If there is a mismatch, then we define it as a timing error.

5.3.2 SW-layer: Timing Error Injection

We inject the timing errors extracted by the *Timing Error Extraction* phase to the neural networks by using second phase *Timing Error Injection*. During the forward propagation in the neural network inference, we inject the errors into the arithmetic computations (addition and multiplication) in the convolutional layer (Conv layer), fully-connected layer (FC layer), average pooling layer (AvgPool layer), batch normalization layer (BatchNrom layer), and local binary pattern layer (LBP layer). There are several noteworthy facts must be highlighted regarding the error injection in the software layer: First, the XNOR operation and pop-count accumulation

in BCNN and the comparison operation in an LBP Layer are not implemented in conventional arithmetic and logic units (ALUs) on CPUs or processing elements (PEs) on GPUs. We have to use multiplier and adders to carry out the 1-bit XNOR and the following accumulation in BCNN. For the comparison in an LBP Layer, we use the sign bit of subtraction to produce the comparison result instead. Therefore, the TER from adders and multiplier can affect the outputs of binarized Conv, binarized FC, and LBP layers.

On a circuit, different input could excite different paths, resulting in an input-specific timing error behavior. To mimic this, an exhaustive look-up table containing the entire input space for each bit position of each computation unit under all operating conditions needs to be implemented. Then, the computations need to look up the table to check whether it has a match on any input operands in the input space. This makes the inference process prohibitively slow. To approximate the situation, we inject the timing errors as [SDF+11]: let the computation units return a random value each time they have timing errors. We inject the error into the computation with the pair of adder TER and multiplier TER extracted from the *Timing Error Extraction* phase to mimic the time error behavior. For example, if adder has a TER at 0.1, we inject errors to 10% of the total additions. This probability is determined by operating conditions and computation logic (addition or multiplication), which can represent the impact of timing errors on computation logic. We vary the error injection probability for each operating condition.

5.4 Experimental Results

In this section, we measure timing errors under twenty operating conditions. Then, we measure HNNs accuracy as a function of varying timing error rates. Finally, we characterize the HNNs accuracy under dynamic variations using MLP, CNN, BCNN and LBPNet.

5.4.1 Experimental Setups

In this work, we use tiny-dnn [Nom16], a header only, dependency-free deep learning library written in C++, as our deep learning platform for MLP and CNN. This platform is

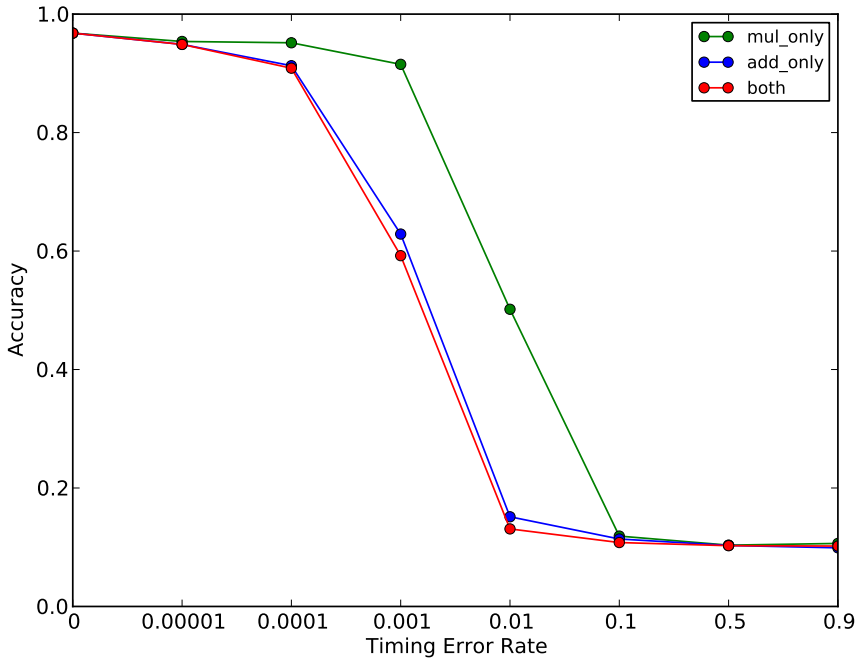

Figure 5.4. MLP accuracy as a function of TER.

light weighted and is designed for deep learning on the limited computational resource, such as embedded systems and IoT devices. For CNN, we use LeNet-5 like architecture and replace LeNet-5's RBF layer with normal fully-connected layer. For MLP, we use 3-layer MLP with a hidden layer of 60 neurons. We adopt the same structure of the BCNN for MNIST in the BNN paper [LXZ+17], and the LBPNet for MNIST in the previous chapter and the LBPNet paper [LYGT18]. The synthesizable C codes for BCNN and LBPNet implemented by us for FPGA accelerators are used for the error injection. All the for sets of weights and kernels are pre-trained either from the referred tiny-dnn source or by us on an Nvidia Tesla K40 GPU.

We use MNIST (Mixed National Institute of Standards and Technology) database of handwritten numbers [LCB98] as our dataset to evaluate the neural network accuracy. This dataset is a well-known dataset for evaluating the performance of neural network classifiers. The dataset is split into training set and test set with $60,000$ and $10,000$ 28×28 images. We vary the voltage from 0.81V to 0.90V with a step at 0.01V and the temperature from 50°C to 100°C.

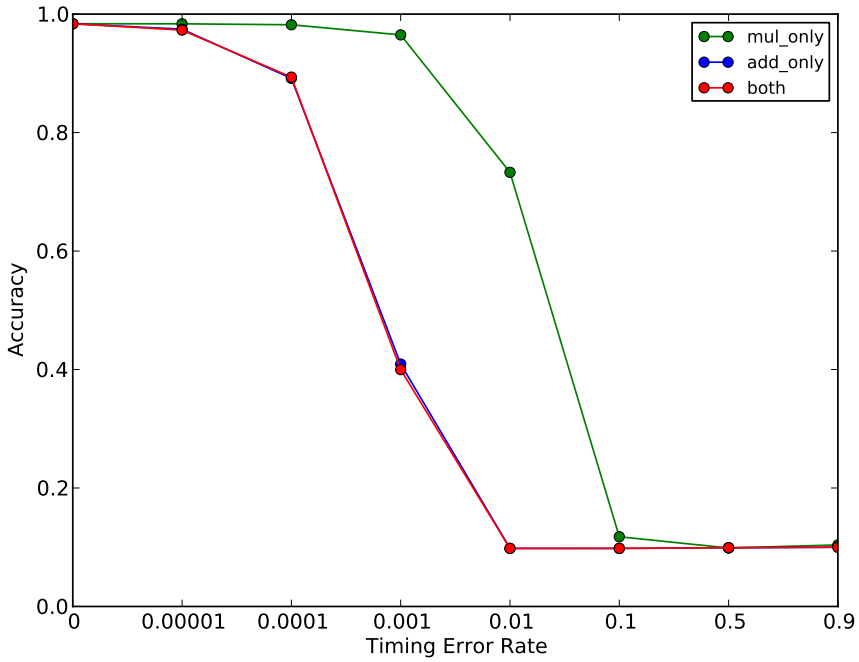

Figure 5.5. CNN accuracy as a function of TER.

5.4.2 Accuracy under the Threat of Timing Errors

Before the error extraction, we assess the performance degeneration as a function of timing error rates. The accuracy is evaluated for both MLP and CNN under the TER at 0, 0.00001, 0.0001, 0.001, 0.01, 0.1, 0.5, and 0.9 at three configurations as shown in Figure 5.4 and Figure 5.5; *add_only* means we only inject timing errors to adder, *mul_only* means we only inject timing errors to multiplier and *both* means we inject errors to adder and multiplier at the same time. We observe that for both MLP and CNN, as the TER increases, the accuracy drops monotonically. When the TER is 0.00001, the HNN can still deliver a decent accuracy close to original accuracy. Once the TER of adder reaches 0.0001, the accuracy drops to around 90% and continue dropping to 60% when the TER of adder reaches 0.001. In contrast, the multiplier exhibits much less significant impact on HNN accuracy: the HNN can still deliver 90% accuracy even when the TER of multiplier reaches 0.001. In fact, for all examined TERs, the *mul_only* resulted accuracy is always higher than that of *add_only*. Moreover, the accuracy under *both* configuration is almost identical to that of *add_only* configuration, suggesting that adders-induced

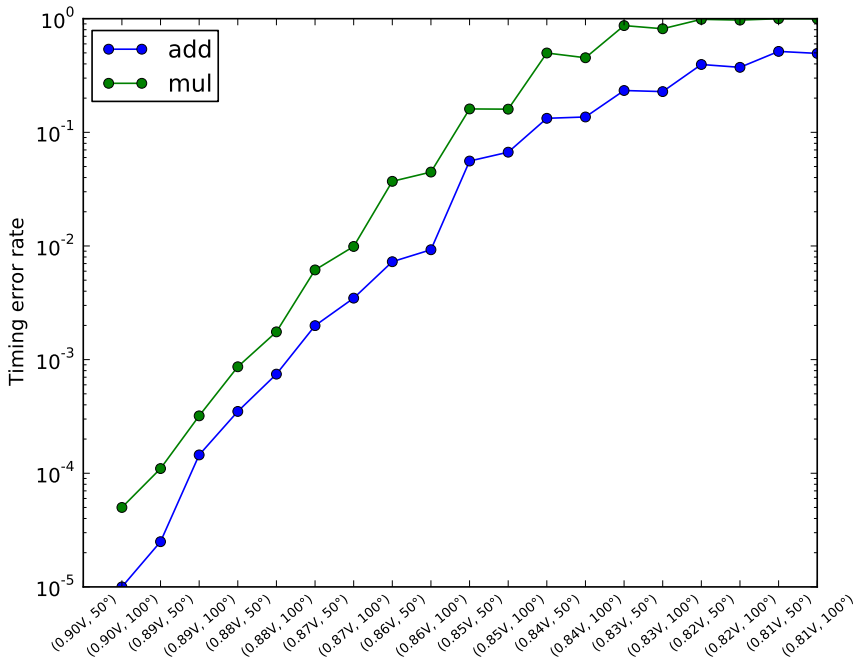

Figure 5.6. TER of adder and multiplier under different operating conditions.

errors contribute to most of the accuracy drop.

One main reason is that the accumulated convolution sum or dot-product sum are fed into a nonlinear activation function and thereby directly affect the activation, while the errors from multipliers will be averaged and diluted. This suggests that more hardware design effort should be made on the adder to ensure its low TER. On the other hand, the worst accuracy of both NN genres is around 10%, when either *add_only* or *mul_only* is 0.1. We can observe that such an accuracy drop starts saturating at 0.1 TER, almost identical to a random guess of the 10-class recognition task. In summary, such observations show that even though neural networks have inherent error resilience, the timing errors still can significantly affect neural network accuracy and motivate this work.

5.4.3 Accuracy Versus Dynamic Variations

We then use the real dynamic operating conditions to obtain realistic timing error rates and thereby characterize the vulnerability of HNNs to dynamic variations. Particularly, we use the *Timing Error Extraction* described in Section 5.3.1 to characterize the timing error behavior

Table 5.1. HNN accuracy under dynamic variations.

HNN	(0.90V, 50°C)	(0.90V, 100°C)	(0.89V, 50°C)	(0.89V, 100°C)	(0.88V, 50°C)	(0.88V, 100°C)
MLP	96.79%	96.03%	94.90%	87.93%	75.56%	57.76%
CNN	98.37%	97.31%	95.87%	85.15%	70.34%	48.64%
BCNN	99.58%	98.09%	97.49%	96.79%	82.65%	73.29%
LBPNet	99.52%	99.53%	99.49%	99.49%	99.48%	99.37%
HNN	(0.87V, 50°C)	(0.87V, 100°C)	(0.86V, 50°C)	(0.86V, 100°C)	(0.85V, 50°C)	(0.85V, 100°C)
MLP	25.67%	15.89%	10.45%	10.33%	9.42%	9.91%
CNN	18.85%	11.13%	9.81%	9.80%	9.81%	9.81%
BCNN	36.00%	10.52%	9.81%	9.83%	9.89%	10.02%
LBPNet	99.21%	98.20%	87.58%	50.98%	30.25%	11.03%
HNN	(0.84V, 50°C)	(0.84V, 100°C)	(0.83V, 50°C)	(0.83V, 100°C)	(0.82V, 50°C)	(0.82V, 100°C)
MLP	9.89%	9.80%	9.72%	9.60%	10.15%	9.60%
CNN	9.75%	9.81%	9.89%	9.80%	9.91%	9.84%
BCNN	9.99%	9.72%	9.90%	9.74%	9.99%	9.93%
LBPNet	11.72%	10.23%	10.56%	10.25%	10.19%	10.67%

of 32-bit floating point adder and multiplier under different operating conditions as shown in Figure 5.6. Besides the ideal condition without any error, the selected operating conditions cover a wide range of TERs: at the best condition (0.90V, 50°C) with TERs less than 0.0001; at the worst condition (0.81V, 50°C), 0.5 and 1.0 TER are found in adders and multipliers, respectively. By comparing these two computing units, we find that TER of the multiplier is always higher than the adder under the same condition. This is because the multiplier design has more critical paths than the adder, resulting in more timing violations. The TER of adder reaches 1% when the operating condition is around 0.86V. Based on Figure 5.4 and Figure 5.5, the accuracy drop starts to saturate when the TER of adder reaches 0.01; thus we expect to see the worst accuracy starting at around 0.86V.

We then present the accuracy of MLP, CNN, BCNN, and LBPNet under the twenty operating conditions, as shown in Figure 5.7 and Table. 5.1, where we observe several important facts:

1. First, the lowest accuracy under worst-case operating conditions is around 10% for all the four networks across multiple conditions from (0.85V, 100°C) to (0.81V, 100°C). (For better space utilization, we do not present the accuracy under 0.82V in Table. 5.1, where the accuracies are around 10%.) For MLP, CNN, and BCNN, this observation is expected

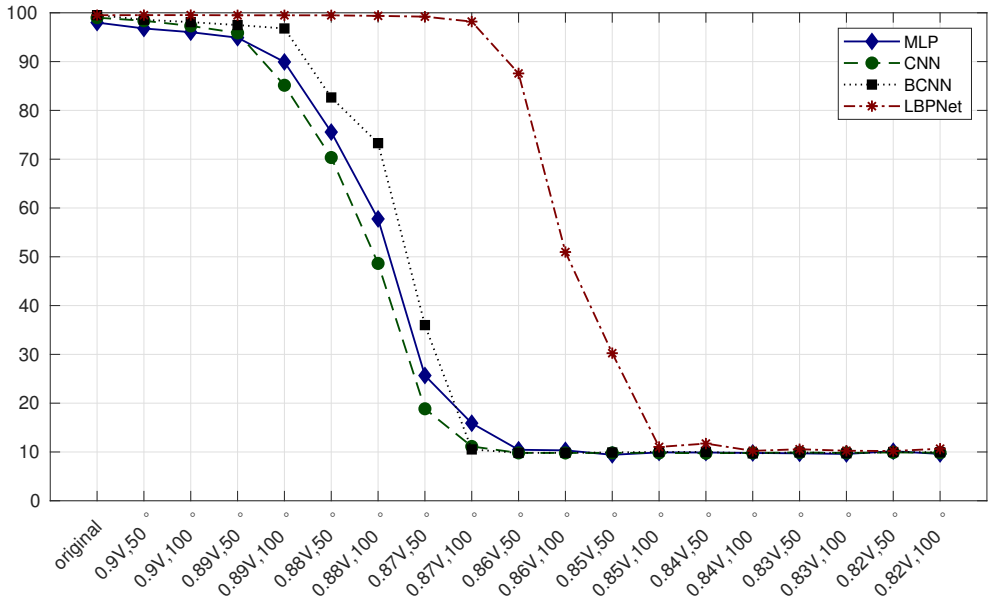

Figure 5.7. HNN accuracy as a function of dynamic variations.

as we can see from Figure 5.4 and Figure 5.5 where the accuracy drops to 10% when the TER of either unit reaches 0.01.

2. Second, the four curves can be categorized into two groups because the MLP, CNN, and BCNN behaviors similarly, and the LBPNet's accuracy curve demonstrate a high immunity to the TER residing in adders and multipliers.

3. Third, Table. 5.1 shows that under the condition between (0.90V, 50°C) and (0.86V, 50°C), where the TER of adder is less than 0.01, the accuracy drop of MLP to its original accuracy is less than that of CNN, indicating MLP might be more resilient than CNN within a certain TER. Part of the reason for this is that given the same TER, the amount of errors in CNN is larger than MLP because CNN has more arithmetic operations, and the percentile of multiplications among all arithmetic computations are higher in CNN.

4. Fourth, BCNN sustains slightly more timing errors than MLP and CNN because the binarized values and operations in BCNN rectify a portion of the injected errors and thereby enhances the robustness.

5. Fifth, LBPNet keeps immune against the variation until we impose much harsher conditions. A 10% accuracy deterioration is observed at (0.86V, 50°C), while all the other three models significantly lose classification ability and fall around 10% accuracy. LBPNet totally fails to classify upon (0.85V, 100°C), at with the TERs climb to 0.1 and 0.5 for adders and multipliers, respectively.

6. Last but not least, we find the voltage and temperature both play an important role in determining the inference accuracy. By fixing the temperature at 100°C, reducing the voltage by 0.01V from 0.89V to 0.88V results an accuracy drop of the CNN model from 85.15% to 48.64%; by fixing the voltage as 0.88V, increasing the temperature by 50°C results an accuracy drop from 70.34% to 48.64%. By comparing the accuracy at (0.90V, 50°C) and (0.86V, 50°C), we find the accuracy drops to worst case at around 10% from best case at around 98% by a voltage reduction of 0.04V.

The immunity of LBPNet outperforms the other models with a remarkable gap. There are multiple causes contribute to this immunity that can be qualitatively justified through a re-visit of the details in an LBP Layer. The comparison is simulated with the sign bit from the adder's subtraction output. Then, the sign bits corresponding to an LBP kernel are produced by adders in parallel and form a bit sequence to represent an integer on the output feature map. Whenever the adder is stochastically selected for an error injection, the sign bit is flipped randomly according to a uniform distribution. Therefore, on the one hand, an injected error can only affect a single bit rather than an output value as in MLP and CNN. Furthermore, if the selected bit is not the most significant bit (MSB) of the output value, the effect of error injection is scarce. On the other hand, the sign bits are combined with a bit shift and a logic OR operations in parallel, which are relatively less affected by the hardware variations given their circuitry simplicity and not discussed in the scope of this work. The absence of accumulation helps LBPNet to preclude the error accumulation and hinder the propagation of errors.

5.5 Discussion

Threats to Validity: In this work, we mainly focus on variation-induced timing errors in computation logic. However, the timing errors could also occur in control logic, which might lead to more severe accuracy drop or malfunction. Fortunately, it was observed that control logic only contributes a small set of critical paths [WDF+17], making it less vulnerable to timing errors.

Future Work: In this work, we focus on assessing the effects of hardware variations on neural network performance. The next question is how we can mitigate such timing errors. For future work, we focus on integrating the timing errors as a vector for backpropagation to enable an adaptive training method. Moreover, we plan to design a reconfigurable architecture that can automatically select suitable weights for a given voltage and temperature from a set of pre-stored weights.

5.6 Related Work

We describe the related work in three parts: combating timing errors, neural network resiliency and the main difference of our work with them.

Various hardware techniques have been developed to combat timing errors. Razor [ESKD+03], used a shadow flip-flop to detect timing errors and used recovery circuits to correct them. Error-detection sequential circuits (EDS) [BTK+09], double sampled and compared signals arriving at different timings through such flip-flops and then corrected them. Several learning methods were used to predict timing errors for functional units or instructions to enable an adaptive design [JRN+15, JJRG16, JJRG17]. A multi-armed bandit based optimization method was proposed to enable dynamic timing speculation [ZG17]. Going up to the system level, Bayesian networks have been used to calculate the system reliability with both hardware and software in consideration and acquired higher accuracy [JZS+13, JZL+14].

More recent approaches to improving cost and energy efficiency have advocated tolerance

to computational approximations, such as approximate adders [CSE16, JCC+17]. These errors, originating from the inexact logic design of computing units, have been used in hardware neural networks to improve operational efficiency [DLC+15, MSS+16, ZWT+15]. [DLC+15] substituted the normal multipliers with inexact multipliers that provide inexact logic but with less hardware cost. [MSS+16] further optimized such design with a uniform structure suitable for hardware implementation. [ZWT+15] provided a framework for hardware neural network designers to choose which parts were suitable for an approximation that led to less impact on accuracy based on a criticality ranking. These works intentionally designed inexact hardware and introduced logic errors in exchange for less hardware cost.

Compared to logic errors, timing errors were less exploited in neural networks because of its unpredictability and uncertainty. Logic errors could be determined once the design is fixed but timing errors can only be obtained through simulations. A retraining-based method has been proposed to mitigate the timing errors in hardware neural networks [WDF+17]. However, these works assumed a fixed timing variation for each gate without considering hardware variations as the root cause, which might be unrealistic.

In summary, there have been no prior works assessing the neural network vulnerability to dynamic variations. In this work, we do not introduce the errors intentionally but focus on the unintentional timing errors caused by hardware variations. We link the timing errors with low-level hardware variations and characterize them under different operating conditions and present the importance of considering variations when designing hardware neural networks.

5.7 Conclusions

In this chapter, we assess the effects of dynamic voltage and temperature variations on the performance of hardware neural networks. We first extract the timing errors of post place-and-route computation units under twenty operating conditions through gate-level simulations. We then inject such errors to the neural network inference phase and evaluate the resulted accuracy.

With the results on MLP, CNN, BCNN, and LBPNet, we demonstrate that dynamic voltage and temperature variations can cause a significant drop in inference accuracy. The variation immunity of LBPNet is the highest among the tested models and hence and sustain more tough conditions in practical applications.

Chapter 5 contains the re-organized reprints of Xun Jiao, Mulong Luo, Jeng-Hau Lin, Rajesh K. Gupta. “An Assessment of Vulnerability of Hardware Neural Networks to Dynamic Voltage and Temperature Variations”. *I*n Proceedings of the 36th International Conference on Computer-Aided Design (ICCAD). IEEE Press, 2017. p. 945-950. The dissertation author is the co-author and primary investigator of this paper.

Chapter 6

Conclusion and Future Directions

In this dissertation, we have dove into the core of algorithmic demands to pursue a cross-layer optimization for deep learning application. Particularly, we developed energy efficient algorithms, implemented GPU training programs, and finally carried out the trained models on software-programmable FPGAs. Two types of neural networks, BCNNw/SF and LBPNet, have been proposed to loosen the strain of physical resource deficiency. Since the hardware incarnation of neural networks in the practical application can never be isolated from hardware variations, we also extracted timing errors from gate level simulation and injected errors in neural network models and discovered the high noise immunity of LBPNet. The efficiency in terms of computation resource and energy consumption and effectiveness in error immunity can benefit and reduce the guardbands in a system.

Looking beyond this dissertation, several promising directions can be further explored in the future:

Fast Object Detection. The object detection task requires a fast and robust method to produce candidate positions that possibly contain the targets. Given that LBPNet is strong and swift in answering 'Yes'-or-'No' questions, we can combine LBPNet into objection detection methods, such as YOLO or fastRCNN.

Improving the pre-trained LBP Kernels with Data-Driven Methods. Since the result of LBPNet's training is the sparse and discrete sampling positions, we can apply the Monte-Carlo

method to the sampling positions for a better leverage of the input data. However, directly swap sampling points with a heuristic or stochastic method, such as the simulated annealing algorithm, may lead to suboptimal results. A possible combination is to perform Monte-Carlo after the forward-backward algorithm on LBPNets.

Aggregation in LBPNet. The disadvantage of LBPNet is the absence of information aggregation. If a small number of additions are allowed in the LBPNet, the information or prior distributions of input images can be accumulated and propagated across LBPLayers. Although introducing aggregation in LBPLayer increases the complexity of backward propagation, the variety of kernels may benefit the higher abstract feature extraction.

Appendices

We include LBPNet's details of implementation and analysis in this section.

A.1 Examples of Feature Extraction

Figure A.1.1 shows five examples of the input and output of a trained LBP layer. The first three rows are images of label '4', and we see that strokes in the same relative position get enhanced. For the other images with different labels, the trained LBP is also able to enhance different portions, and the resulting output maps include more distinguishable features for classification.

A.2 Learning Curves

Figure A.2.2 shows the learning curves of LBPNets on MNIST and SVHN.

A.3 Sensitivity Analysis of the Scaling Parameter

Figure A.3.3 shows the sensitivity analysis of the parameter α in Eq. 3.1 w.r.t. the training accuracy. The LBPNet structure we use is 3-layer, 39-40-80. We gradually reduce α from 10 to 0.01 to verify the effect on the learning curves. Sub-figure (a) and (c) show that the smaller α is, the lower the error rate is, but it saturates when α is below 1. Sub-figure (b) shows that a smaller α can better reduce the variance of training loss. As a summary, because we approximate the comparison function with a shifted and scaled hyperbolic tangent function. A smaller α implies lower error between the approximation and the original comparison curve, hence simulating

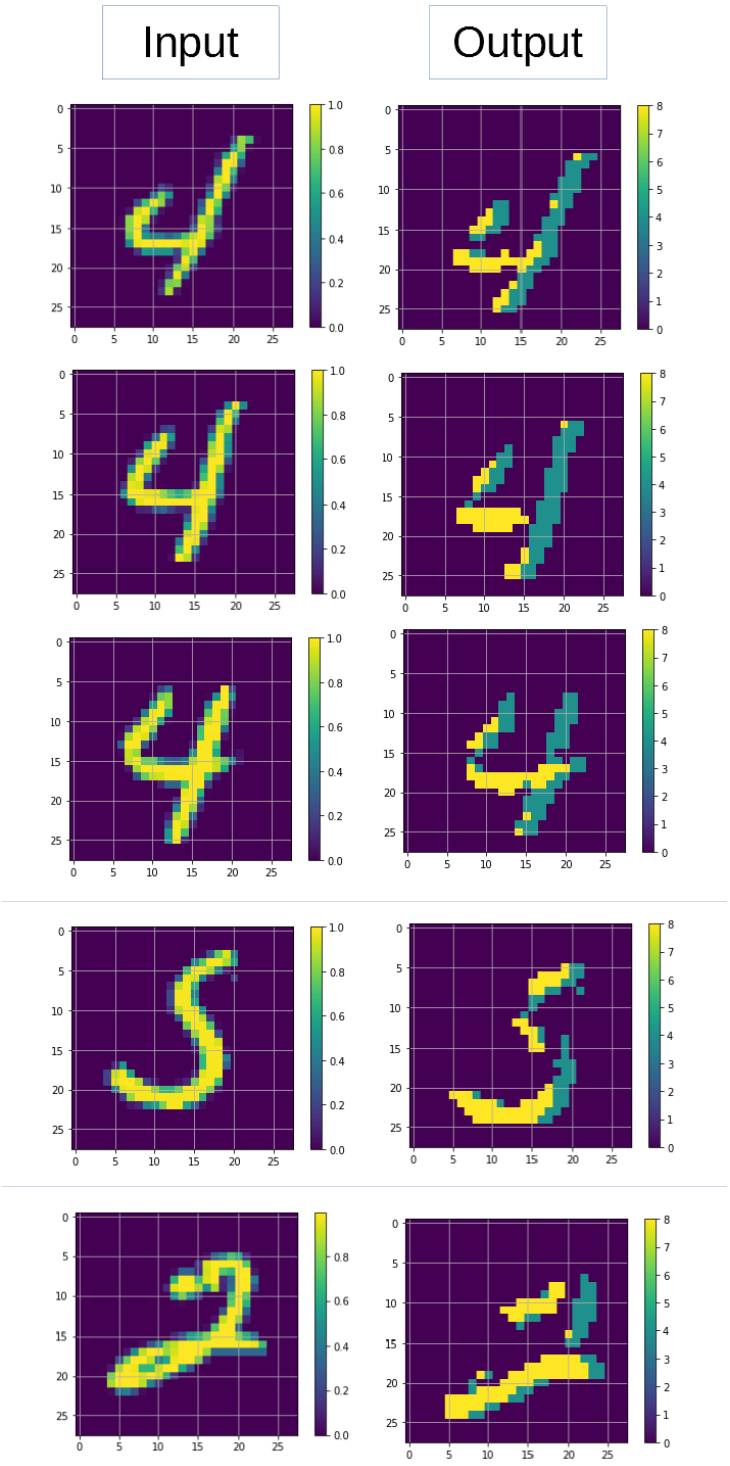

Figure A.1.1. The input images and output features maps of a LBP Layer.

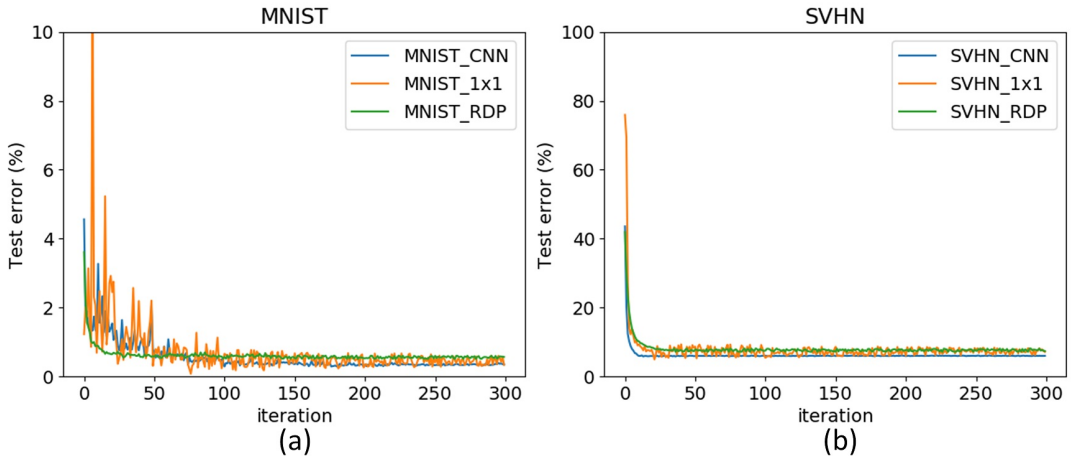

Figure A.2.2. Error curves of LBPNets on benchmark datasets: (a) test errors on MNIST; (b) test errors on SVHN.

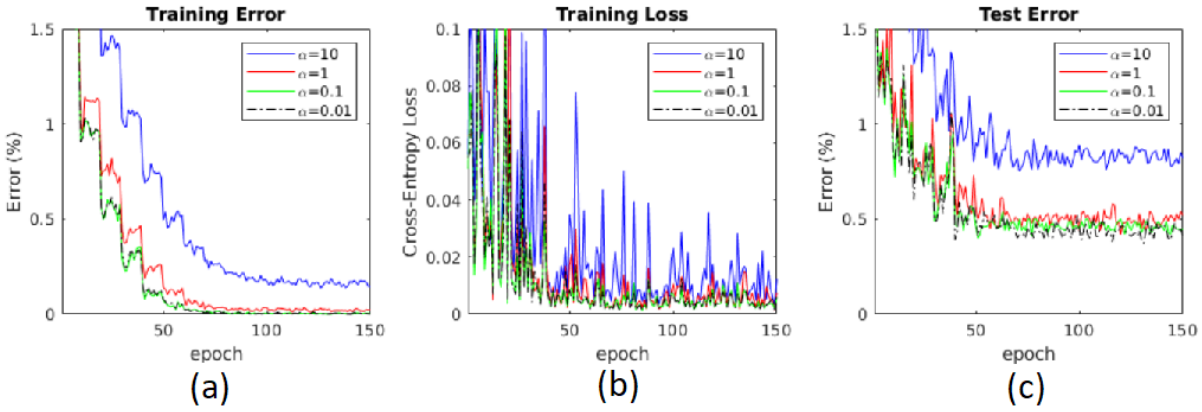

Figure A.3.3. Sensitivity analysis of α w.r.t. training error on MNIST. (a) Training error; (b) training loss; (c) test error.

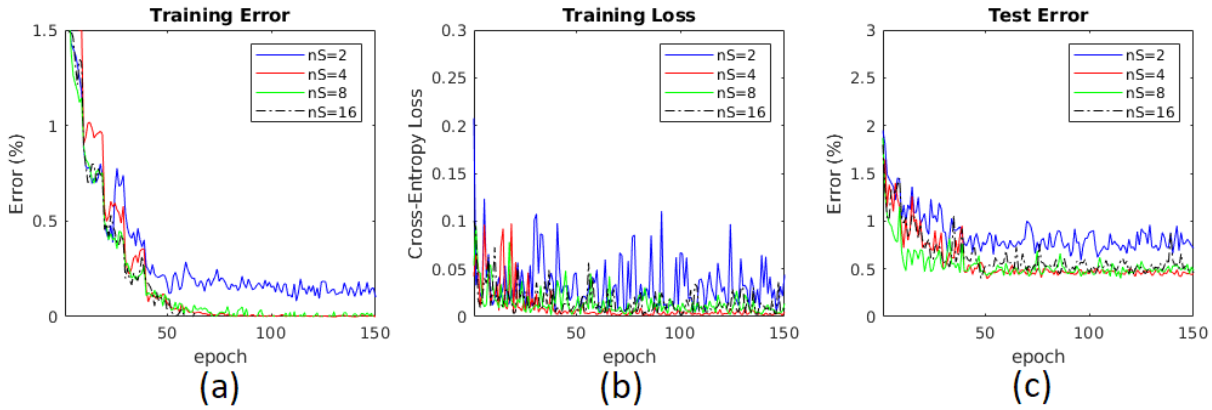

Figure A.4.4. The effect of the number of sampling points on training error on MNIST: (a) Training error; (b) training loss; (c) test error.

the comparison while securing differentiability. In this paper, we choose $\alpha = 0.1$ to balance between classification accuracy and the overflow risk of the gradient summation during backward propagation.

A.4 Sensitivity Analysis of #Sampling Point

Figure A.4.4 shows how the number of sampling points nS affects the training accuracy. Since there is no significant improvement after $nS >= 4$, we choose to use 4 sampling points per kernel to save the memory and computation resource.

A.5 SVHN Results

Table A.5.1 shows the experimental results of LBPNet on SVHN together with the baseline and previous works. The CNN-baseline and LBPNet (RP) share the same network structure, 37-40-80-80-160-160-320-320, and the CNN-lite is limited to the same memory size so that the network structure is 13-20. BCNN-6L outperforms our baseline and achieves 2.53% with smaller memory footprint and higher speed. LBCNN-40L also achieves good memory reduction, but loses the speedup since it stacks 40 building blocks. The 20-layer LBPNet (1x1) with 40 LBP kernels and 32 1-by-1 convolutional kernels achieve 8.33%. The convolution-free LBPNet

Table A.5.1. The performance of LBPNet on SVHN.

	Error ↓	Size ↓ (Bytes)	Latency ↓ (GOPs)	Operations ↑
MLP Block	77.78%	-	-	-
CNN-baseline	6.69%	10.11M	1.86	1X
CNN-lite	54.40%	10.90K	0.003	598X
BCNN-6L	2.53%	153.8K	0.306	6.089X
BCNN-6L-noBN	80.41%	146.6K	0.306	6.094X
BCNN-8L-noBN	79.92%	46.42K	0.632	2.947X
LBCNN-40L	5.50%	6.70M	6.088G	0.306X
LBCNN-40L-noBN	80.41%	1.7M	6.087G	0.306X
LBCNN-8L-noBN	80.41%	461.81K	1.281	1.529X
LBPNet (this work)				
LBPNet (1x1)	8.33%	5.35M	0.058	32X
LBPNet (RP)	7.10%	10.62K	0.010	193X

Table A.6.2. The error rates of fixed LBP kernels. From left to right, the number of fixed LBP layers are increased from the first layer to the last layer.

Dataset	0	1	2	3	4	5	6	7	8
MNIST	0.50	1.70	5.51	15.84	-	-	-	-	-
SVHN	7.10	32.94	41.76	53.97	58.25	59.30	61.02	65.62	66.16

(RP) for SVHN is built with 8 layers of LBP basic blocks, 37-40-80-80-160-160-320-320, and achieves 7.10% error rate. Compared with CNN-lite's high error rate, the learning of LBPNet's sampling point positions proves to be effective and economical. The error rates of the remaining four models, BCNN-6L-noBN, BCNN-8L-noBN, LBCNN-40L-noBN, and LBCNN-8L-noBN, are similar to random guess, 80% (Note that the SVHN dataset is not balanced). However, LBPNet (RP) reaches the same level of error rate as the CNN-baseline, yet with 10.62KB model size, which is 952x smaller.

A.6 LBPNet Results When the Patterns Are Fixed upon Initialization

Table A.6.2 lists the error rates when part of the LBPNets are not optimized. The increase of error rates as more layers are fixed demonstrates the effectiveness of SGD optimization.

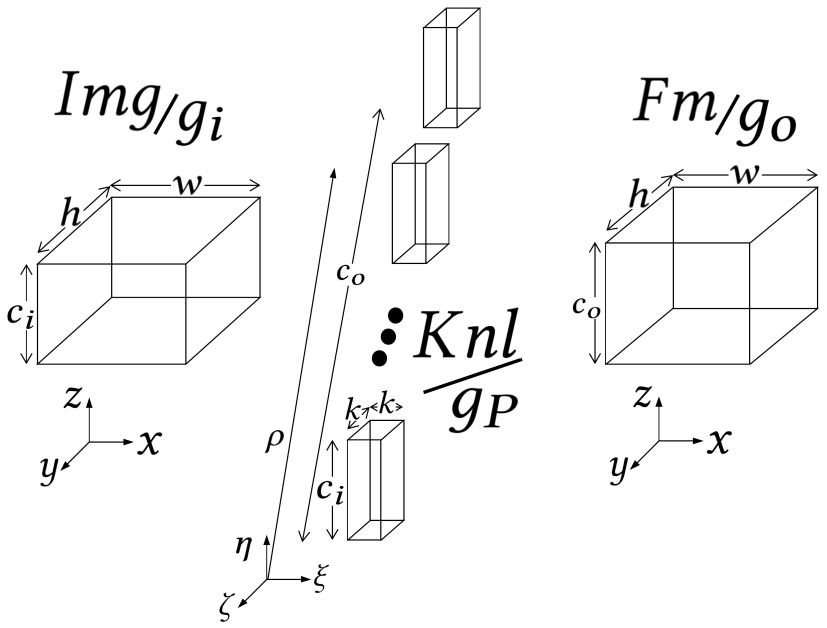

Figure A.7.5. The axes and dimensions for the six tensors in the following sections. Img, Fm, g_i, and g_o share the same coordination, while Knl shares the same coordination with g_P. g_i's shape of is (c_i, h, w), Knl's shape is (c_o, c_i, k, k), and Fm's shape is (c_o, h, w). Img, Fm, and Knl are in the same sizes with g_i, g_o, and g_P, respectively.

A.7 Detailed Description of the LBPNet Algorithm

The training of CNN includes forward, backward propagation, and parameter update. The forward propagation defines how to generate the feature maps. The backward propagation comprises the calculations of two derivatives: the gradient with respect to the input $\frac{\partial \text{loss}}{\partial \text{input}}$ (or g_i) and the gradient with respect to parameter $\frac{\partial \text{loss}}{\partial \text{parameter}}$ (or g_P). The parameters are updated using a convex optimization method. We, first, introduce the training algorithm of a convolutional layer (ConvLayer) and then explain the optimized GPU algorithm for CNN.

A.7.1 Training algorithm of CNN

All the axes and dimensions in the following equations and algorithms are illustrated in Figure A.7.5.

Forward Propagation. As shown in Eq. (1), the spatial non-inverted discrete convolution is composed of three nested summations. Although there are three nested loops, the kernels are moving spatially, i.e., along x- and y-axes, because both the input image and the convolutional kernels have the same number of input channels.

$$Fm[x][y][z] = \sum_{\zeta=1}^{c_i} \sum_{\eta=1}^{k} \sum_{\xi=1}^{k} Knl[\xi][\eta][\zeta][z] Img[x+\xi][y+\eta][\zeta], \tag{1}$$

where Fm is an output feature map, (x,y) denotes a spatial position on fm, z is the index of a concerned output channel, Knl is the four dimensional convolutional kernel, (ξ,η) is a dummy tuple denoting a spatial position on Knl of size c_o-by-c_i-by-k-by-k, ζ is the dummy variable iterating over all input channels, (ξ,η) denotes a spatial position on knl, and Img is the input image. The subject of the summations is the dot-product between the input image and the convolution kernel.

Algorithm 1 describes the forward propagation algorithm of a ConvLayer, which is composed of dot-product and sliding window operations, where $\langle\rangle$ stands for the element-wise multiplication. The two innermost loops describe the sliding window operation, and the only equation inside is a dot-product between a kernel and a portion of the input feature map as shown in Eq. (1). All kernels are involved with the same convolution operation as described as the outermost loop.

Backprop of CNN w.r.t. Input. Eq. (2) shows the backward propagation for the preceding layer.

$$g_i[x][y][z] = \sum_{\rho=1}^{c_o} \sum_{\eta=1}^{k} \sum_{\xi=1}^{k} Knl[\xi][\eta][z][\rho] g_o[x+\xi][y+\eta][\rho], \tag{2}$$

where g_i is the gradient of loss with respect to an input image, (x,y) denotes a spatial position on g_i, z is the index of a concerned input channel, (ξ,η) denotes a dummy spatial position on Knl, ρ is a dummy iterator over the output channel, and g_o is the gradient of loss with respect to an

Algorithm 1: Forward of CNN

input : An input tensor Img of shape (c_i, h, w), previous kernel Knl of shape (c_o, c_i, k,k), and the padding width $d = \lfloor \frac{k}{2} \rfloor$.
output : An output tensor Fm of shape (c_o, h, w).

$ImgP \leftarrow \text{ZeroPadding}(Img, d)$
for $z = 1$ *to* c_o **do**
  for $y = 1$ *to* h **do**
    for $x =$ *to* w **do**
      $Fm[x][y][z] = \langle Knl[1:k][1:k][1:c_i][c_o], ImgP[x:x+k-1][y:y+k-1][1:c_i] \rangle$
    end
  end
end
return Fm

output image.

Algorithm 2 describes the algorithm of CNN backward propagation to calculate gradient w.r.t. input. Analogous to the forward propagation, the two innermost loops perform the sliding window on the g_o to collect the supervised learning errors, which are weighted with the convolutional kernel Knl, as shown in Eq. (1). The outermost loop iterates over the input channels to build g_i.

Algorithm 2: Backprop of CNN: Gradient w.r.t. input

input : An output gradient tensor g_o of shape (c_o, h, w) and previous kernel W of shape (c_o, c_i, k,k).
output : An input gradient tensor g_i of shape (c_i, w, h).

for $z = 1$ *to* c_i **do**
  for $y = 1$ *to* h **do**
    for $x =$ *to* w **do**
      $g_i[x][y][z] = \langle Knl[1:k][1:k][z][1:c_o], g_o[x:x+k-1][y:y+k-1][1:c_o] \rangle$
    end
  end
end
return g_i

Backprop of CNN w.r.t. Parameter. Eq. (3) shows the backward propagation for learnable parameters.

$$g_P[\xi][\eta][\zeta][\rho] = \sum_{x=1}^{h_o} \sum_{y=1}^{h_o} Img[\xi + x][\eta + y][z] g_o[x][y][\rho], \tag{3}$$

where, g_P is the gradient with respect to the learnable parameters, (x, y, z, d) is the 4-ary tuple in format of (column index, row index, input channel index, output channel index), and (ξ, η) is the dummy tuple denoting for the spatial position on g_o. Algorithm 3 describes the algorithm of CNN backward propagation to calculate gradient w.r.t. parameter.

Algorithm 3: Backprop of CNN: Gradient w.r.t. parameters

input : An output gradient tensor g_o of shape (c_o, h, w) and an input feature map Img of shape (c_i, h,w).

output : A parameter gradient tensor g_P of shape (c_o, c_i k, k).

$ImgP \leftarrow \text{ZeroPadding}(Img, d)$
for $\rho = 1$ *to* c_o **do**
  for $\zeta = 1$ *to* c_i **do**
    for $\eta = 1$ *to* k **do**
      for $\xi = 1$ *to* k **do**
        $g_P[\xi][\eta][\zeta][\rho] = \langle Go[1:w][1:h][\rho], ImgP[\xi:\xi+w-1][\eta:\eta+h-1][\zeta] \rangle$
      end
    end
  end
end
return g_P

A.7.2 GPU-accelerated CNN

The three algorithms in the previous section contain loops, which can be replaced entirely with GPU supported primitive functions in the basic linear algebra subprogram (BLAS) library shown in the following list.

- **GEMV**: General 2-D matrix to 1-D vector multiplication.
- **GEMM**: General 2-D matrices multiplication.
- **im2col**: Converting a 2-D matrix into another 2-D matrix simulating the sliding window operation, a.k.a. the Toeplitz matrix format.
- **col2im**: Accumulating the Toeplitz matrix format back to an image. col2im equivalently performs local window summations.

Since GEMV and GEMM carry out the matrix arithmetic in linear algebra with hardware support primitives, we explain the rest two functions in this section.

Image to Toeplitz Matrix Format Conversion. The im2col function trades redundant memory footprint for speedup due to the elimination of loops. Algorithm 4 describes the parallel access and allocation happening in every processing element.

Algorithm 4: im2col

input : A 2-D matrix A of shape (h, w) and the a window size (k, h).
output : Another 2-D matrix B in the Toeplitz matrix format of shape (k^2, $(h-k+1)(w-k+1)$).

/* Invoke k^2-by-$(h-k+1)(w-k+1)$ parallel processing units for the following instructions. */
$nc = w-k+1$
$B[x][y] = A[y\%nc+x\%k][\lfloor y/nc \rfloor + \lfloor x/k \rfloor]$

Toeplitz Matrix Format to Image Conversion. The col2im function accumulates all local elements covered under a window into a value and allocates the value to the image format. Algorithm 5 describes the parallel operations happening in every processing element. The two summations are hardware supported and can be achieved in parallel, respectively.

Algorithm 5: col2im

input : The shape of the output image (h, w), and the a window size (k, h), and a 2-D matrix B in the Toeplitz matrix format of shape (k^2, $(h-k+1)(w-k+1)$).
output : Another 2-D matrix A of shape (h, w).

/* Invoke h-by-w parallel processing units for the following instructions. */
$A[x][y] = \sum_{\eta=1}^{k} \sum_{\xi=1}^{k} B[(k+1-\eta)k+(k+1-\xi)][(y-k+\eta)w+(x-k+\xi)]$

GPU-accelerated ConvLayer With the four BLAS primitives, Algorithm 1, 2, and 3 can be rewritten as loop-free Algorithm 6, 7, and 8.

The temporary matrix *Clns* denotes the resultant of *im2col* conversion, and the shape of *Clns* is $(hw, c_i k^2)$.

A.7.3 LBPNet

The forward and backward propagations of LBPNet are illustrated in the main paper with high-level descriptions. In this section, we formulate the descriptions into Algorithm 9 and

Algorithm 6: Loop-free Forward Propagation of CNN

input : An input tensor Img of shape (c_i, h, w), the previous kernel Knl of shape (c_o, c_i, k,k) and the padding width $d = \lfloor \frac{k}{2} \rfloor$.

output : An output tensor Fm of shape (c_o, h, w).

$ImgP \leftarrow \text{ZeroPadding}(Img, d)$
$ImgP \leftarrow \text{Reshape}(ImgP, (c_i, (h+2d)(w+2d)))$
$Knl \leftarrow \text{Reshape}(Knl, (c_o, c_i k^2))$
$Clns \leftarrow im2col(ImgP, ((h+2d), (w+2d)), (k,k))$
$Fm \leftarrow GEMM(Knl, Clns^T)$
$Fm \leftarrow \text{Reshape}(Fm, (c_o, h, w))$
return Fm

Algorithm 7: Loop-free Backprop of CNN: Gradient w.r.t. input

input : An output gradient tensor g_o of shape (c_o, h, w) and the previous kernel W of shape (c_o, c_i, k,k).

output : An input gradient tensor g_i of shape (c_i, w, h).

$g_o \leftarrow \text{Reshape}(g_o, (c_o, hw))$
$Knl \leftarrow \text{Reshape}(Knl, (c_o, c_i k^2))$
$Clns \leftarrow GEMM(g_o^T, Knl)$
$g_i \leftarrow col2im(Clns, (h,w), (k,k))$
$g_i \leftarrow \text{Reshape}(g_i, (c_i, h, w))$
return g_i

a set of equations in Section 17 first, and then propose a loop-free algorithm, GPU-LBPNet, leveraging the parallelism of GPU to accelerate both the forward and backward propagations.

Forward Propagation of LBPNet

The forward propagation is designed to be multiplication and accumulation free (MAC-free). The LBP operation [OPH96] and random projection [BM01] compose LBPNet's forward

Algorithm 8: Loop-free Backprop of CNN: Gradient w.r.t. parameters

input : An output gradient tensor g_o of shape (c_o, h, w) and the input feature map Img of shape (c_i, h,w).

output : A parameter gradient tensor g_P of shape (c_o, c_i k, k).

$ImgP \leftarrow \text{ZeroPadding}(Img, d)$
$ImgP \leftarrow \text{Reshape}(ImgP, (c_i, (h+2d)(w+2d)))$
$Clns \leftarrow im2col(ImgP, ((h+2d), (w+2d)), (k,k))$
$g_P \leftarrow GEMM(Clns^T, g_o^T)$
$g_P \leftarrow \text{Reshape}(g_P, (c_o, c_i, k, k))$
return g_P

propagation. We can implement the two ideas together to avoid redundant memory accesses and comparisons if we loop up the projection map before comparison.

Algorithm 9: Forward of LBPNet

input : An input tensor Img of shape (c_i, h, w), the LBP kernel P of shape (c_o, n_s), and the fixed projection map M of shape (c_o, n_s). The pattern width k and padding width $d = \lfloor \frac{k}{2} \rfloor$. Please note that every element of P is a tuple denoting a spatial position on the sliding window.

output : An output feature map Fm.

```
1  ImgP ← ZeroPadding(Img, d)
2  for z = 1 to c_o do
3      for y =  to h do
4          for x = 1 to w do
5              for i = 1 to n_s do
6                  ζ = M[z, i]
7                  (ξ, η) = P[z, i]
8                  I_pivot_i = ImgP[x + d][y + d][ζ]
9                  I_lbp_i = ImgP[x + ξ][y + η][ζ]
10                 if I_lbp_i > I_pivot_i then
11                     Fm[x][y][z] = 1 ≪ i_s
12                 end
13             end
14         end
15     end
16 end
17 return Fm
```

Algorithm 9 summarizes the forward algorithm of an LBP layer. The three outermost nested loops form the sliding window operation to generate an output feature maps, and the innermost loop is the LBP operation. The core of LBPNet is implemented with bit shifting and bitwise-OR as shown in the if-condition block of Algorithm 9 or the exponentially weighted summation in Eq. 4.

$$Fm[x][y][z] = \sum_{i=1}^{n_s} [2^{i-1}(I_{lbp_i} > I_{pivot_i})], \tag{4}$$

where n_s is the number of samplings in an LBP pattern, I_{lbp_i} is the i-th sampled pixel value as shown in line 9 of Algorithm 9 and I_{pivot_i} is the center pixel value in the window in line 8 of Algorithm 9. The exponential term 2^{i-1} simulates the bit allocation of the comparison result. Please note that the design goal of LBPNet is to leverage bit-wise operation in edge devices, the

accumulation in Eq. 4 is a mathematically equivalent form to the bit allocation for the derivation of backward propagation. While designing hardware accelerators on FPGA, the bitwise-OR can be implemented in parallel with proper block ram partition.

Backward Propagation of LBPNet

As shown in Eq. 3.1, an approximation from comparison to a scaled and shifted hyperbolic tangent function is adopted to make the LBPNet trainable with convex optimization methods.

Substituting Eq. 3.1 into Eq. 4, we can get Eq. 5.

$$Fm[x][y][z] = \sum_{i=1}^{n_s} \left[2^{i-2} (tanh(\frac{I_{lbp_i} - I_{pivot_i}}{\alpha}) + 1) \right], \tag{5}$$

Eq. 6 is the calculation of the gradient of loss with respect to an input image.

$$g_i[x][y][z] = \sum_{\rho=1}^{c_o} \sum_{\eta=1}^{k} \sum_{\xi=1}^{k} (g_o[x+\xi][y+\eta][\rho] \frac{\partial Fm[x][y][\rho]}{\partial I_{lbp_i}}), \tag{6}$$

where $\frac{\partial Fm[x][y][\rho]}{\partial I_{lbp_i}}$ is shown in Eq. 7.

$$\frac{\partial Fm[x][y][\rho]}{\partial I_{lbp_i}} = \frac{2^{i-2}}{\alpha} \left[1 - tanh^2(\frac{I_{lbp_i} - I_{pivot_i}}{\alpha}) \right] \tag{7}$$

Eq. 8 shows the calculation of the gradient of loss with respect to the learnable sampling position.

$$\begin{aligned} &g_P[\xi][\eta][\zeta][\rho] = \\ &\sum_{y=1}^{h_o} \sum_{x=1}^{w_o} \left[g_o[x][y][\rho] \frac{\partial Fm[x][y][\rho]}{\partial I_{lbp_i}} (\frac{\partial I_{lbp_i}}{\partial x} \hat{x} + \frac{\partial I_{lbp_i}}{\partial y} \hat{y}) \right], \end{aligned} \tag{8}$$

where g_o is the backward propagated error, $\frac{\partial s}{\partial I_{lbp_i}}$ is the gradient of an output pixel on Fm with respect to a sampled input pixel, and $(\hat{x}, \hat{y})$ denotes the unit vector of the spatial axes. The last

Algorithm 10: Loop-free Backprop of LBPNet: Gradient w.r.t. input

input : An output gradient tensor g_o of shape (c_o, h, w), the input image Img of shape (c_i, h, w), the previous discrete kernel P, and the fixed random projection map M. Both P and M are in the same shape (c_o, n_s), but every element in P is a tuple of sampling position in a sliding window. The pattern width k and padding width $d = \lfloor \frac{k}{2} \rfloor$.

output : An input gradient tensor g_i of shape (c_i, w, h).

$ImgP \leftarrow \text{ZeroPadding}(Img, d)$
$g_o \leftarrow \text{Reshape}(g_o, (c_o, hw))$
$Th \leftarrow BuildTanhI(ImgP, M, P, k, d)$
$Th \leftarrow \text{Reshape}(Th, (c_o, c_i k^2 hw))$
$tGo \leftarrow \text{tile}(g_o, (c_i k^2, 1))$
$preClns \leftarrow Th \bigstar tGo$
$Clns \leftarrow GEMV(preClns^T, onesI)$
$g_i \leftarrow col2im(Clns, (h, w), (k, k))$
$g_i \leftarrow \text{Reshape}(g_i, (c_i, h, w))$
return g_i

term of Eq. 8 ($\frac{\mathbf{d}I_{lbp_i}}{\mathbf{d}x}\hat{x} + \frac{\mathbf{d}I_{lbp_i}}{\mathbf{d}y}\hat{y}$) is the image gradient of Img at the sampling position.

A.7.4 GPU-Accelerated Forward propagation of LBPNet

Because the forward propagation of LBPNet shown in Algorithm 9 involves bitwise operation and allocation, it cannot be implemented efficiently with the four primitives mentioned in Section A.7.2. However, we can avoid the sliding window operation through the low-level CUDA programming to define each processing element, i.e., CUDA core, to perform the innermost loop in Algorithm 9, and thereby generate the feature map. The resultant algorithm is the innermost loop of Algorithm 9.

GPU-Accelerated Backprop of LBPNet w.r.t. Input

We propose Algorithm 10, which is a loop-free algorithm, to accelerate the calculation of g_i.

$onesI$ is a constant vector of length c_o. The $BuildTanhI(.)$ function implementing Eq. 7 converts the sparse and irregular LBP kernel into a regular dense intermediate matrix Th of shape $(c_o, c_i k^2 hw)$ as shown in Algorithm 11.

An Even Aggressive Approximation. Since an LBP layer works similarly with a

Algorithm 11: BuildTanhI

input : The input image Img of shape (c_i, h, w), the previous discrete kernel P, and the fixed random projection map M. The pattern width k and the padding width $d = \lfloor \frac{k}{2} \rfloor$.

output : A rank 6 tensor Th of shape (c_o, c_i, k, k, h, w)).

/* Invoke c_o-by-n_s-by-h-by-w parallel processing units for the following instructions. */

$\zeta = M[\rho, i]$

$(\xi, \eta) = P[\rho, i]$

$I_{pivot_i} = ImgP[x+d][y+d][\zeta]$

$I_{lbp_i} = ImgP[x+\xi][y+\eta][\zeta]$

$Th[x][y][\xi][\eta][\zeta][\rho] = \frac{2^i}{4k} \left[1 - tanh^2(\frac{I_{lbp_i} - I_{pivot_i}}{k})\right]$

pooling layer for the sampling apertures, we can bypass the calculation of g_i and merely forward the errors to those input pixels which are sampled and used. We propose an even aggressive approximation of LBPNet's training, which is to skip the calculation of g_i. We name this method GPU-LBPNet(P).

GPU-Accelerated Backprop of LBPNet w.r.t. Parameter

Algorithm 12 shows a loop-free algorithm accelerating the calculation of g_P.

Algorithm 12: Loop-free Backprop of LBPNet: Gradient w.r.t. parameter

input : An output gradient tensor g_o of shape (c_o, h, w), an input feature map Img of shape (c_i, h,w), the previous discrete kernel P, and the fixed random projection map M. The pattern width k and the padding width $d = \lfloor \frac{k}{2} \rfloor$.

output : An parameter gradient tensor g_P of shape (c_o, n_s), while every element is a tuple of a force vector to push the sampling vector for a convex optimization method.

$ImgP \leftarrow \text{ZeroPadding}(Img, d)$

$ImgGrad \leftarrow \text{CalcImgGrad}(ImgP)$

$g_o \leftarrow \text{Reshape}(g_o, (c_o, hw))$

$tGo2 \leftarrow \text{tile}(g_o, (c_i n_s, 1))$

$(Th_x, Th_y) \leftarrow BuildTanhP(ImgP, ImgGrad, M, P)$

$Th_x \leftarrow \text{Reshape}(Th2_x, (c_o, c_i n_s hw))$

$Th_y \leftarrow \text{Reshape}(Th2_y, (c_o, c_i n_s hw))$

$preClns_x \leftarrow Th_x \bigstar tGo2$

$preClns_y \leftarrow Th_y \bigstar tGo2$

$g_P \leftarrow \text{pair}(GEMV(preClns_x^T, onesP), GEMV(preClns_y^T, onesP))$

$g_P \leftarrow \text{Reshape}(g_P, (c_o, c_i, k, k))$

return g_P

The subroutine $BuildTanhP(.)$ is similar to $BuildTanhI$ except for its element-wisely multiplication of the resultant tensor and the image gradient, in which every element is a tuple of

the x-gradient and y-gradient. *onesP* is a constant vector of length *hw*.

Bibliography

[AGÁGSM18] Álvaro Arcos-García, Juan A Álvarez-García, and Luis M Soria-Morillo. Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods. *Neural Networks*, 99:158–165, 2018.

[AP16] Jose Alvarez and Lars Petersson. DecomposeMe: Simplifying ConvNets for End-to-End Learning. *arXiv e-print*, arXiv:1606.05426, Jun 2016.

[APG15] Shailesh Acharya, Ashok Kumar Pant, and Prashnna Kumar Gyawali. Deep learning based large scale handwritten devanagari character recognition. In *SKIMA*, pages 1–6. IEEE, 2015.

[AYS+18] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K Gupta, and Hadi Esmaeilzadeh. Snapea: Predictive early activation for reducing computation in deep convolutional neural networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 662–673. IEEE, 2018.

[BFB94] John L Barron, David J Fleet, and Steven S Beauchemin. Performance of optical flow techniques. *International journal of computer vision*, 12(1):43–77, 1994.

[BM01] Ella Bingham and Heikki Mannila. Random projection in dimensionality reduction: applications to image and text data. In *ACM SIGKDD*, 2001.

[BMP02] Serge Belongie, Jitendra Malik, and Jan Puzicha. Shape matching and object recognition using shape contexts. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 4:509–522, 2002.

[BTK+09] Keith A Bowman, James W Tschanz, Nam Sung Kim, Janice C Lee, Chris B Wilkerson, Shih-Lien L Lu, Tanay Karnik, and Vivek K De. Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance. *IEEE Journal of Solid-State Circuits*, 44(1):49–63, 2009.

[CBD15] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. *Advances in Neural Information Processing Systems (NIPS)*, pages 3123–3131, 2015.

[CDB14] Matthieu Courbariaux, Jean-Pierre David, and Yoshua Bengio. Low precision storage for deep learning. *arXiv preprint arXiv:1412.7024*, 2014.

[CDS+14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, volume 49, pages 269–284. ACM, 2014.

[CHS+16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *arXiv e-print*, arXiv:1602.02830, Feb 2016.

[CLL+14] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Teman. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622. IEEE Computer Society, 2014.

[Cou16] Matthieu Courbariaux. BinaryNet. https://github.com/MatthieuCourbariaux/BinaryNet/, 2016.

[CPS06] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[CSE16] Vincent Camus, Jeremy Schlachter, and Christian Enz. A low-power carry cut-back approximate adder with fixed-point implementation and floating-point precision. In *Proceedings of the 53rd Annual Design Automation Conference*, page 127. ACM, 2016.

[CSML15] Zhiyong Cheng, Daniel Soudry, Zexi Mao, and Zhenzhong Lan. Training binary multilayer neural networks for image classification using expectation backpropagation. *arXiv preprint arXiv:1503.03562*, 2015.

[CWB+11] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[DCBV09] Teófilo Emídio De Campos, Bodla Rakesh Babu, and Manik Varma. Character recognition in natural images. *VISAPP (2)*, 7, 2009.

[DDP11] Florent De Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 4:18–27, 2011.

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[DLC+15] Zidong Du, Avinash Lingamneni, Yunji Chen, Krishna V Palem, Olivier Temam, and Chengyong Wu. Leveraging the error resilience of neural networks for designing highly energy efficient accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1223–1235, 2015.

[DQX+17] Jifeng Dai, Haozhi Qi, Yuwen Xiong, Yi Li, Guodong Zhang, Han Hu, and Yichen Wei. Deformable convolutional networks. In *ICCV*, 2017.

[DSH13] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8609–8613. IEEE, 2013.

[DT05] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *CVPR*, pages 886–893, 2005.

[DWSP09] Piotr Dollár, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection: A benchmark. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 304–311. IEEE, 2009.

[ESKD+03] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO-36.*, 2003.

[FHCW16] Gan Feng, Zuyi Hu, Song Chen, and Feng Wu. Energy-efficient and high-throughput fpga-based accelerator for convolutional neural networks. In *ICSICT*, pages 624–626, 2016.

[GWFM+13] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, and Yoshua Bengio. Maxout Networks. *Int'l Conf. on Machine Learning (ICML)*, pages 1319–1327, Feb 2013.

[GWL+18] Lei Gong, Chao Wang, Xi Li, Huaping Chen, and Xuehai Zhou. Maloc: A fully pipelined fpga accelerator for convolutional neural networks with all layers mapped on chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2601–2612, 2018.

[Gyb89] G Gybenko. Approximation by superposition of sigmoidal functions. *Mathematics of Control, Signals and Systems 2, no. 4*, pages 303–314, 1989.

[GYC16] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *NIPS*, 2016.

[HAM07] S Himavathi, D Anitha, and A Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, 2007.

[HCS+16a] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized Neural Networks. *Advances in Neural Information Processing Systems (NIPS)*, 2016.

[HCS+16b] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115, 2016.

[HDY+12] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[HHG+13a] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. *Conference on Information and Knowledge Management (CIKM)*, pages 2333–2338, 2013.

[HHG+13b] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 2333–2338. ACM, 2013.

[HKPH91] John Hertz, Anders Krogh, Richard G Palmer, and Heinz Horner. Introduction to the theory of neural computation. *Physics Today*, 44:70, 1991.

[HLM+16] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. IEEE, 2016.

[HLvdMW17] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[HMD15] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2015.

[Hor14] Mark Horowitz. 1.1 computing's energy problem (and what we can do about it). In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 10–14. IEEE, 2014.

[HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[HPTD15] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143, 2015.

[HS14] Kyuyeon Hwang and Wonyong Sung. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE, 2014.

[HSW93] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *Neural Networks, 1993., IEEE International Conference on*, pages 293–299. IEEE, 1993.

[HZRS15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[IHM+16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[int16] SAE international. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. *SAE International,(J3016)*, 2016.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *Int'l Conf. on Machine Learning (ICML)*, pages 448–456, 2015.

[JCC+17] Xun Jiao, Vincent Camus, Mattia Cacciotti, Yu Jiang, Christian Enz, and Rajesh K Gupta. Combining structural and timing errors in overclocked inexact speculative adders. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 482–487. IEEE, 2017.

[JJRG16] Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh K Gupta. Wild: A workload-based learning model to predict dynamic delay of functional units. In *Computer Design (ICCD), 2016 IEEE 34th International Conference on*, pages 185–192. IEEE, 2016.

[JJRG17] Xun Jiao, Yu Jiang, Abbas Rahimi, and Rajesh K Gupta. Slot: A supervised learning model to predict dynamic timing errors of functional units. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1183–1188. IEEE, 2017.

[JK17] Yunho Jeon and Junmo Kim. Active convolution: Learning the shape of convolution for image classification. In *CVPR*, 2017.

[JLM10] Vidit Jain and Erik Learned-Miller. Fddb: A benchmark for face detection in unconstrained settings. Technical Report UM-CS-2010-009, University of Massachusetts, Amherst, 2010.

[JRN+15] Xun Jiao, Abbas Rahimi, Balakrishnan Narayanaswamy, Hamed Fatemi, Jose Pineda de Gyvez, and Rajesh K. Gupta. Supervised learning based model for predicting variability-induced timing errors. In *Proc. of NEWCAS*. IEEE, 2015.

[JVZ14] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up Convolutional Neural Networks with Low Rank Expansions. *arXiv e-print*, arXiv:1405.3866, May 2014.

[JXBS17] Felix Juefei-Xu, Vishnu Naresh Boddeti, and Marios Savvides. Local binary convolutional neural networks. In *CVPR*, 2017.

[JZL+14] Yu Jiang, Hehua Zhang, Han Liu, William Hung, Xiaoyu Song, Ming Gu, and Jiaguang Sun. System reliability calculation based on the run-time analysis of ladder program. *IEEE Transactions on Industrial Electronics*, 2014.

[JZS+13] Yu Jiang, Hehua Zhang, Xiaoyu Song, Xun Jiao, William NN Hung, Ming Gu, and Jiaguang Sun. Bayesian-network-based reliability analysis of plc systems. *IEEE transactions on industrial electronics*, 2013.

[KB14] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv e-print*, arXiv:1412.6980, Dec 2014.

[KGV17] Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 kb ram for the internet of things. In *Int'l Conf. on Machine Learning (ICML)*, pages 1935–1944, 2017.

[KS16] Minje Kim and Paris Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems (NIPS)*, pages 1097–1105, 2012.

[LBD+89a] Yann LeCun, Bernhard Boser, John S. Denker, Donnie Henderson, Richard E. Howard, Wayne Hubbard, and Lawrence D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1.4:541–551, 1989.

[LBD+89b] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[LCB98] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[LCMB15] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*, 2015.

[LCY14] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *Int'l Conf. on Learning Representations (ICLR)*, 2014.

[LDS+89] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPS*, 1989.

[LDS90] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605, 1990.

[LK81] Bruce D Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. *Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, 1981.

[Low04] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[LWF+15] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *CVPR*, 2015.

[LXZ+17] Jeng-Hau Lin, Tianwei Xing, Ritchie Zhao, Mani Srivastava, Zhiru Zhang, Zhuowen Tu, and Rajesh Gupta. Binarized convolutional neural networks with separable filters for efficient hardware acceleration. *Computer Vision and Pattern Recognition Workshop (CVPRW)*, 2017.

[LYGT18] Jeng-Hau Lin, Yunnan Yang, Rajesh Gupta, and Zhuowen Tu. Local binary pattern networks. *arXiv preprint arXiv:1803.07125*, 2018.

[ME12] Volodymyr Mnih and Geoffrey E. Learning to label aerial images from noisy data. *Int'l Conf. on Machine Learning (ICML)*, pages 567–574, 2012.

[MH12] Volodymyr Mnih and Geoffrey E Hinton. Learning to label aerial images from noisy data. In *Proceedings of the 29th International conference on machine learning (ICML-12)*, pages 567–574, 2012.

[MSS+16] Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, and Kaushik Roy. Design of power-efficient approximate multipliers for approximate artificial neural networks. In *2016 International Conference On Computer Aided Design (ICCAD)(prijato)*, page 7, 2016.

[Nom16] Taiga Nomi. tiny-cnn. https://github.com/nyanp/tiny-cnn, 2016.

[Nvi11] CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

[OPH96] Timo Ojala, Matti Pietikäinen, and David Harwood. A comparative study of texture measures with classification based on featured distributions. *Pattern recognition*, 29(1):51–59, 1996.

[PL00] Thodore Papadopoulo and Manolis IA Lourakis. Estimating the jacobian of the singular value decomposition: Theory and applications. *European Conference on Computer Vision (ECCV)*, pages 554–570, 2000.

[QWY+16] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 26–35. ACM, Feb 2016.

[RGA+17] Brandon Reagen, Udit Gupta, Robert Adolf, Michael M Mitzenmacher, Alexander M Rush, Gu-Yeon Wei, and David Brooks. Weightless: Lossy weight encoding for deep neural network compression. *arXiv preprint arXiv:1711.04686*, 2017.

[RHW85a] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[RHW85b] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. *CALIFORNIA UNIV SAN DIEGO LA JOLLA INST FOR COGNITIVE SCIENCE*, 1985.

[RORF16a] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[RORF16b] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *European Conference on Computer Vision (ECCV)*, Oct 2016. arXiv:1603.05279.

[RSLF13] Roberto Rigamonti, Amos Sironi, Vincent Lepetit, and Pascal Fua. Learning Separable Filters. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2754–2761, 2013.

[SCD+16] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-Optimal OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, pages 16–25, Feb 2016.

[SCL12] Pierre Sermanet, Soumith Chintala, and Yann LeCun. Convolutional neural networks applied to house numbers digit classification. In *ICPR*, pages 3288–3291. IEEE, 2012.

[SDF+11] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general

low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[SG64] Abraham Savitzky and Marcel JE Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, pages 1627–1639, Jul 1964.

[SHM14a] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: Parameter-free training of multilayer neural networks with continuous or discrete weights. In *Advances in Neural Information Processing Systems*, pages 963–971, 2014.

[SHM14b] Daniel Soudry, Itay Hubara, and Ron Meir. Expectation backpropagation: parameter-free training of multilayer neural networks with continuous or discrete weights. *Advances in Neural Information Processing Systems (NIPS)*, pages 963–971, 2014.

[SIV16] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv e-print*, arXiv:1602.07261, Feb 2016.

[SLF14a] Amos Sironi, Vincent Lepetit, and Pascal Fua. Multiscale centerline detection by learning a scale-space distance transform. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2697–2704, 2014.

[SLF14b] Amos Sironi, Vincent Lepetit, and Pascal Fua. Multiscale centerline detection by learning a scale-space distance transform. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2697–2704, 2014.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[SPM+16] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[SSP03] Patrice Y Simard, David Steinkraus, and John C Platt. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR*, volume 3, pages 958–962, 2003.

[SVI+15] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *arXiv e-print*, arXiv:1512.00567, Dec 2015.

[SWLS+15] Christian Szegedy, Yangqing Jia Wei Liu, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[SZ14] Karen Simonyan and Anderw Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-print*, arXiv:1409.1556, Sep 2014.

[SZ15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[The16] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-print*, arXiv:1605.02688, May 2016.

[Tie13] T Tieleman. affnist. *URL https://www. cs. toronto. edu/˜ tijmen/affNIST/, Dataset URL https://www. cs. toronto. edu/˜ tijmen/affNIST/.[Accessed on: 2018-05-08]*, 2013.

[UFG+17] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017.

[VB16] S. I. Venieris and C. S. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *FCCM*, 2016.

[WDF+17] Ying Wang, Jiachao Deng, Yuntan Fang, Huawei Li, and Xiaowei Li. Resilience-aware frequency tuning for neural-network-based approximate computing chips. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2017.

[WHY09] Xiaoyu Wang, Tony X Han, and Shuicheng Yan. An hog-lbp human detector with partial occlusion handling. In *CVPR*, 2009.

[WWCN12] Tao Wang, David J Wu, Adam Coates, and Andrew Y Ng. End-to-end text recognition with convolutional neural networks. In *Pattern Recognition (ICPR), 2012 21st International Conference on*, pages 3304–3308. IEEE, 2012.

[XGD+17] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *CVPR*. IEEE, 2017.

[YBL14] Cong Yao, Xiang Bai, and Wenyu Liu. A unified framework for multioriented text detection and recognition. *IEEE Transactions on Image Processing*, 23(11):4737–4749, 2014.

[YBSL14] Cong Yao, Xiang Bai, Baoguang Shi, and Wenyu Liu. Strokelets: A learned multi-scale representation for scene text recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4042–4049, 2014.

[YJZ+06] Hongmei Yan, Yingtao Jiang, Jun Zheng, Chenglin Peng, and Qinghui Li. A multilayer perceptron-based medical decision support system for heart disease diagnosis. *Expert Systems with Applications*, 30(2):272–281, 2006.

[YLW+16] Yongtao Yu, Jonathan Li, Chenglu Wen, Haiyan Guan, Huan Luo, and Cheng Wang. Bag-of-visual-phrases and hierarchical deep models for traffic sign detection and recognition in mobile laser scanning data. *ISPRS journal of photogrammetry and remote sensing*, 113:106–123, 2016.

[YWZL13] Fei Yin, Qiu-Feng Wang, Xu-Yao Zhang, and Cheng-Lin Liu. Icdar 2013 chinese handwriting recognition competition. In *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on*, pages 1464–1470. IEEE, 2013.

[ZG17] Jeff Jun Zhang and Siddharth Garg. Bandits: Dynamic timing speculation using multi-armed bandit based optimization. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 922–925. IEEE, 2017.

[ZHMD17] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. *ICLR*, 2017.

[ZSZ+17] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24. ACM, 2017.

[ZWT+15] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. Approxann: an approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 701–706, 2015.