저작자표시-비영리-변경금지 2.0 대한민국

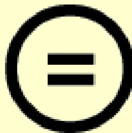
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Ph.D. Dissertation

Hardware Friendly Neural Network Architecture and Accelerator Design for Efficient Inference

효율적인 추론을 위한 하드웨어 친화적 신경망 구조 및 가속기 설계

August 2020

Department of Electrical and Computer Engineering
College of Engineering
Seoul National University

Joonsang Yu

Hardware Friendly Neural Network Architecture and Accelerator Design for Efficient Inference

효율적인 추론을 위한 하드웨어 친화적 신경망 구조 및 가속기 설계

지도교수 이 혁 재

이 논문을 공학박사 학위논문으로 제출함

2020년 7월

서울대학교 대학원

전기 정보 공학부

유 준 상

유준상의 공학박사 학위 논문을 인준함

2020년 7월

위 원 장:	김태환	(인)
부위원장:	이혁재	(인)
위 원:	유승주	(인)
위 원:	류수정	(인)
위 원:	이진호	(인)

Abstract

Joonsang Yu
Dept. of Electrical and Computer Engineering
The Graduate School
Seoul National University

Deep learning is the most promising machine learning algorithm, and it is already used in real life. Actually, the latest smartphone use a neural network for better photograph and voice recognition. However, as the performance of the neural network improved, the hardware cost dramatically increases. Until the past few years, many researches focus on only a single side such as hardware or software, so its actual cost is hardly improved. Therefore, hardware and software co-optimization is needed to achieve further improvement. For this reason, this dissertation proposes the efficient inference system considering the hardware accelerator to the network architecture design.

The first part of the dissertation is a deep neural network accelerator with stochastic computing. The main goal is the efficient stochastic computing hardware design for a convolutional neural network. It includes stochastic ReLU and optimized max function, which are key components in the convolutional neural network. To avoid the range limitation problem of stochastic numbers and increase the signal-to-noise ratio, we perform weight normalization and upscaling. In addition, to reduce the overhead of binary-to-stochastic conversion, we propose a scheme for sharing stochastic number generators among the neurons in the convolutional neural network.

The second part of the dissertation is a neural architecture transformation. The network recasting is proposed, and it enables the network architecture transformation. The primary goal of this method is to accelerate the inference process through the

transformation, but there can be many other practical applications. The method is based on block-wise recasting; it recasts each source block in a pre-trained teacher network to a target block in a student network. For the recasting, a target block is trained such that its output activation approximates that of the source block. Such a block-by-block recasting in a sequential manner transforms the network architecture while preserving accuracy. This method can be used to transform an arbitrary teacher network type to an arbitrary student network type. It can even generate a mixed-architecture network that consists of two or more types of block. The network recasting can generate a network with fewer parameters and/or activations, which reduce the inference time significantly. Naturally, it can be used for network compression by recasting a trained network into a smaller network of the same type.

The third part of the dissertation is a fine-grained neural architecture search. InheritedNAS is the fine-grained architecture search method, and it uses the coarse-grained architecture that is found from the cell-based architecture search. Basically, fine-grained architecture has a very large search space, so it is hard to find directly. A stage independent search is proposed, and this method divides the entire network to several stages and trains each stage independently. To break the dependency between each stage, a two-point matching distillation method is also proposed. And then, operation pruning is applied to remove the unimportant operation. The block-wise pruning method is used to remove the operations rather than the node-wise pruning. In addition, a hardware-aware latency penalty is proposed, and it covers not only FLOPs but also memory access.

keywords: Stochastic Computing, Deep Neural Network Accelerator,
Network Compression, Network Transformation, Neural Architecture Search
student number: 2015-20950

Contents

Abstract **i**

Contents **iii**

List of Figures **vi**

List of Tables **xi**

1 Introduction **1**

1.1 DNN Accelerator with Stochastic Computing 2

1.2 Neural Architecture Transformation 4

1.3 Fine-Grained Neural Architecture Search 6

2 Background **8**

2.1 Stochastic Computing 8

2.2 Neural Network 10

2.2.1 Network Compression 10

2.2.2 Neural Network Accelerator 13

2.3 Knowledge Distillation 17

2.4 Neural Architecture Search 19

3 DNN Accelerator with Stochastic Computing **23**

3.1 Motivation 23

3.1.1 Multiplication Error on Stochastic Computing 23
3.1.2 DNN with Stochastic Computing 24
3.2 Unipolar SC Hardware for CNN 25
3.2.1 Overall Hardware Design 25
3.2.2 Stochastic ReLU function 27
3.2.3 Stochastic Max function 30
3.2.4 Efficient Average Function 36
3.3 Weight Modulation for SC Hardware 38
3.3.1 Weight Normalization for SC 38
3.3.2 Weight Upscaling for Output Layer 43
3.4 Early Decision Termination 44
3.5 Stochastic Number Generator Sharing 49
3.6 Experiments 53
3.6.1 Accuracy of CNN using Unipolar SC 53
3.6.2 Synthesis Result 57
3.7 Summary 58

4 Neural Architecture Transformation 59

4.1 Motivation 59
4.2 Network Recasting 61
4.2.1 Recasting from DenseNet to ResNet and ConvNet 63
4.2.2 Recasting from ResNet to ConvNet 63
4.2.3 Compression 63
4.2.4 Block Training 65
4.2.5 Sequential Recasting and Fine-tuning 67
4.3 Experiments 69
4.3.1 Visualization of Filter Reduction 70
4.3.2 CIFAR 71
4.3.3 ILSVRC2012 73

4.4 Summary 76

5 Fine-Grained Neural Architecture Search 77

5.1 Motivation 77

5.1.1 Search Space Reduction Versus Diversity 77

5.1.2 Hardware-Aware Optimization 78

5.2 InheritedNAS 79

5.2.1 Stage Independent Search 79

5.2.2 Operation Pruning 82

5.2.3 Entire Search Procedure 83

5.3 Hardware-aware Penalty Design 85

5.4 Experiments 87

5.4.1 Fine-Grained Architecture Search 88

5.4.2 Penalty Analysis 90

5.5 Summary 92

6 Conclusion 93

Abstract (In Korean) 113

List of Figures

2.1 Stochastic computing multipliers. (a) Stochastic multiplication in unipolar encoding with range [0, 1]. (b) Stochastic multiplication in bipolar encoding with range $[-1, 1]$. 9

2.2 Simplified knowledge distillation method. 17

2.3 Example of gradient-based neural architecture search and continuous relaxation. 21

3.1 Mean absolute errors for multiplications of two 10-bit streams in stochastic computing. (a) Unipolar encoding (AND gate error). (b) Bipolar encoding (XNOR gate error). 24

3.2 Convolutional neural network based on stochastic computing. (a) A simplified convolutional neural network consisting of convolutional, pooling, and fully connected layers. (b) Structure of a stochastic computing neuron, which can be used for both convolutional and fully-connected layers. (c) Max pooling hardware structure. 26

3.3 Simplified integrated-and-fire (IF) neuron in spiking neural networks. 28

3.4 Stochastic ReLU function based on finite state machine. (a) State diagram of SReLU. (b) Results of SReLU and saturated ReLU for $3,000$ random inputs.(c) Mean error of SReLU to approximate saturated ReLU. 29

3.5 Stochastic max function. (a) The conventional stochastic max function. (b) Our proposed stochastic max function (SMax). 32

3.6 Results of the stochastic max functions for $1,000$ random bitstream pairs. (a) The conventional stochastic max function. (b) Our proposed stochastic max function (SMax). 33
3.7 Error of stochastic max functions. (a) The previous work. (b) Our proposed max function. 34
3.8 Mean absolute errors of 2-input max operation and 2×2 max pooling hardware. X-axis means the length of input/output bitstreams. 35
3.9 The scaled adder hardware. 36
3.10 2×2 stochastic average function. (a) The conventional stochastic average pooling function (scaled adder). (b) Our proposed stochastic average function (SAvg). 37
3.11 Mean absolute errors of 2-input average operation and 2×2 average pooling hardware. X-axis means the length of input/output bitstreams. 39
3.12 Distribution of activations (output values) for each layer. The number of activations is normalized to the number of neurons in each layer, and zero activations are not counted. (a) Comparison of distributions for each layer when max normalization is applied. (b) Comparison of different normalization schemes. 42
3.13 Distribution of top-2 differences for misclassified images. 44
3.14 Early decision termination steps. Ground truth is 2 in both cases. . . . 45
3.15 Simplified sharing of a random number generator among the different stochastic number generators. 49
3.16 Stochastic number generator sharing method for the convolutional layer. 51
3.17 Stochastic number generator sharing method for the fully-connected layer. 51

3.18 Test error for CIFAR-10 dataset. The test error of floating-point is 16.14%. In SC, test error is minimized when both 99.55% normalization and weight upscaling are used, and its test error is 16.43%. In addition, test error becomes 17.08% when RNG sharing is applied to fully-connected layer. 54
3.19 Early decision termination result on CIFAR-10 dataset. More than half of the test data can be classified with lower than 2^9-bit precision, but 20% of the data still require 2^{10}-bit precision. 55
3.20 Early decision termination result on MNIST dataset. Almost test data can be correctly classified with 2^6-bit precision. 55
3.21 Synthesis result of conventional bipolar neuron and proposed unipolar neuron. (a) Area comparison of bipolar and unipolar neuron. LFSR is used to calculate area in all neuron. (b) Energy consumption of each neuron. All circuit execute 2^{10}(=1024)-bits. 57

4.1 ResNet and DenseNet Top-1 validation errors for different numbers of multiplications (*left*) and inference times (*right*). To measure the inference time, single NVIDIA Titan X (Pascal) is used and batch size is set to 16. DenseNet has much fewer multiplications than ResNet, but its inference time is much longer. 60
4.2 Basic concept of the network recasting. The target block of the student network is trained by mimicking the source block of the teacher network. 61
4.3 Block recasting of a dense block into a basic block (Case 1) and a convolution block (Case 2). The basic block has shorter inference time than the dense block because it has much smaller activation load. The convolution block is even faster than the basic block, but its capacity is much smaller and so it can cause accuracy loss. 62

4.4 Block recasting of a residual block—basic block (Case 1) and bottleneck (Case 2)— into a convolution block. The recasting of the basic block keeps the same number of input and output channels. However, since the bottleneck block uses a smaller number of channels for the feature extraction, we recast it into a convolution block that has the same number of input and output channels as the original 3×3 convolution. 62
4.5 Examples of the VGG-16 and WRN-28-10 compression. Both example shows recasting of the first layer in each network. 64
4.6 Dimension mismatch and proposed block training method. The dimension mismatch happens when the source block is recast into a smaller target block. The next block is used to match the dimension of output activation. After rebuilding the next block, both blocks are trained by minimizing $\mathcal{L}_{mse}(W_T, W_S)$. 66
4.7 Example of sequential recasting for ResNet-50. All blocks are recast in this example. In each step, the target block and the next block (shaded blocks) are initialized randomly and trained by minimizing $\mathcal{L}_{mse}(W_T, W_S)$. 68
4.8 Example of the mixed-architecture network. It has both residual and dense block. 68
4.9 Visualization of filters in the first layer of AlexNet (*left*) and a student network (*right*). Redundant filters are removed after network recasting. 70

5.1 The overall process of InheritedNAS. First, coarse-grain architecture is searched. After then the fine-grain architecture is searched with pre-trained coarse-grain architecture. To reduce the search space, we divide and train the network using the knowledge distillation, and the teacher network gives layer-wise/stage-wise hints to the student network for the fine-grain architecture search. Each super cell has its own architecture parameters, so each block has intrinsic architectures after search. 80
5.2 The two-point matching distillation. This method can break the dependency from s-th to (s+1)-th stage, so each stage can be trained independently. 81
5.3 Forward propagation for the connectivity parameters θ. The probability of connection is calculated with the sigmoid function, and it works as the scaling factor of each operation. 83
5.4 Comparison of searched architecture. (*left*) The normal cell of DARTS. (*right*) The first block of our searched network (OS1). 89

List of Tables

3.1 Accuracy and gate count comparison for max function 35
3.2 Accuracy and gate count comparison for average pooling 39
3.3 Average signal-to-noise ratio for different normalizations 42
3.4 Comparison with previous works in terms of configuration and test error 56

4.1 Candidates for the network recasting 66
4.2 Error rates (%) of architecture transform results on CIFAR datasets (B/M: billion/million) 72
4.3 Error rates (%) of compression results on CIFAR datasets (B/M: billion/million) 73
4.4 Error rate (%) of network recasting results on ILSVRC2012 (B/M: billion/million, I/B: image/batch) 74
4.5 Comparison of error rate (%) with previous works on ILSVRC2012 (B/M: billion/million) 75

5.1 Experimental results of InheritedNAS 89
5.2 Architecture search results through the hardware penalties 90
5.3 Latency on the CPU and GPU 91

Chapter 1

Introduction

Nowadays, we live in the ear of big data, and almost global companies use them for technological advancements and people's convenience. Deep learning (DL) is considered the most popular and promising machine learning (ML) algorithm, and it is also the most famous big data application. In recent image classification challenges, a convolutional neural network (CNN), which is a kind of deep neural network (DNN) architecture, is widely used and achieves the highest classification accuracy. Several CNN architectures have been introduced to achieve even higher accuracy [1, 2, 3, 4, 5, 6], and the networks become deeper and deeper to take the exponential advantage of depth [7]. To train a deep network, He et al. [4] proposed the residual network (ResNet), which consists of the summation of identity mapping and output of convolutional layers. It helps to propagate gradients from the top layer to bottom layer, so it can alleviate the vanishing gradient problem. In addition, ResNet shows top-5 accuracy (probability of having the right answer in the top-5 predictions) higher than 95%, and top-1 accuracy higher than 80% [4], exceeding the capability of a human.

To achieve state-of-the-art accuracy, many network architectures and training methods are proposed, but it causes the inefficiency on inference. Deeper network architectures help to achieve higher accuracy, but those require a huge amount of computation. A DNN consists of an enormous number of multiply-and-accumulate (MAC) oper-

ations for matrix-vector multiplication, which is very inefficient on the conventional hardware. In addition, several previous works show that DNN has tremendous redundancy, and it can be removed easily [8]. So, it is important to achieve the speed up and energy consumption by removing the redundancy. Moreover, hardware aware network architecture design is required for further optimization.

1.1 DNN Accelerator with Stochastic Computing

To reduce the computational cost while maintaining a reasonable level of accuracy, various kinds of methods have been introduced. For example, the fixed-point hardware is widely used [9] because it is much cheaper than floating-point hardware. Note, however, that fixed-point number representation has a much narrower dynamic range than the floating-point counterpart, and thus the hardware should be designed carefully to avoid overflow or underflow. Alternatively, analog computing can be adopted to perform the MAC operations at a cost lower than that of conventional digital hardware [10]. A memristor crossbar is an example of analog matrix-vector multiplications, which has been shown to have a huge improvement in terms of performance and energy consumption over digital implementations.

Approximate computing is another way of reducing computational cost by sacrificing accuracy. The concept fits well with DNNs, since sacrificing a certain level of accuracy for the internal computations of a DNN does not necessarily degrade its prediction quality. Stochastic computing (SC) can be considered as approximate computing; it has several advantages over conventional fixed-point computation and analog computation. First, compared with fixed-point, SC multiplier has a smaller hardware footprint, lower power consumption, and lower latency. Thanks to those characteristics, more neurons can be integrated into the same area compared to the conventional fixed-point hardware. Secondly, SC requires a smaller number of conversions than analog. DNNs using analog MAC hardware typically require digital-to-analog con-

version and analog-to-digital conversion much more pervasively, i.e., before and after every MAC layer. The conversion overhead of SC, on the other hand, can be confined to primary inputs (binary-to-stochastic), weight parameters (binary-to-stochastic), and primary outputs (stochastic-to-binary) [11]. In addition, SC allows dynamic change of precision without any hardware modification [12]. This characteristic of SC can be exploited to considerably reduce the latency of SC DNN at no increase in hardware cost [11].

There have been several attempts to design efficient SC hardware for DNNs, targeting fully-connected networks [11] as well as CNN [13, 14, 15, 16, 17]. However, the previous SC DNN designs have two main problems. First, they do not scale well in terms of recognition accuracy beyond the MNIST dataset. The MNIST consists of simple high-contrast gray-scale handwritten digit images. Actually, each pixel has an 8-bit value, but 80.9% of the pixels have value 0, and 7.4% of the pixels have value 250 or larger. On the other hand, general object recognition datasets such as CIFAR-10 and ImageNet have three channels with many mid-range pixels making it really difficult to classify them, which is why there is no reported result yet on those datasets in the previous SC DNN papers. Second, they incur large overhead due to the conversion of fixed-point data and weights into stochastic bitstreams, which significantly reduces energy- and area-efficiency of SC DNNs.

In the first part of the dissertation (Section 3), we address the three central problems of SC DNNs by proposing the following set of novel techniques. First, we propose to use unipolar encoding for SC DNN designs, which can help reduce random errors of SC and make SC DNN more accurate. We also propose a set of unipolar SC-based hardware modules, such as SReLU and Smax, which are SC versions of ReLU and max, respectively. Second, we propose data-driven weight normalization and weight upscaling tailored for SC DNNs. Thirdly, to minimize the conversion overhead associated with SC, we propose a novel SNG (stochastic number generator) sharing scheme.

Through our experiments, we show that our SC DNN achieves significantly im-

proved recognition accuracy and efficiency compared with the state-of-the-art result. In terms of recognition accuracy, our optimized version achieves 16.43% test error on CIFAR-10, which is very close to the floating-point test error. In terms of efficiency, we show that, with the proposed SNG sharing scheme, a fully-connected neuron and a convolutional neuron need only 47.5% and 16.2% area, respectively, compared with the conventional SC neuron without sharing. In addition, our experiments show that the energy efficiency of the fully-connected and convolutional neurons can be made $5.3\times$ and $9.2\times$ higher, respectively, by using the proposed sharing scheme.

1.2 Neural Architecture Transformation

As mentioned before, deeper network architectures help to achieve higher accuracy, but those have a huge amount of parameters and computation redundancies. In conventional neural network training, L1 or L2 (weight decay) regularization is used to improve the generalization performance. Both methods decrease the weight values, so weights become close to zero. For this reason, most weight values are located in the near-zero area, and those hardly affect the final prediction. Actually, many filters cannot extract the meaningful features [18], so those can be removed. To obtain an efficient network, the network compression method is introduced by removing the redundancies of the trained network. The weight and filter pruning methods are introduced to remove redundant filter [8, 19], and they remove the weight or filter whose absolute value is smaller than the given threshold value. The pruning can reduce the network size effectively, so other pruning approaches are proposed for further reduction.

On the other hand, there are architectural approaches that are designing the computation efficient network. Szegedy et al. [2] propose the inception module, and it supports several filter size of convolution. 3×3 and 5×5 require many multiplications, so they introduce the 1×1 convolution to reduce the number of multiplications. 1×1 convolution is used before the main convolution, and it reduces the number of

activation channels. By using 1×1 convolution, we can reduce not only the number of multiplications but also the number of weight parameters. For this reason, recent networks use 1×1 convolutions [4, 6].

However, in many cases, both pruning and 1×1 convolution cannot reduce the inference time effectively. First, the pruning method can reduce the model size effectively, but its actual speedup is much smaller than the compression ratio. Weight pruning method can remove unimportant parameters, but the filter becomes sparse matrix after pruning. The sparse matrix is hard to accelerate on conventional hardware such as CPU and GPU. Second, 1×1 convolution causes more memory access, so it increases the actual inference time. Memory access already occupies the most of inference time, but 1×1 convolution causes the growth of the memory access. Actually, even if the number of multiplication and parameters is small enough, the network can take a much longer time for the inference.

In the second part of the dissertation (Section 4), we focus on the inference time reduction rather than parameter and multiplication reduction. To reduce the inference time, we propose the *network recasting* method by transforming the network architecture for a smaller activation load. We transform the network architecture through the block-wise recasting of source blocks into target blocks. The recasting is done by training the target block to mimic the output activation of the source block, so the accuracy can be preserved after recasting. We can obtain a *mixed-architecture network* by recasting parts of the trained network. By the mixed-architecture network, we mean a network having multiple types of the block that can exploit the advantages of individual block types within a single network. In addition, we can use the network recasting method for network compression by recasting each block to a smaller one of the same type. We have achieved up to $3.2\times$ actual speedup with 0.22% top-5 accuracy loss on ILSVRC2012 dataset by the DenseNet-121 recasting.

1.3 Fine-Grained Neural Architecture Search

Recently, neural architecture search (NAS) is proposed to design network architecture automatically. NAS is one of the automated machine learning (AutoML) research, and it finds the neural network architecture with the neural network. Zope et al. [20] propose the first modern NAS algorithm, and it can find architecture using the deep reinforcement learning (RL). The overall process is similar to the design space exploration, but the deep RL can reduce the search space effectively. The evolutionary algorithm also can reduce the search space, so it is also combined with NAS algorithm [21]. However, those method requires a huge amount of time because every network architecture has to be trained during the search process. To reduce the search time, Liu et al. [22] propose the gradient-based NAS approach. Gradient-based NAS is finding the network architecture with conventional cross-entropy loss. This method trains network only one time, so it can reduce search time dramatically.

For the convergence, many NAS algorithms restrict the search space. Cell-based architecture search finds cell architecture and reuses them over the entire network. It has a much smaller search space compared with the whole architecture search, so it is much easier. However, a cell-based approach reduces the diversity of each cell, so it can reduce the optimization chance. According to the position in the network, each layer has a different characteristics. For example, the bottom layers have large input and output activation, and the top layers have large parameters. By adopting the proper structure, parameter, and memory overhead can be relaxed. ProxylessNAS [23] and FBNet [24] allow the layer-wise search space by using the simpler cell structure for the convergence.

In the last part of the dissertation (Section 5), we proposed *InheritedNAS*, the fine-grained architecture search method. The proposed method gives diversity for each block, so the network can achieve a higher accuracy or more efficient inference. To obtain the fine-grained architecture, we propose *the stage independent architecture search*, which can reduce the complexity of search space preserving the block diver-

sities. In addition, we also proposed *the operation pruning*, which is the operation removing method. This method estimates the importance of each operation and prunes the less important operations. The operation pruning optimizes each block and helps to achieve a more efficient network. On the other hand, we also proposed *the mixed penalty* that consists of FLOPs and memory access for hardware-aware architecture search. This penalty also keeps the block diversity and helps to find hardware friendly block designs.

Chapter 2

Background

2.1 Stochastic Computing

The stochastic computing (SC) is the digital hardware design scheme, and it is based on the probability theorem. SC uses *the bitstream* to express the number, and it is called *the stochastic number*. The bitstream consists of only 0 and 1, and the probability of emerging 1 means the actual value of a stochastic number. Only one bit (0 or 1) can be observed in every cycle, and the bits emerge during the several cycles. For example, nine 1s and one 0s are observed during ten cycles, the value of this stochastic number becomes $9/10$ in the binary system. If every bit is 1 or 0 in a bitstream, this value becomes 1 or 0 in the binary system. Therefore stochastic number has a range limitation form 0 to 1, and it is also the same as the probability. By generating the 1s with probability p, it means that stochastic number has p value in the binary system. Basically, an additional sign bit is required for the negative value because there is no negative probability. Therefore, the number of bit lines becomes twice due to the additional sign bit. However, the additional bit line is can be saved when 0 bit is regarded as -1. For the same example, the value becomes $(9 - 1)/10$ when there are nine 1s and one 0. In this case, the lower limit of the stochastic number becomes -1. Therefore, the value of stochastic number and its range limitation is changed according to

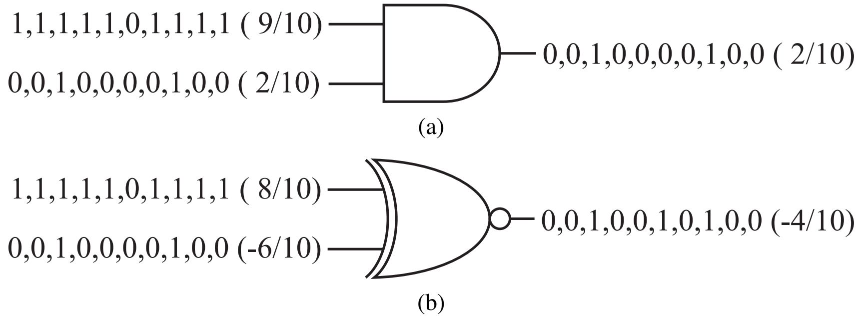


Figure 2.1: Stochastic computing multipliers. (a) Stochastic multiplication in unipolar encoding with range $[0, 1]$. (b) Stochastic multiplication in bipolar encoding with range $[-1, 1]$.

the meaning of 0 bit. The former case is called as *the unipolar encoding*, and the latter case is *the bipolar encoding*.

SC has a very small hardware footprint compared with the binary system. For example, multipliers can be implemented with only one gate in SC. Figure 2.1 shows the multiplier of SC for the unipolar and bipolar encoding. In unipolar encoding, single AND gate work as the multiplier. The value of the upper bitstream is $9/10$, so 1 is generated with a probability 0.9, and 1 is also generated with a probability 0.2 in the lower bitstream. So, the probability that both bitstreams have 1 bit is 0.9×0.2, and this is the same with AND gate operation. The XNOR gate is used for the multiplication in the bipolar encoding. Likewise, the scaled addition and subtraction are implemented with only one multiplexer, so those also have a very small hardware footprint. Thanks to this characteristic, SC is used in the signal processing hardware [25, 26, 27, 28, 29]. The edge detection logic can be implemented using only two XOR gates and one MUX [25], and gamma-correction logic also consists of one OR gate and one DFF [26].

Another advantage of SC is that the computation precision can be changed without any hardware modifications. The precision of the stochastic number and computation logics is directly related to bitstream length. So, the precision of the given number can be adjusted by changing the bitstream length. When the bitstream length is reduced to 2^{n-1}, its precision is also reduced to $n-1$-bit. In addition, the precision of arithmetic

logics depends on the precision of given input bitstreams, so the bitstream length has to be changed to adjust the precision of the entire hardware. Thanks to this characteristic, the concept of the progressive precision is introduced [25]. The quality (or prediction accuracy) of processing result is improved according to the computation cycles. The decision time is can be reduced when simple data (or operation) is given. For example, in the edge-detection, the sharp edges can be found easily, so the computation cycles of sharp edges can be reduced effectively.

2.2 Neural Network

The neural network is a kind of machine learning algorithm. The artificial neural network (ANN) mimics the behaviors of the neuron [30], and it was widely used for the regression problem because ANN can approximate any kind of function [31]. After introducing the backpropagation [32], the deep neural network (DNN) can be trained. As the depth of the neural network increases, the DNN achieves much higher prediction accuracy for the classification problem. These days, the DNN shows the state-of-the-art result for the many artificial intelligence areas compared with other machine learning algorithms. However, the computation cost and inference time also increase according to the network depth. To solve this problem, there is a lot of researches in both software and hardware aspects.

2.2.1 Network Compression

For the software approach, the network compression method is proposed to reduce the DNN model size and its inference time. The network depth and width dramatically increases to improve the prediction accuracy, but the actual improvement is much smaller than the network expansion. Therefore, many researchers have been focusing on the redundancy of the trained network. To reduce the network size, the weight pruning, and quantization methods are introduced [33, 34, 8]. Moreover, the matrix decompo-

sition method also used to reduce the parameter by using a low-rank approximation technique [35]. Nowadays, many researches focus on the network pruning and the quantization method and reduce the model size preserving the accuracy of the trained network.

Pruning

To reduce the search space or running time, the pruning method had been widely used in the various research areas. Han et al. [8] propose a weight pruning method, which removes useless weight connections. It is the first approach introducing the pruning method to neural network research. The basic concept of the first weight pruning is that the near-zero weight can be removed with any significant accuracy loss. After pruning, the removed weights are never used. To give the degree of freedom, the other works reuse the pruned weights. Guo et al. [36] introduce the pruning mask, which decides the pruning candidates. The mask is determined in every pruning epoch, and the pruned weight can be revived. Han et al. [37] also propose a re-initialization method, and it means that the pruning weights are revived with random initial values. This re-initialized weight can improve the prediction accuracy after the fine-tuning. Furthermore, the filter-wise pruning methods are also proposed [19, 38, 39, 40, 41]. In previous observation, several filters do not extract the meaningful features [18], and it can be removed with the group lasso regularization [42, 43]. Some previous approaches for the filter pruning are similar to the weight pruning method [19, 44]. The average percentage of zeros (APoZ) and the sum of absolute value are used to estimate the importance of each filer, and filters are pruned iteratively according to the estimated importance. Luo et al. [39] propose an activation-based pruning method, where pruning candidates are determined by the squared error of activations. They assume that small activation is not important for classification because it barely affects higher-level feature extraction. In recent work, Lin et al. [40] use reinforcement learning (RL) to prune filters according to the input data. This work introduces runtime

neural pruning that removes the filters using deep Q-network for each input data in runtime. Basically, these filter pruning methods achieve faster inference on a GPU, but still have limitations in compression; the methods for estimating the importance of filters should be designed manually with prior knowledge, and nonetheless, those methods cannot find redundant filters effectively due to a great deal of complexity.

Quantization

Another network compression approach is the quantization, and it can be used for the weight as well as activation value. The neural network does not require high computation precision, because it generates the decision probability [45]. The differences between the first and the second probable class are commonly large enough for the correct classification. In other words, those input data can be classified with lower precision when the computation error is smaller than the difference. By reducing the computation precision, the network size also can be reduced because it depends on the total memory of the weight values. Moreover, the computation cost also can be reduced according to the computation precision. For this reason, lots of the network quantization methods are proposed for efficient inference. The homogeneous quantization method is firstly proposed and it shows high compression results with reasonable accuracy loss [46]. The stochastic rounding method is proposed to use a much lower computation precision [47]. Han et al. [48] propose the weight sharing based quantization method, which can be used to achieve much higher compression ratio by using Huffman encoding. The layer-wise quantization methods are introduced for further improvement and those researches reveal that the input and output layer requires high precision but other layers can be quantized much lower precision [49, 50]. To reduce the model size extremely, the ternary weight and binary weight quantization methods are proposed [34, 51, 52]. Recently, many researches use the additional loss function to alleviate the accuracy loss in the quantization process. The explicit loss-error-aware Quantization (ELQ) is proposed to ternarize or binarize the weight value with much smaller

accuracy loss [53]. The ELQ considers both the weight approximation error and its impact on the cross-entropy loss, so it can help to preserve the prediction accuracy. The learnable quantization method is also introduced [54], and the new quantization parameters are included in the training process. Furthermore, the automated quantization method is also proposed [55]. By using the deep reinforcement learning algorithm (the actor-critic model proposed by [56]), the quantization levels are determined.

2.2.2 Neural Network Accelerator

DNN consists for a huge amount of multiplications and accumulations (MACs). For example, the number of multiplication is 4.09 billion in ResNet-50 model [4]. There is a tremendous amount of computation, so it spend a very long time in conventional computer architecture. However, the multiplication and accumulation emerge regularly, and its computation pattern is fixed when the network is chosen. For this reason, it can be accelerated easily by increasing the parallel computation, so many DNN accelerators are introduced. In addition, the network compression methods can help to achieve much faster inference, so the compression-aware accelerator design is very important.

Conventional Binary System

First of all, the DianNao [57] is designed to increases the utilization of the processing elements (PE) by using a tiling approach, and it achieves much higher speed-up compared with the conventional GPU architecture. And, the Eyeriss propose the spatial architectures that based on the CNN row stationary dataflow [58]. The systolic array is used to improve the efficiency of the convolution/fully-connected operation for the data center [59]. For further improvement, the network compression methods are considered to design the accelerator. The weight pruning can remove the connections of neurons, but it makes the sparse matrix so it is hard to be accelerated in conventional GPU architecture. However, it is easy to implement skipping operation for the zero

weights, so DNN accelerator can achieve further speed-up and energy-saving [60]. Most of the activation values also become zero due to the ReLU activation function that is widely used in modern neural network architecture. Another previous work propose both zero weight and activation skipping hardware, and they also solve the load imbalance problem caused by lots of zero values [61]. DNN accelerators also exploit quantized weight and activation values. Basically, a huge amount of area and energy consumption can be reduced by using the fixed-point rather than floating-point hardware [57]. The mixed-precision hardwares are proposed to use the advantages of the layer-wise quantization. The bit-serial hardware is proposed to support varying the bit precision [62]. The bit-flexible hardware also is proposed, and it used the bunch of 2-bit arithmetic operations to supporting the power-of-2 bitwidth operations [63]. By combining the quantization with sparsity, the DNN accelerator shows a much powerful result compared with conventional computer architecture [64]. In recent works, the prediction method is also used for further improvement. The SnaPEA architecture used the reordering and sampling method to predict the pre-activation values [65]. They calculate the partial sum of the convolution, and they skip the remaining operation when the temporal partial sum becomes smaller than zero. The ComPEND architecture also focuses on zero value prediction, but they change the two's complement operation [66]. The two's complement value consists of the large negative value and smaller positive values to express the negative. They invert this composition, and they calculate positive first, and then stop operation when the partial sum drops below zero. The convolution is used in the spatial domain, so it has local similarity. By using this characteristic, several works focus on the value prediction and its hardware for spatial domain [67].

Analog System

To implement a lower precision accelerator, an analog circuit can help to reduce area and energy consumption. The ISAAC proposes the memristor crossbar array archi-

tecture for the multiply-accumulate (MAC) operation [10]. For the MAC operation, digital-to-analog conversion and analog-to-digital conversion are required, but it has a huge amount of area and energy consumption overhead. They solve those problems by using the flipped form of the weight values, and it reduces ADC size effectively. The Prime architecture uses ReRAM crossbar array similar to ISAAC, but it chooses processing-in-memory (PIM) architecture for its implementation [68]. The PipeLayer proposes the highly parallel design according to the parallelism granularity and weight duplication by exploiting the inter-layer parallelism [69].

Stochastic Computing System

Approximate computing is another way of reducing computational cost by sacrificing accuracy. The concept fits well with DNN, since sacrificing a certain level of accuracy for the internal computations of a DNN does not necessarily degrade its prediction quality. SC can be considered as approximate computing; it has several advantages over conventional fixed-point computation and analog computation. First, compared with fixed-point, SC multiplier has a smaller hardware footprint, lower power consumption, and lower latency. Thanks to those characteristics, more neurons can be integrated into the same area compared to the conventional fixed-point hardware. Secondly, SC requires a smaller number of conversions than analog. DNN using analog MAC hardware typically requires digital-to-analog conversion and analog-to-digital conversion much more pervasively, i.e., before and after every MAC layer. In addition, SC allows dynamic change of precision without any hardware modification [12]. For those reasons, many previous works attempt to implement the SC DNN accelerator.

SC logics for the neural computation is proposed [70], and ANN is implemented with proposed logics [71]. However, only a single layer ANN model can be implemented, and SC cannot be used for the modern DNN model. SC has computation error and it is amplified pass through the many layers in DNN, so it cannot classify any given data. In addition, SC logics proposed in previous work [70] only support

only a single-input case. To solve those problems, the weight scaling method and new stochastic hyperbolic tangent are proposed [11], and it is the first full SC hardware for the modern DNN. In addition, they also use the progressive precision characteristic of SC and propose the early decision termination method to considerably reduce the latency of SC DNN at no increase in hardware cost. The DSCNN architecture supports SC-CNN, and they propose the improved version of single input stochastic hyperbolic tangent function [13]. After then, many CNN architectures with full SC logics are proposed. The SC-DCNN propose the optimized SC MAX pooling logic and reorganize the structure of convolutional neuron by switching the activation function and pooling operation [14]. To achieve a more efficient SC-DNN accelerator, li et al. [15] investigate the relationship and accuracy result for the MUX based inner product, parallel counter-based inner product, and order of activation function and pooling operation. The new version of SC tanh logic and SC ReLU function logics are proposed and those hardwares can improve the energy efficiency [16].

On the other hand, partial SC hardwares are also proposed. For the partial SC hardware, the binary-to-stochastic and stochastic-to-binary conversion is needed similar to the analog DNN accelerator. The SC-MAC hardware is proposed to solve conversion overhead, and it also improves the multiplication accuracy by reducing the effect of stochastic error [72]. They use the thermal coding for one stochastic number so it only requires down counter for the multiplication. And it shows similar accuracy with conventional SC multiplication because the randomness of other value is still guaranteed. The dynamic precision scaling technique is also proposed to change the precision in runtime [73]. The other work proposes the differential Multiply-and-Accumulate (DMAC) logics, and it helps to achieve the much higher speed-up compared with the previous SC-MAC hardware [74].

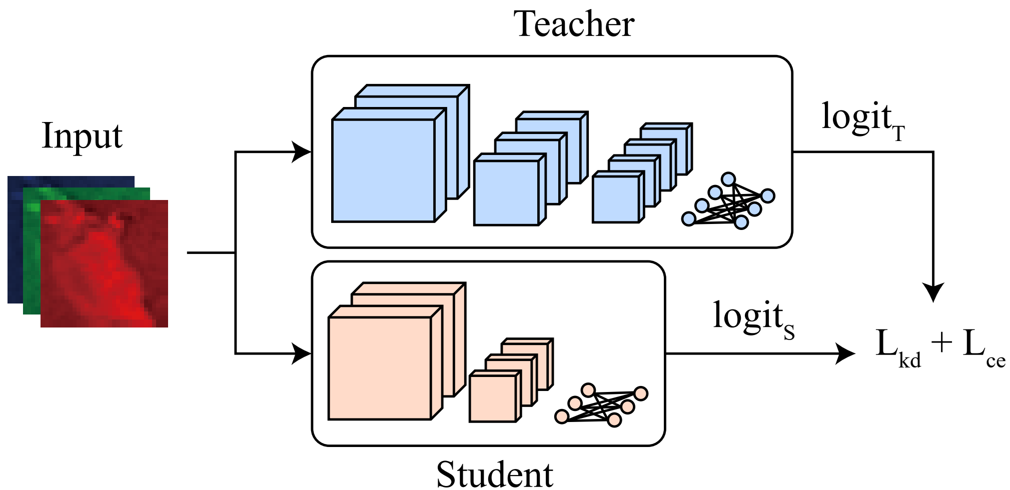


Figure 2.2: Simplified knowledge distillation method.

2.3 Knowledge Distillation

To train a smaller network with higher accuracy, mimic learning and knowledge distillation (KD) are introduced [75, 76]. These methods train a smaller network called *student network* using logits of a large network called *teacher network*. The teacher network was trained by minimizing cross-entropy loss that is widely used in conventional neural network training, and then the student network is trained by following the logits of the teacher network. The logits of the teacher network is considered as the soft target values, so it is much easier to follow the logit values rather than the original labels as shown in Figure 2.2. To follow the behavior of teacher network, Ba and Caruana [75] trains a student network by minimizing L2 loss between logits of student and teacher networks. Hinton at al. [76] uses logits of the teacher network to generate the soft target, and train the student network by minimizing cross-entropy loss with the soft target. Both works achieve a higher accuracy than conventional training method by using the proposed method, and Hinton at al. [76] also obtains the network that has the accuracy of the ensemble model. It is hard to train a deep student network due to the vanishing-gradient problem, so several KD methods have been proposed

to train a deep student network [77, 78]. Romero at al. [77] propose the hint training method that trains the student network with a convolutional regressor. They use a thinner and deeper student network, and train half of the student network by following the teacher network. Due to the dimension of the feature map is not matched, they use convolutional regressor and then train student network. After then, they also use the KD method for training of the entire network. Their method shows the higher accuracy compared with the original KD method, and they also show reasonable training result for the deeper network without any shortcut path. Luo et al. [78] also propose the method to train the deeper network. They also use a feature map of a hidden layer, but they use an additional path from the hidden layer to the output layer for gradient propagation. It also shows a better result compared with the conventional backpropagation method.

Recently, the KD method is widely used for the various areas; network compression, quantization, and network transformation. For the network compression, the activation matching method is widely used [79, 80]. The activation matching is a kind of hint training, but they never use the convolutional regressor. Zagoruyko et al. [79] introduce the attention transfer method to reduce the number of residual blocks while conserving the accuracy. The attention map is the channel-wise accumulation result in the feature map, and it shows the effective points in forward propagation. By following the attention map of the teacher network, the student network also can follow the behavior of the teacher network. They show the layer reduction of ResNet [4], and it shows a much better result compared with conventional training. Yim et al. [80] also propose the residual block reduction method using the relationship between input and output activations. Rather than the attention map, they propose the flow of the solution process (FSP), and this concept mimics the teaching method of a real-world teacher. In the real-world, the teacher does not give the solution, but they show the process to reach the solution. Similar to the real-world teacher, they give the relationship of input and output feature map to the student network. By mimicking the FSP, the stu-

dent network can be trained well. For the quantization, KD can also be applied easily [81, 82, 83]. Mishra et al. [81] use the low precision network as the student network, and it mimics the behavior of the teacher network similar to the original KD method. They show several kinds of quantization methods with KD, and the fine-tuning method achieves the best result. The fine-tuning method use a fully trained teacher and student network and then apply quantization. To restore the accuracy, the student network is fine-tuned by minimizing KD loss. Polino et al. [82] propose the differentiable quantization method, and it optimizes the location of quantization points. They define the differentiable quantization function, the quantization precision can be determined through the standard SGD. Their quantization process consists of the differentiable quantization and quantized distillation. After the quantization, they also use the KD method to recover the accuracy loss. For the network transformation, several previous works show the noticeable results [84, 85, 86]. Furlane et al. [84] firstly show the possibility of KD between different network architecture. They show the ResNet training result using logits of DenseNet, and it shows a better result compared with conventional backpropagation. Heo et al. [85] also show that KD can be applied between different network architecture. Li et al. [86] use the KD for the neural architecture search (NAS) application.

2.4 Neural Architecture Search

The neural architecture search (NAS) is the method to find better neural network architecture using the neural network itself. It is one of the automated machine learning research areas (AutoML), and Zoph and Le [20] propose the first NAS method. They propose the controller network using RNN, and it is trained to find network architecture. The controller generates the architecture hyperparameters such as filter size, stride, the number of output channels, the number of layers, etc. The network is built with the generated architecture hyperparameters, and it is trained to estimate valida-

tion accuracy. The validation accuracy is used to calculate the reward value to train the controller network, so it works as the environment in reinforcement learning. The controller is trained to increase the reward, and finally it can find the reasonable network architecture and shows good validation accuracy. To find better network architecture, the concept of a cell structure is proposed [87, 88]. The cell is the building block of network, and the network consists of several cells. This concept is already widely used for the network architecture design. For example, DenseNet [6] consists of the dense block and ResNet [4] also consists of residual or bottleneck block. Thanks to the concept of cell structure, the search space is dramatically reduced. They only find two kinds of cell; normal cell and reduction cell [87]. Training of the controller becomes much easier than previous work because the search space is much smaller.

The evolutionary algorithm is used to find neural network topology [89]. They firstly use the evolutionary algorithm for the network search, and it only covers the simple multi-layer perceptron. Liu et al. [90] propose an architecture search algorithm with the evolutionary algorithm for modern CNN architecture. They combine the evolutionary algorithm with the cell structure, and they only use a single cell as the network building block. To reduce the feature map size, they use the separable convolution with stride 2 that is proposed in [91]. Real et al. [21] combined evolutionary algorithm with the cell structure, and find normal and reduction cell. They find the network that has higher accuracy and lower computation cost compared with RL based architecture search. Elsken et al. [92] propose multi-objective architecture search for evolutionary architecture search, it covers accuracy, inference time, and the number of parameters.

One-shot architecture search is one of the neural architecture searches, which train all operations and select proper operation after training [93, 94, 95, 96, 22]. Basically, a one-shot search reuses the trained weight parameter for final prediction. The differentiable architecture search is one of the one-shot architecture searches, which is found network architecture by training itself. Liu et al. [22] propose the concept of gradient-

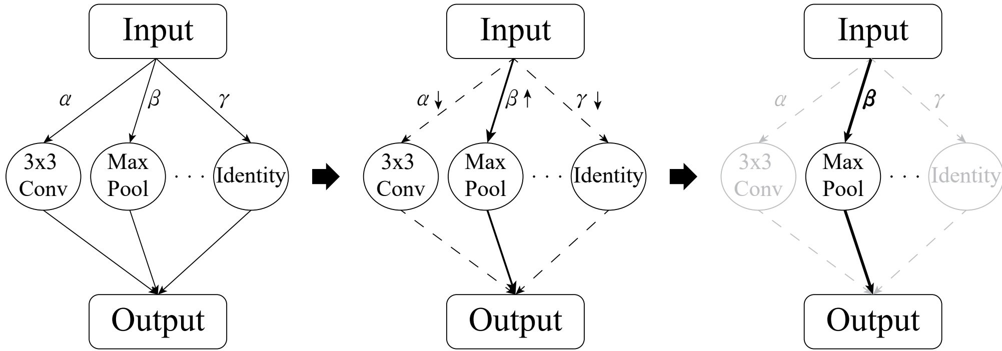


Figure 2.3: Example of gradient-based neural architecture search and continuous relaxation.

based architecture search (DARTS), and it can reduce the network search time dramatically. Previous RL-based and evolutionary search requires a huge amount of time to train each candidate network. However, gradient-based architecture searches train weight parameters, and it also search network architecture concurrently. Due to the network architecture is discrete, it is hard to train the network architecture with conventional training loss. To solve this problem, DARTS propose continuous relaxation for the operation selection. They replace the discrete selection with softmax function, and train architecture parameter to find proper architecture as shown in Figure 2.3. A gradient-based search takes a few GPU days to find network architecture, and it is much faster than previous RL-based search. For this reason, the gradient-based search is widely used in recent work. To improve the performance and efficiency of DARTS, several methods are proposed. ProxylessNAS reduces the memory consumption by using binarization method, so it can support a large-scale tasks (e.g. ImageNet) [23]. DARTS has to compute all operations for the training, so its memory consumption is proportionate to the number of operations. Actually, DARTS uses 8 operations for each edge, so its memory consumption is about 8 times compared with the standard network. Due to this characteristic, DARTS cannot find large-scale network directly. To solve this problem, ProxylessNAS select operation according to the probability, and then train architecture parameters with the binarization method [97]. In forward

propagation, the operation is binarized, and the selected operation is only calculated. In backward propagation, the gradient is calculated the same as conventional back-propagation, and architecture parameters are updated for selected operation as well as non-selected operation. ProxylessNAS also propose the latency penalty, and it can help to find a faster network for the given hardware. They measured the latency of each block for the given hardware (GPU and mobile CPU), and use those measured value for the latency penalty. FBNet [24] and FBNetV2 [98] use similar approaches, but FBNetV2 also support spatial and channel dimension search. The previous gradient-based search uses pre-defined spatial and channel dimensions, and those are regarded as the trainable parameters in FBNetV2. They propose a channel masking method to determine channel dimension dynamically, and resolution subsampling is used for the spatial dimension. They use several masks that have different size, and one mask is selected according to the gradient descent similar to gradient-based operation selection. Recently, several works focus on the latency constraint in the network architecture search. ProxylessNAS and FBNet show the latency estimation method using the linear combination of every block. However, this method cannot estimate the latency of complex network architecture such as DARTS. For this reason, Xu et al. [99] propose the latency prediction model (LPM), and it can predict the latency of the determined network using architecture configuration. By using LPM, latency becomes a differentiable term, so it can be applied for the latency penalty.

Chapter 3

DNN Accelerator with Stochastic Computing

3.1 Motivation

3.1.1 Multiplication Error on Stochastic Computing

In stochastic computing, a multiplier can be implemented with a single AND gate for unipolar encoding and a single XNOR gate for bipolar encoding as we mentioned Section 2.1. The two input stochastic bitstreams are multiplied by a multiplier (AND or XNOR gate), generating outputs cycle by cycle (one bit at a cycle), which is similar to bit-serial logic. The multiplication results are close to the accurate ones but may have some stochastic errors. Figure 3.1 shows the multiplication errors for the two encodings; each point represents the absolute error averaged over $1,000$ multiplication results. The error is maximized when both input values are 0.5 in unipolar encoding, while it is maximized when they are 0 in bipolar. In both cases, the maximum error occurs at the center, but the mean absolute error of the bipolar case is 3.76 times higher than that of the unipolar case.

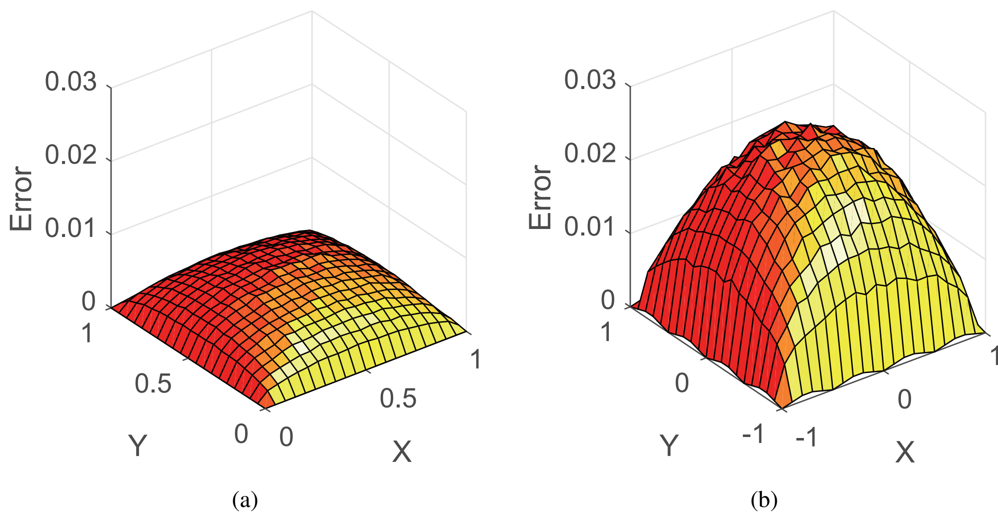


Figure 3.1: Mean absolute errors for multiplications of two 10-bit streams in stochastic computing. (a) Unipolar encoding (AND gate error). (b) Bipolar encoding (XNOR gate error).

3.1.2 DNN with Stochastic Computing

Sigmoid, hyperbolic tangent, and ReLU are commonly used activation functions in DNNs, but ReLU is most popular because it suffers less from the gradient vanishing problem during the backward propagation in a training phase [100], and so deeper networks can be better trained.

When using ReLU in SC hardware, unipolar encoding can be a better choice, because ReLU generates many zero activations (during the forward propagation, ReLU generates 0 at the output when the input is negative). Actually, more than 50% activation values are 0 when ReLU is used [61], so zero values have to be multiplied with weights in the next layer. In bipolar encoding, zero multiplication error is the biggest, and zero value also maximizes switching activity in the hardware. However, zero multiplication is accurate in unipolar encoding, and there is no switching activity. Therefore, unipolar encoding is well-matched with ReLU, and better than bipolar encoding in terms of computation accuracy and energy consumption.

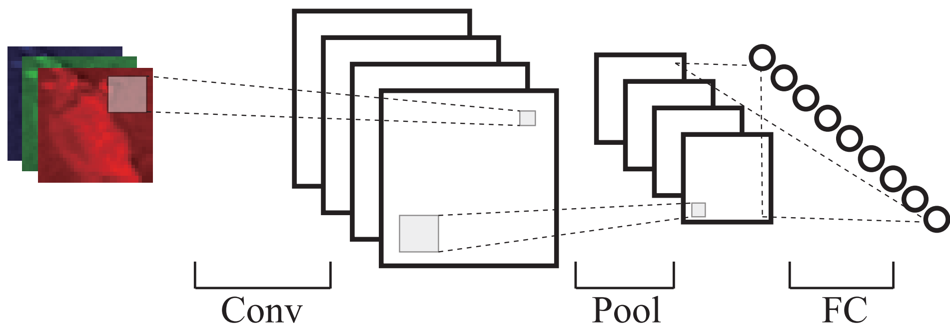
For those problems, unipolar encoding can be a better choice than bipolar encoding. By using unipolar encoding, we can compute near-zero values with a much smaller stochastic error. In addition, we can use unipolar encoding without additional sign bit due to the ReLU function. ReLU function removes all negative values, so every activation value becomes positive. Thanks to this characteristic, we can save a bit-line and improve computation accuracy at the same time. For those reasons, we choose the unipolar encoding for the SC DNN designs, and it is well-matched with DNN.

3.2 Unipolar SC Hardware for CNN

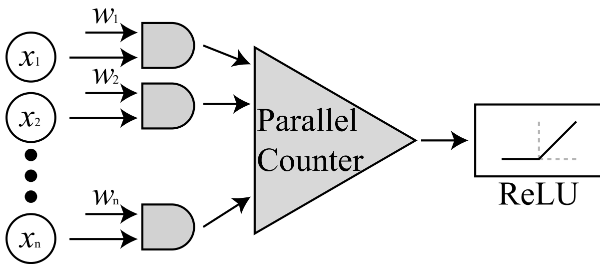
3.2.1 Overall Hardware Design

A CNN consists of three major layers: convolutional layer, pooling layer, and fully-connected (FC) layer. The feature maps of an input image are extracted by the convolutional layer. After that, the pooling layer reduces the size of each feature map, which is directly related to the amount of computation. And then, the FC layer classifies the input image with the resulting feature maps. Figure 3.2(a) illustrates a simplified network and the order of the three basic layers. The layers can be stacked to make a stacked network (deep neural network), which is widely used these days to classify more complex images and to improve the classification accuracy. For example, LeNet-5 [32] is a famous network for MNIST dataset; it consists of two convolutional layers, two pooling layers, and two FC layers.

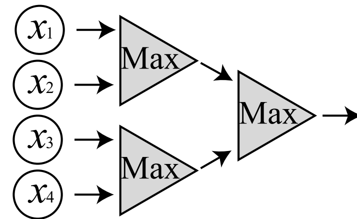
The convolutional layer and the FC layer are two main layers where most of the computations take place. Each of the layers typically contains many neurons and each neuron basically performs MAC operations regardless of the layer types. Actually, we can implement the FC layer with the convolutional layer when its filters cover all pixels of input feature map. Thus the two layers can be implemented with neurons of the same structure. Figure 3.2(b) shows the SC hardware structure of such a neuron. The neuron consists of AND gates for multiplications, a parallel counter for accumulation,



(a)



(b)



(c)

Figure 3.2: Convolutional neural network based on stochastic computing. (a) A simplified convolutional neural network consisting of convolutional, pooling, and fully connected layers. (b) Structure of a stochastic computing neuron, which can be used for both convolutional and fully-connected layers. (c) Max pooling hardware structure.

and a stochastic ReLU module for activation. Stochastic bitstreams coming from the previous layer are multiplied with weight bitstreams using the AND gates. The input bitstreams are always positive or 0 because of ReLU, but the weight values can be negative; thus, the sign of a multiplication result will be the same as the sign of the weight. Therefore, for the multiplication result, we can make the AND gate calculate only the magnitude in unipolar encoding and then take the sign of the weight. For the accumulation of the multiplication results, we use two small adder trees per neuron, one for the group of negative multiplication results and the other for that of positive ones. Because the sign of a multiplication result depends only on the weight fixed by the training, the grouping of the multiplication results is also fixed and the implementation can be easy and efficient. Each adder tree for accumulation is implemented by a parallel counter. In every cycle, the counters count the number of ones in the multiplication results to generate negative and positive sums. The two sums are added together to obtain the final accumulation result for the corresponding cycle. Finally, the output activation bitstream is generated by the stochastic ReLU (SReLU) function. SReLU generates positive unipolar streams from the accumulated values (refer to Section 3.2.2 for the details). Figure 3.2(c) illustrates the max pooling layer consisting of several stochastic max (Smax) functions. In this paper, we mainly use the 2×2 max pooling to implement networks for CIFAR-10 and MNIST datasets because 2×2 is a popular size for a max pooling layer (refer to Section 3.2.3 for the details). For the efficient average pooling, we also proposed designing method and we will cover its detail in Section 3.2.4.

3.2.2 Stochastic ReLU function

The basic concept of SReLU is inspired by the integrate-and-fire (IF) neuron in spiking neural networks (SNN) [101]. Figure 3.3 illustrates the behaviors of integrate-and-fire neuron. The IF neuron integrates all input spikes into its own membrane voltage, and increase or decrease the membrane voltage according to the synaptic weight values. If

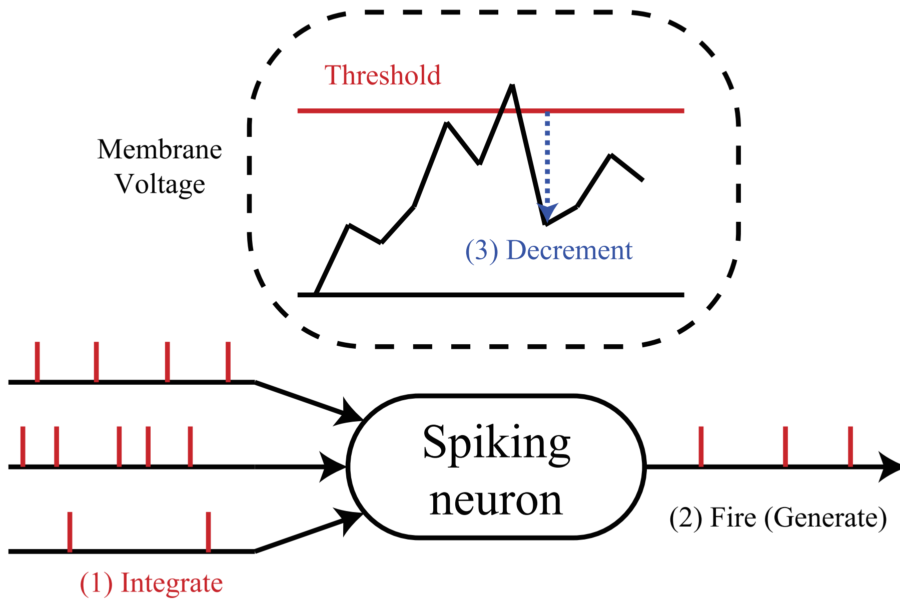
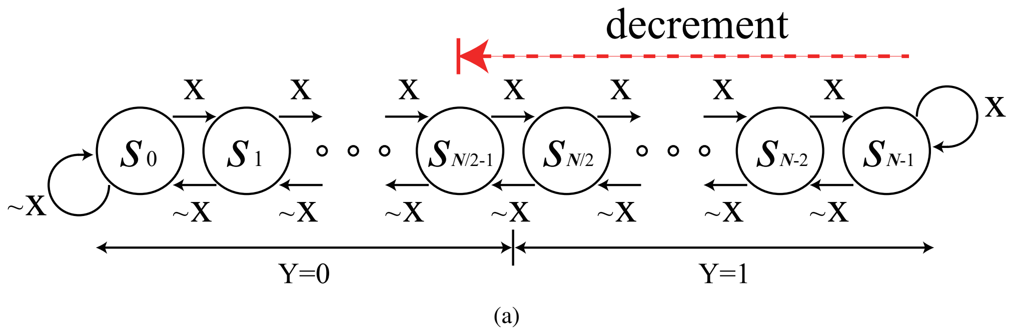


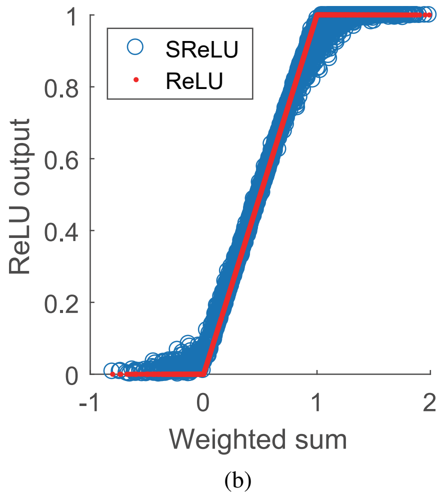
Figure 3.3: Simplified integrated-and-fire (IF) neuron in spiking neural networks.

the resulting membrane voltage becomes larger than a preset threshold, an output spike is generated, and the membrane voltage is decreased by the threshold value. Otherwise, the membrane voltage remains the same. SReLU is an FSM (finite state machine) that mimics the three key operations of the IF neuron: integration, output generation, and decrement. The integration of the input spikes coming from the parallel counter is implemented as a transition to a higher state in SReLU. Differ from the IF neuron, SReLU just integrates accumulated value with its state (membrane voltage) because the weighted sum is covered by AND gates and parallel counter. If the resulting state is higher than the threshold after the integration, the output bit becomes 1, and a transition is made to a lower state to mimic decrement of the membrane voltage. Otherwise, the output bit becomes 0 and no state transition occurs.

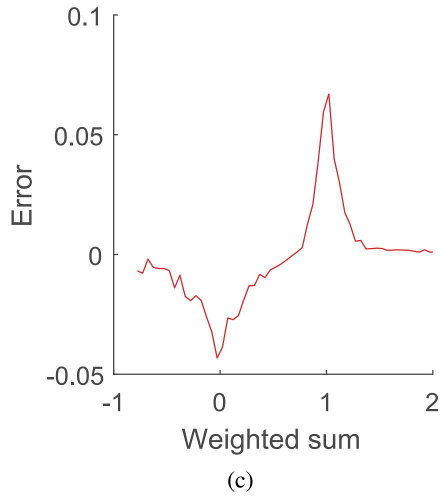
Figure 3.4(a) illustrates the state diagram of SReLU. Each state works as the corresponding membrane voltage. Among the N states, the $(N/2-1)$-th state represents 0 volt, and the lower states and the upper states represent negative values and positive



(a)



(b)



(c)

Figure 3.4: Stochastic ReLU function based on finite state machine. (a) State diagram of SReLU. (b) Results of SReLU and saturated ReLU for $3,000$ random inputs.(c) Mean error of SReLU to approximate saturated ReLU.

values, respectively. The number of states is determined based on the number of input bitstreams so that the FSM can cover the dynamic range of the incoming accumulation result. According to our experience, it is sufficient to set the number of states to twice the number of input bitstreams. For the sake of efficient implementation, we set the number of states to power of two.

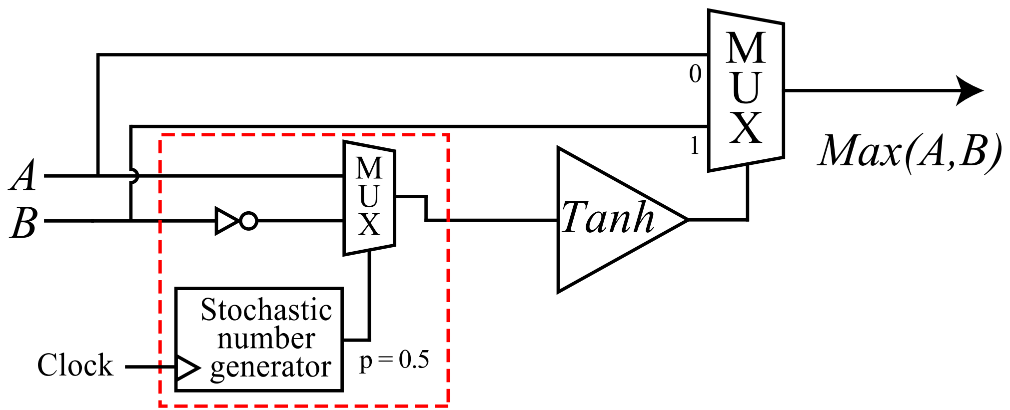
We set the threshold for output spike generation to half of the highest state value. Then the decision to fire a spike depends only on the MSB of the state value. Thus, in every cycle, the SReLU module first checks the MSB of its own state value. If it is 1, the module lowers the state by the threshold value. Then the input value (accumulation result) is added to the state. Note that the output is always the same as the MSB of the state value and thus no additional hardware is needed.

Figure 3.4(b) shows the comparison between SReLU and conventional ReLU with saturation, and Figure 3.4(c) shows mean error of SReLU function. $3,000$ random bitstreams are used to calculate the outputs and the mean errors. The maximum value of SC bitstream is 1 because of range limitation, so the SReLU output cannot express numbers larger than 1. For this reason, as shown in Figure 3.4(b), the output is saturated to 1 when the weighted sum is larger than 1. However, the error due to saturation is not really an issue since the problem can be alleviated by applying normalization, which will be explained in Section 3.3. The accuracy of negative and linear regions is more important than that of the saturation region, but as shown in Figure 3.4(c), the errors are much smaller in those regions.

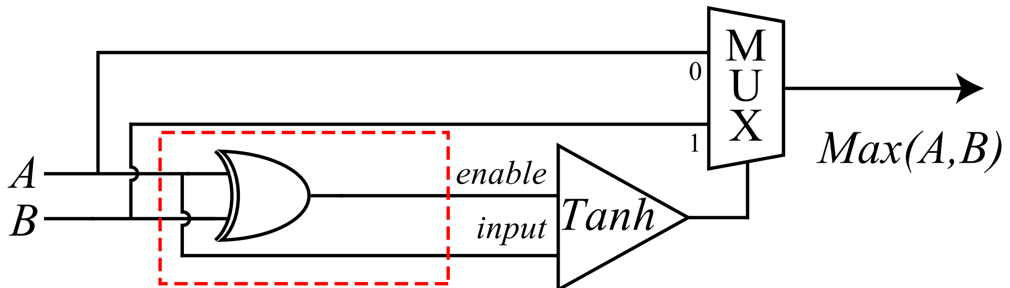
3.2.3 Stochastic Max function

Figure 3.5(a) illustrates the stochastic max function was proposed in [29]. The basic concept of the previous max function is that A increases and B decreases the state of the hyperbolic tangent. This is the reason why B is inverted. And then, A or B is randomly selected with the even probability. It means that we can compare two values by computing average value. If A is larger than B, the state become higher than middle

point. Conversely, the state become lower than middle point when B is larger than A. The stochastic number generators (SNGs) is used to implement those behavior, but it occupies most of its area. For example, an SNG based on a linear feedback shift register (LFSR), which is the most popular digital random number generator, requires 54 gates (NAND-2 equivalent) occupying about 48% of the max function area. We propose an optimized stochastic max (Smax) function shown in Figure 3.5(b) to reduce the SNG overhead. The basic concept of Smax is updating only the difference of the two input SC bitstreams. The difference can be easily calculated with a single XOR gate. In Figure 3.5(b), for example, if A and B are different, the Tanh module (implemented as an FSM) is enabled to update its own state. The input from A to Tanh works as a bipolar-encoded number, so 1 increases the state and 0 decreases it. Thus, if A is larger, Tanh tends to stay on the high state side; if B is larger, it tends to stay on the low state side. The Mux in Figure 3.5(b) selects A when Tanh is at a state higher than half of the highest state. When the enable value is 0 (i.e., A and B are the same), Tanh does not update its state.



(a)



(b)

Figure 3.5: Stochastic max function. (a) The conventional stochastic max function. (b) Our proposed stochastic max function (SMax).

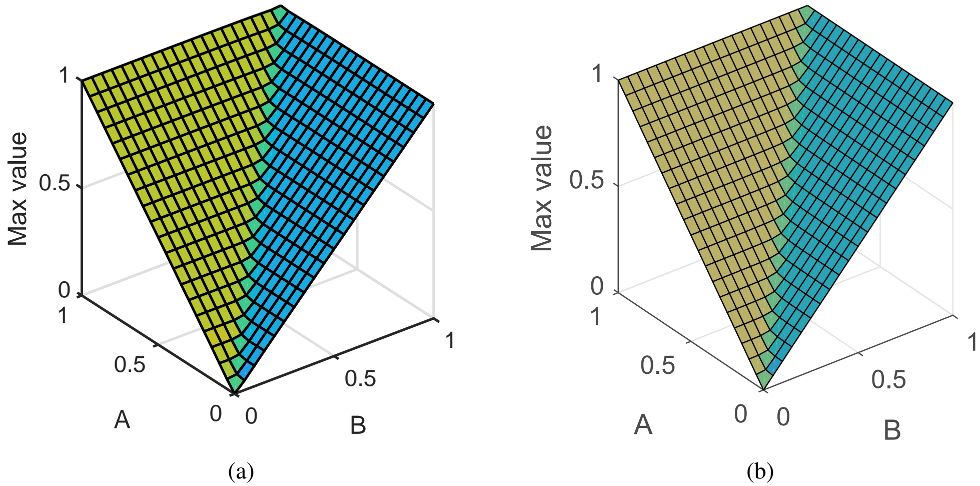


Figure 3.6: Results of the stochastic max functions for $1,000$ random bitstream pairs. (a) The conventional stochastic max function. (b) Our proposed stochastic max function (SMax).

Figure 3.6 shows the max function results for both previous work and our proposed hardware. To check the functionality of each logic, we generated $1,000$ random bitstream pairs for each point. The proposed hardware do not have any random selection logic, but its result is very similar to the previous work in every point. In addition, Figure 3.7 shows the computation errors of each hardware. Figure 3.6(a) shows mean absolute errors for the previous stochastic hardware. The result of the previous hardware shows irregular errors for every points. This phenomenon is caused by the random selection operation. If the larger value is selected more than the smaller value, error can decreases. However, the max operation error can also increases when the smaller value is selected more. For this reason, the error of the previous hardware depends on the randomly generated selection bitstream. The proposed hardware does not have random selection, so its accuracy only depends on the input bitstreams. Figure 3.6(b) shows the computation errors of the proposed hardware. Compared with the previous work, the proposed hardware shows the smaller errors for the every point. Moreover, the proposed hardware shows more regular error patterns. Thanks to this characteristic,

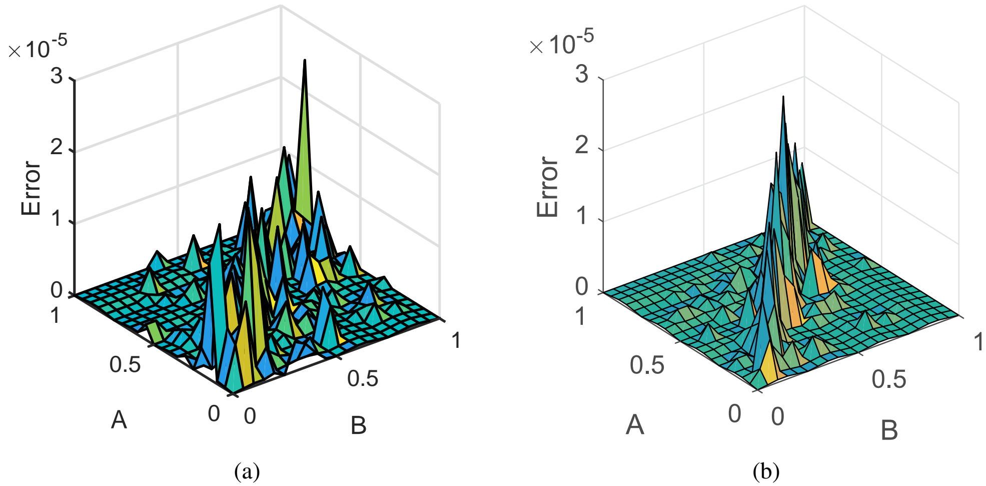
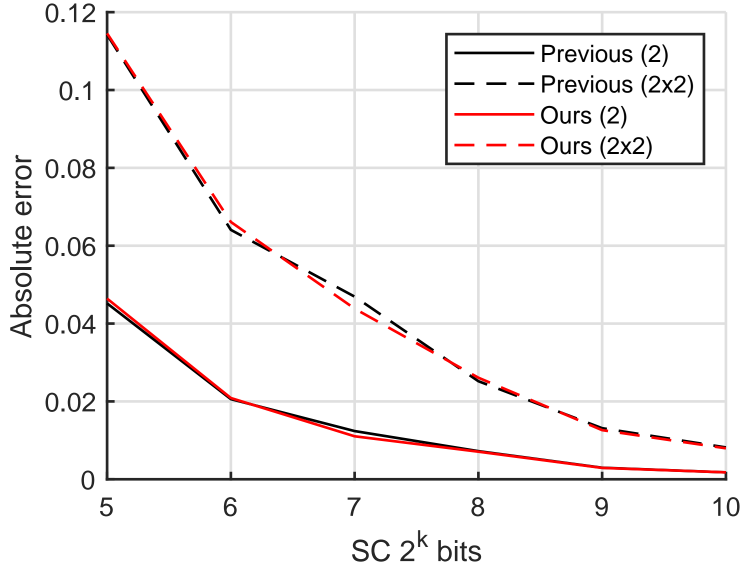


Figure 3.7: Error of stochastic max functions. (a) The previous work. (b) Our proposed max function.

we can handle the computation errors more easily.

By using the cascade max function hardware as shown in Figure 3.2(c), we can design the 2×2 max pooling hardware for DNN. Figure 3.8 shows the absolute mean errors for the max operation and the 2×2 max pooling. The solid line means the standard max operation that select the larger bitstream when two input bitstreams is given. The dashed line shows the error of the 2×2 max pooling for each hardware. The proposed hardware shows the similar to the previous hardware for the every bitstream length. Table 3.1 compares the mean absolute error and gate count (NAND-2 equivalent) for two hardware. The previous hardware requires 111 gates, and the proposed hardware requires only 58 gates. We can save 54 gates by using the proposed hardware (removing SNG), and only 6 gates are required for the XNOR gate and AND gate (enable logic). Therefore, the proposed hardware only have 58 gates, and it is just 52% of the previous hardware.

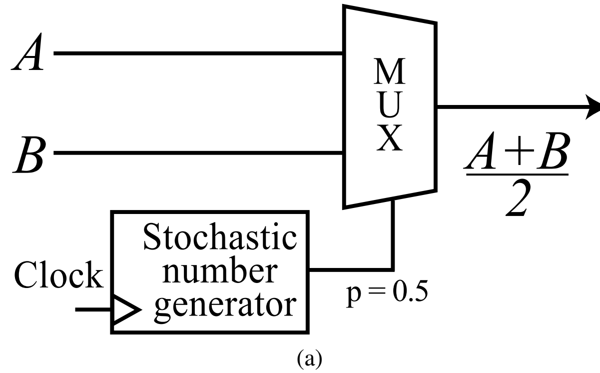


(a)

Figure 3.8: Mean absolute errors of 2-input max operation and 2×2 max pooling hardware. X-axis means the length of input/output bitstreams.

Table 3.1: Accuracy and gate count comparison for max function

	Error		Gate count	
Input	2	2×2	2	2×2
Previous work [29]	0.0018	0.0082	111	333
Proposed	0.0018	0.0080	58	174



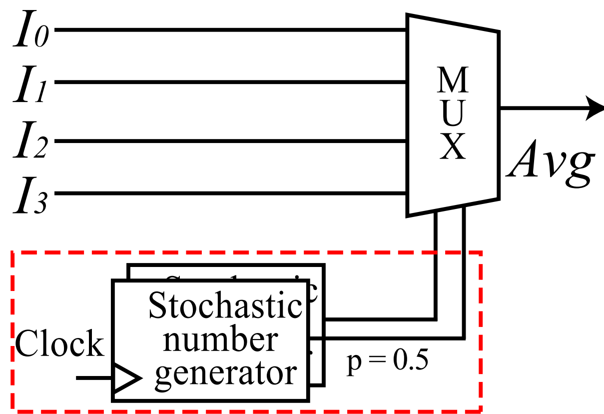
(a)

Figure 3.9: The scaled adder hardware.

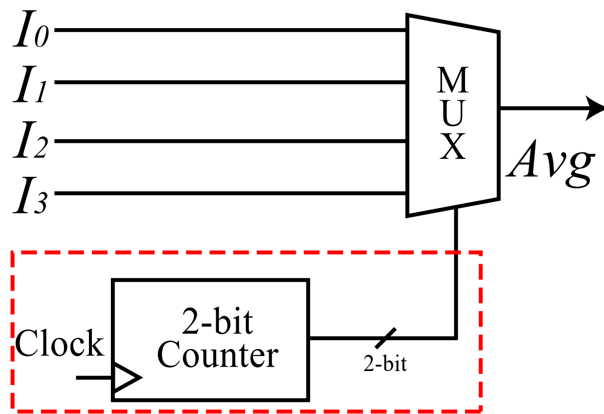
3.2.4 Efficient Average Function

We can simply implement average pooling by using conventional scaled adder logic. Figure 3.9 illustrates the scaled adder. We can compute scaled addition for given values by selecting randomly. In this example, we set the probability to 0.5, so we can obtain averaged value. According to the probability value, this hardware can compute weighted sum for given input bitstreams. For example, if the probability is 0.75, output becomes $(A + 3B)/4$.

To implement average pooling hardware, we can use 4 input mux and two SNGs as shown in Figure 3.10(a). We can obtain averaged result when probability of both SNGs is 0.5. However, the SNGs occupy about 94% of total average pooling area. To reduce SNG overhead, we also proposed new stochastic average pooling hardware (SAvg). Figure 3.10(b) illustrates the proposed SAvg hardware. The basic concept of the proposed hardware is sequential selection. We use the simple counter to select input values, and it has only 11 gates (two SNGs have 108 gates in Figure 3.10(a). Actually, sequential selection may cause the computation error. However, its computation error depends on the correlation between the input bitstreams. If the given input bitstreams are uncorrelated, we can obtain similar result with the scaled adder implementation. In addition, the computation errors that comes from the correlation can also be reduced when the number of inputs increases. By increases the number of input values, the



(a)



(b)

Figure 3.10: 2×2 stochastic average function. (a) The conventional stochastic average pooling function (scaled adder). (b) Our proposed stochastic average function (SAvg).

randomness also increases and correlations are also decreases, so computation errors decreases.

Figure 3.11 shows the mean absolute errors for both average pooling hardware. In this experiment, we use two or four input bitstreams is randomly generated, so the inputs are not highly correlated. The proposed hardware shows the very similar result to the conventional hardware for both 2 and 2×2 (4) inputs experiments. If the inputs are highly correlated, the error of the proposed hardware may increases. However, every arithmetic hardware in SC is vulnerable to the correlation, so it always have to be controlled for the accuracy of overall hardware. For this reason, we does not concern the accuracy for the high correlation case. Table 3.2 shows the exact number for the error of both hardware and gate counts. Both hardware have similar errors but the proposed hardware has much smaller number of gates (NAND-2 equivalent). The proposed hardware does not have SNG, so we can reduce the area effectively. Actually, the proposed hardware has only 8 gates for the 2 average operation and 18 gates for the 2×2 average pooling operation. Compared with the scaled adder based average pooling, the proposed hardware has 3.4% and 1.6% area for 2 and 2×2 average pooling, respectively.

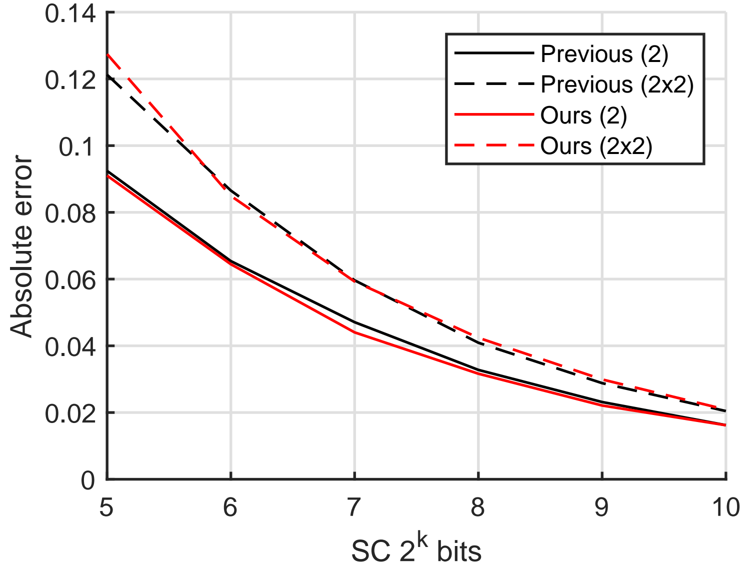
3.3 Weight Modulation for SC Hardware

3.3.1 Weight Normalization for SC

As mentioned in Section 3.2.2, the SReLU output cannot express values larger than 1. However, the problem can be avoided if we can scale down the input value of SReLU. Note that the ReLU function f is homogeneous of degree 1 when scaling factor α is a positive real number, i.e.,

$$f(\alpha x) = \alpha f(x). \tag{3.1}$$

Thus, by scaling the input value with an arbitrary positive real number, one can also scale the output value by the same factor. In other words, one can make the maximum



(a)

Figure 3.11: Mean absolute errors of 2-input average operation and 2×2 average pooling hardware. X-axis means the length of input/output bitstreams.

Table 3.2: Accuracy and gate count comparison for average pooling

	Error		Gate count	
Input	2	2×2	2	2×2
Scaled adder	0.0162	0.0204	58	115
Proposed	0.0162	0.0209	8	18

output value bounded above by 1, through the normalization of the input values with the maximum possible output value that could be obtained without normalization. In addition, composite functions can also be normalized as follows

$$f_1(\alpha \sum f_2(\beta x_i)) = \alpha\beta f_1(\sum f_2(x_i)), \tag{3.2}$$

where α and β are positive real numbers. By using this property, the maximum output values of each layer can be normalized to 1. Moreover, this normalization effect can be obtained by normalizing weight values. Therefore, the saturated ReLU function can be used with normalized weights for the inference in stochastic computing. The weight normalization results in down-scaled outputs, but it is not a problem since the classification accuracy does not depend on the absolute output values but depends on their relative differences.

The weight normalization was introduced for SNN [102]. Basically, SNNs also suffer from the range limitation problem similar to SC, since the spike rate is bounded above by the maximum rate. Thus, max normalization is proposed to solve this problem. For this, all weights are normalized (down-scaled) with the maximum value among the outputs in each layer, so that the maximum output becomes 1.

In SNN, the max normalization also has the effect of normalizing the spike rates. If the spike rate is too low, input information would not be propagated well to the output and the system could be slowed down. This problem can also be avoided by the max normalization. In this case, the weights are up-scaled. In [103], 99.9% normalization is proposed to increase the rates more aggressively and thus further accelerate the SNN inference. 99.9% normalization means that weights are normalized by the 99.9% of the maximum output value. Thus, the output values that belong to the top 0.1% are saturated to 1.

In our SC hardware design, we consider stochastic error as an additional factor. SC hardware has stochastic errors for arithmetic operations, and the impact of the errors has to be minimized to maintain the classification accuracy. When max normal-

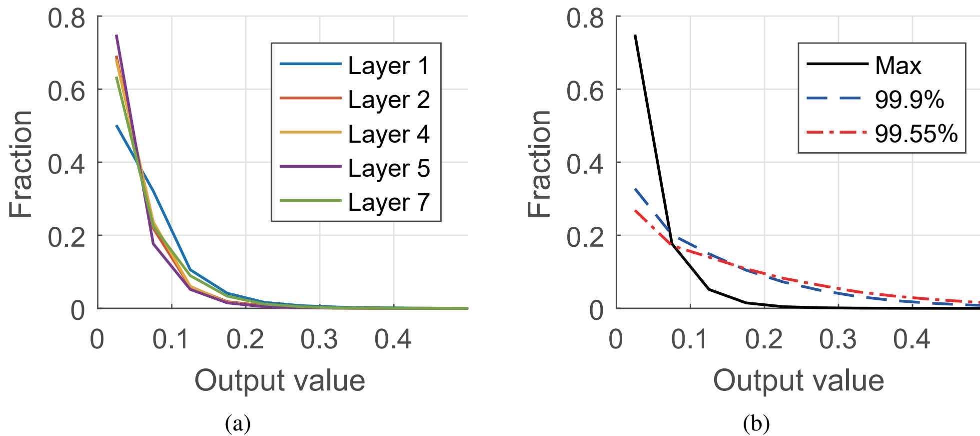
ization is applied, most output values of each layer become close to 0 as shown in Figure 3.12(a). In this case, stochastic errors may affect the accuracy more seriously since the signal levels are relatively low. To reduce the impact of the stochastic errors, output values can be up-scaled by normalizing more aggressively. Figure 3.12(b) compares the results of three different normalizations for layer 5. It shows that 99.9% and 99.55% normalizations considerably decrease the number of near-zero activations compared to the max normalization.

By using the 99.9% and 99.55% normalizations, many output values are scaled up, and thus the number of saturated values increases, thereby degrading the accuracy. As a metric to show the effect of trade-offs between stochastic errors and saturation errors, we define signal-to-noise ratio (SNR) as follows.

$$SNR = \left|\frac{Signal}{Noise}\right| = \left|\frac{Binary}{Binary - SC}\right|, \tag{3.3}$$

where *Binary* is the binary computation result and *SC* is the stochastic computation result. We assume that binary computation result is the correct value. Thus, bigger SNR means more tolerance to the noise. To compare the SNR values of different normalization schemes, we calculate average SNR (ASNR) over 1,000 test images for each layer.

Table 3.3 compares ASNR values for different normalization schemes. It shows that the max normalization has the worst ASNR values. As the number of saturated values increases, the ASNR values increase due to reduced stochastic errors, but if it is increased too aggressively (99% normalization), the ASNR values decrease due to saturation errors. According to our experiment, we find that the ASNR values are maximized at around 99.55%.



(a) (b)

Figure 3.12: Distribution of activations (output values) for each layer. The number of activations is normalized to the number of neurons in each layer, and zero activations are not counted. (a) Comparison of distributions for each layer when max normalization is applied. (b) Comparison of different normalization schemes.

Table 3.3: Average signal-to-noise ratio for different normalizations

	Layer				
Method	1	2	4	5	7
Max Norm.	1.89	1.97	1.78	1.45	1.53
99.9% Norm.	3.55	3.77	3.35	2.46	2.59
99.55% Norm.	4.02	3.67	3.40	2.64	2.82
99% Norm.	3.82	3.35	3.19	2.58	2.75

3.3.2 Weight Upscaling for Output Layer

Most of the ReLU-based networks do not use ReLU in the output layer, so its weight can be up-scaled to improve ASNR. Actually, the output layer is more important than other layers because its result is directly used for classification. To improve ASNR in the output layer, all weights of the output layer can be fully up-scaled. ASNR is improved since the output values increase while the level of noise generated by stochastic errors in the output layer remains about the same. Note that there is no saturation problem since there is no ReLU in the output layer. However, since each weight value is represented by a stochastic number, the weight upscaling is limited to 1. For example, when weights are $[0.1\ -0.5\ 0.35]$, up-scaled weights become $[0.2\ -1\ 0.7]$ and the scaling factor is 2.

An SC network may classify an image differently from a conventional binary networks due to stochastic errors. The problem can be alleviated by weight upscaling since it improves ASNR and thus the effect of stochastic errors is reduced. However, ASNR is not a good metric to show the improvement since it is averaged over all test images and over all neurons in the output layer. To show more clearly how the weight upscaling improves classification accuracy, we introduce the concept of *top-2 difference*, which is the difference between the highest and the second highest output values. It is important because it indicates the reliability of the decision made by the network; if the difference is large, it is highly probable that the decision of selecting the top class (the class having the highest output value) is correct.

For the investigation, we calculate the difference in the output layer for 1,000 test images for two SC networks, one without upscaling and the other with upscaling. We also calculate the difference for a conventional binary network as a reference. Figure 3.13 shows the distribution of top-2 difference for the test images classified differently from the conventional binary network (we expect to obtain the same classification results for both SC and binary networks when the top-2 differences are large, because it is directly related to the reliability of the decision). As expected, the mis-

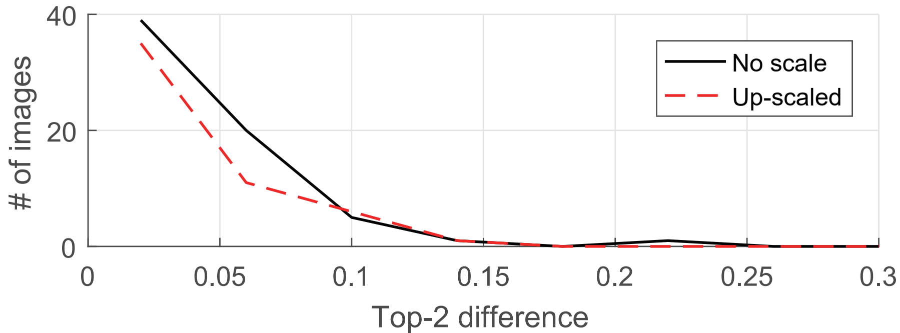


Figure 3.13: Distribution of top-2 differences for misclassified images.

classified images are concentrated in the region of low top-2 difference. By upscaling the weights in the output layer, some of those images become correctly classified, and thus the accuracy is restored to some extent. However, as shown in the figure, many images still remain misclassified since they are mostly affected by the stochastic errors generated in the previous layers.

3.4 Early Decision Termination

The precision of the stochastic number and arithmetic functions in SC is directly related to the bitstream length. In other words, the precision of SC hardware can be adjusted by changing the computation cycles because it decides the bitstream length. As we briefly mentioned in Section 2.1, we can change the precision of computation without any hardware modification. This characteristic is called progressive precision [12]. The early decision termination (EDT) is the technique exploits this characteristic for the fast inference in SC hardware [11]. By using EDT, we can classify the easy images with low precision and the hard images with high precision. Due to precision depends on computation cycle, we can reduce the inference time for the easy images. In previous work, the precision granularity of EDT is set to 32 bits. In every EDT step (32 bits), we make a decision for the confidently classified images. If the confidence is low, we run more steps until its confidence is high enough.

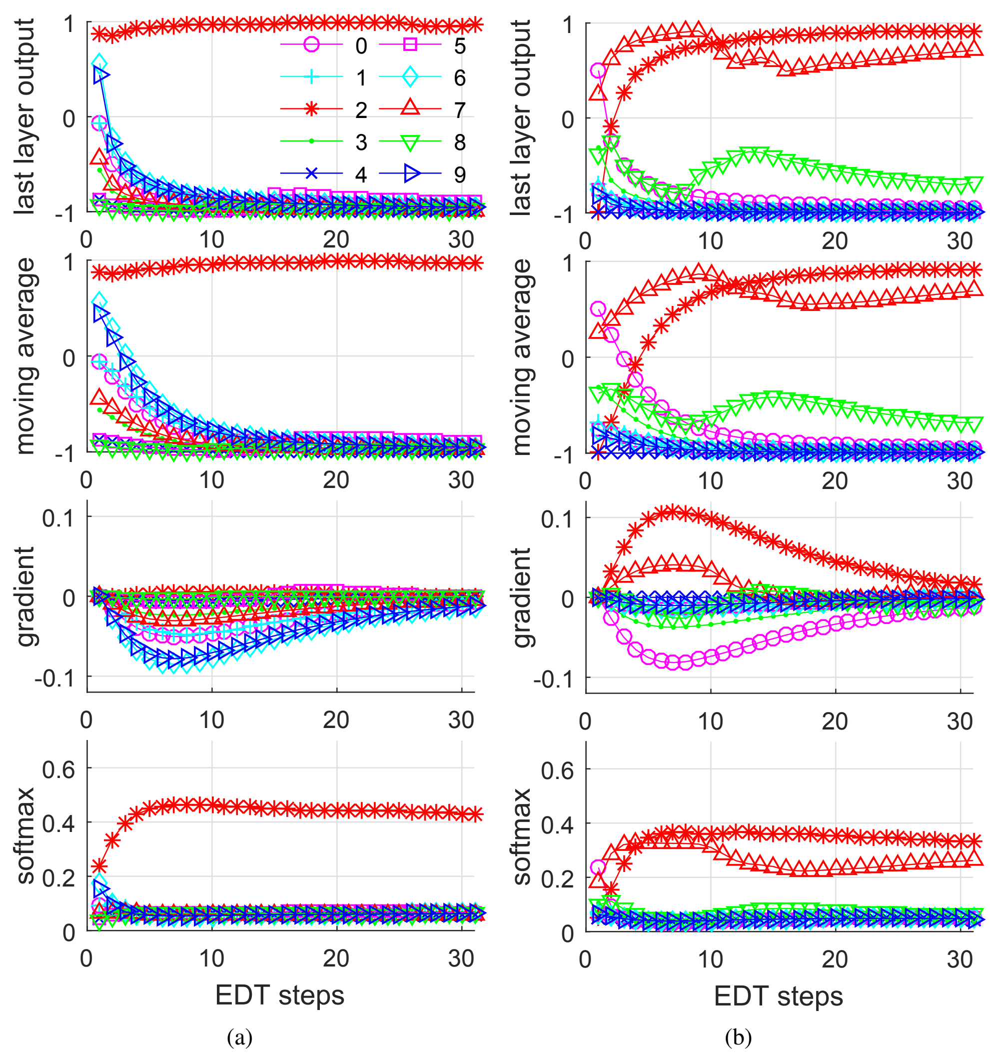


Figure 3.14: Early decision termination steps. Ground truth is 2 in both cases.

Figure 3.14 illustrates the output values and its statistics for the easy and hard examples. In the easy example, the output value of second neuron (class number 2) always higher than the others. However, the prediction for the hard image is reversed according to the EDT steps. To classify correctly for the all kinds of images, several statistical approaches are used.

First, we calculate the moving average values. If the output values are fluctuated, correct decision is very hard. the moving average works as a low-pass filter, so the fluctuation (high frequency noise) can be reduced. The equation of moving average follows,

$$MA_{c,i} = \alpha \cdot Logits_{c,i} + (1 - \alpha) \cdot MA_{c,(i-1)}, \quad (3.4)$$

where c indicates a class index; $Logits_{c,i}$ means the logit value for class c in EDT step i; $MA_{c,1} = Y_{c,1}$; α is empirically set to 0.35. The second row of Figure 3.14 shows the moving average result, and it becomes smoother. And then, we use the gradient value to bring decision time forward. The gradient value indicates the trend of output, so it shows that what values are increasing or decreasing. The final top-1 class (most probable class) will be changed when the gradient of some other class is much higher than that of the current top-1 class. In other words, the output value of some other class steeply increases, more than that of the top-1 class. As shown in the third row of Figure 3.14, the gradient value of the final prediction is much higher than that of the primitive prediction. In addition, the gradient trends shows the final prediction beforehand, so we can reduce the EDT step using this characteristic. To reflect the gradient trends, we defines the objective value in each EDT step as follows,

$$O_{c,i} = MA_{c,i} + \beta \cdot Grad_{c,i}, \quad (3.5)$$

where $Grad_{c,i}$ is the gradient value of class c in the i-th step; β is a weight factor. Finally, we use the softmax function to convert the objective values to class categorical probabilities, which is the same as the classification process of conventional neural

network.

$$SM_{c,i} = \frac{e^{O_{c,i}}}{\sum_k e^{O_{k,i}}}. \tag{3.6}$$

As shown in the fourth row of Figure 3.14, the normalized value of a candidate class represents its relative strength. For example, class 2 dominates other classes in case of Figure 3.14(a), and thus it can be selected as the final decision in an early EDT step. However, for a complex input as shown in Figure 3.14(b), more complex decision rules are needed for an efficient classification.

Algorithm 1 shows the proposed decision rules. In line 7, it processes an EDT step (i.e., 32 bits) to make a decision and its softmax value is calculated by (3.6). If index of the largest value is changed, accumulated value (gap_{accum}) is reset as shown in line 9–10. The current gap between the largest and the second largest value of softmax (gap_{curr}) is calculated in line 12 and the gap is accumulated into gap_{accum} in line 13. With the current gap and the accumulated value, we apply three decision rules. First, if the current gap is larger than max threshold (Th_{max}), the class having the largest softmax value is selected as the final decision as shown in line 20–22. Secondly, if the current gap is larger than min threshold (Th_{min}) and increases gradually during threshold number of steps (Th_{step}), we also select the class having the largest softmax value in line 23–25. Thirdly, if the gap is lower than the min threshold but the accumulated value is higher than a threshold (Th_{accum}), we select the class having the largest softmax value. Four threshold values (Th_{max}, Th_{min}, Th_{step}, and Th_{accum}) are empirically set to 0.4, 0.12, 5, and 3, respectively. If the class is not decided until the last EDT step, the class of largest softmax is selected as a general decision process in line 33. The portion of the four kinds of decision rule (current gap dominant, gradual increment, accumulative, and general decision) are 29.82%, 60.33%, 7.65%, and 2.20%, respectively, in our experiment.

Algorithm 1 Mixed Threshold Decision

// SM_{max} is the largest value of softmax $SM_{c,i}$.
// SM_{second} is the 2nd largest value of softmax $SM_{c,i}$.
// idx_{max} is the index of SM_{max}.
// gap_{curr} is defined as $SM_{max} - SM_{second}$.
// N_{step} is the maximum number of EDT steps.
// $Th_{max}, Th_{min}, Th_{step}$, and Th_{accum} are user-defined thresholds.
OUTPUT: Y is a final decision

```
 1: idx_prev_max ← 0                                  // idx_max value from previous step.
 2: gap_accum ← 0                                     // accumulated value of gap.
 3: gap_prev ← 0                                      // gap from previous step.
 4: count_step ← 0                                    // incremental count.
 5:
 6: for i = 1 to N_step do
 7:     [idx_max, SM_max, SM_second] ← run_single_step()
 8:
 9:     if idx_max ≠ idx_prev_max then
10:         gap_accum ← 0
11:
12:     gap_curr ← SM_max − SM_second
13:     gap_accum ← gap_accum + gap_curr
14:
15:     if gap_curr > gap_prev then
16:         count_step ← count_step + 1
17:     else
18:         count_step ← 0
19:
20:     if gap_curr > Th_max then
21:         Y ← idx_max
22:         return
23:     if gap_curr > Th_min && count_step > Th_step then
24:         Y ← idx_max
25:         return
26:     if gap_accum > Th_accum then
27:         Y ← idx_max
28:         return
29:
30:     idx_prev_max ← idx_max
31:     gap_prev ← gap_curr
32:
33: Y ← idx_max                                       // early decision failed.
```

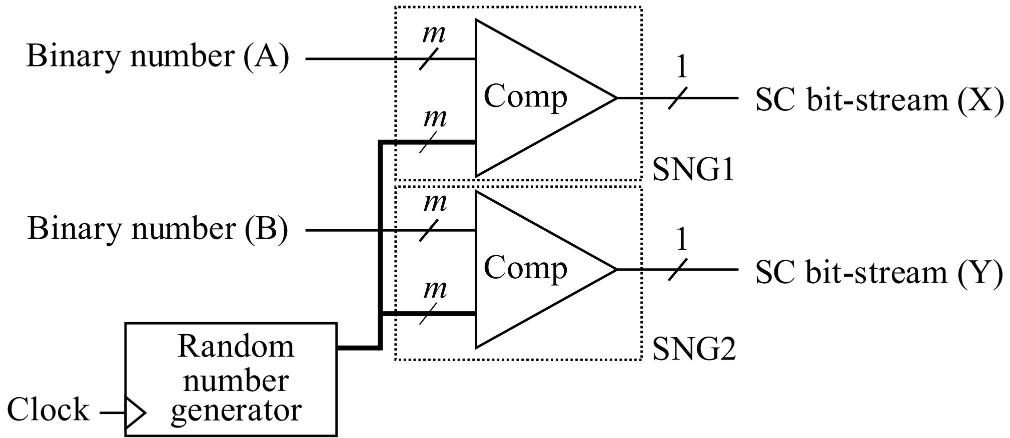


Figure 3.15: Simplified sharing of a random number generator among the different stochastic number generators.

3.5 Stochastic Number Generator Sharing

The overhead of stochastic number generator (SNG) is one of the most serious problems because its area and power consumption are much larger than those of other SC circuit elements. A conventional SNG consists of an m-bit random number generator (RNG) and an m-bit comparator, and is used to convert a conventional m-bit binary number to a 2^m-bit stream. For the inference with SC hardware, each of primary input, primary output, and all weights require an SNG to generate SC bitstreams from binary numbers. Therefore, SNG have to be used for every input connection to generate weight bitstreams, so it occupies huge amount of area. For instance, a 10-bit linear feedback shift register (LFSR) based SNG requires 105 gates (NAND-2 equivalent) whereas an AND gate takes only 1.5 equivalent gates; according to our experiment, SNGs take about 85% of total area in a 200 input neuron. There are several researches on efficient RNG designs [104, 105] to reduce the SNG overhead. Since it is hard to implement an efficient RNG using conventional digital logic, [104] exploits the random switching behavior of a nanomagnet, and [105] uses the characteristics of random telegraph noise (RTN). Both of them can generate random numbers more efficiently

than a conventional LFSR based RNG. Another research focuses on sharing an RNG among different SNG [106] as shown in Figure 3.15. This method can generate SC bitstreams with only one RNG, but the SC bitstreams generated by a shared RNG are highly correlated. In SC, correlation between bitstreams can adversely affect the behavior of SC arithmetic operations [107], e.g., AND multiplication accuracy can be severely degraded if the two inputs have high correlation.

To reduce the number of SNGs in a CNN, we exploit the relationship between correlation in SC and CNN arithmetic operations. In a neuron, the inputs are multiplied with the weights, accumulated, and fed to the activation function. Our SC design first multiplies input bitstreams with weight bitstreams, accumulates products, and then calculates the neuron output bitstream using SReLU function. The correlation between input and weight bitstreams can severely impact the accuracy of AND operation, so weights and inputs must be generated using different RNGs. In addition, different input-weight pairs must use different RNGs to avoid a biased accumulation result. For example, if an RNG is shared by multiple weights, the distribution of 1s or 0s will be biased column-wise, and thus can cause degeneration of SReLU accuracy. Therefore, RNG sharing within a single neuron (intra-neuron RNG sharing) degrades SC arithmetic accuracy, and thus should be avoided.

On the other hand, inter-neuron RNG sharing may not cause much accuracy degradation. In a convolutional layer, local input features of a layer are fetched from the previous layer with a sliding window with weights (i.e., filter weights) as shown in Figure 3.16. Each location of the window has its own neuron and the input features fetched at a location are fed to the corresponding neuron. The neurons using the same window can share the weight bitstreams provided that the sharing does not generate correlated outputs. Since the primary inputs and weight bitstreams are uncorrelated, the correlation between the inputs and the weight bitstreams is not a problem for the first layer (we assume that the primary inputs are not correlated with each other), and thus each neuron in the first layer can generate accurate outputs. We observe that the

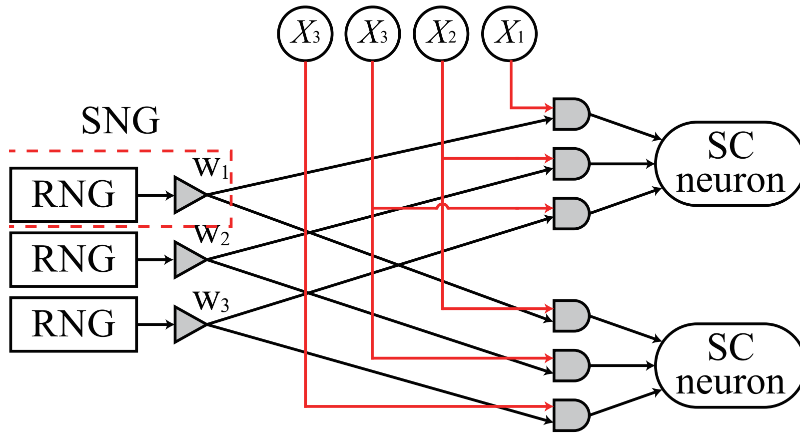


Figure 3.16: Stochastic number generator sharing method for the convolutional layer.

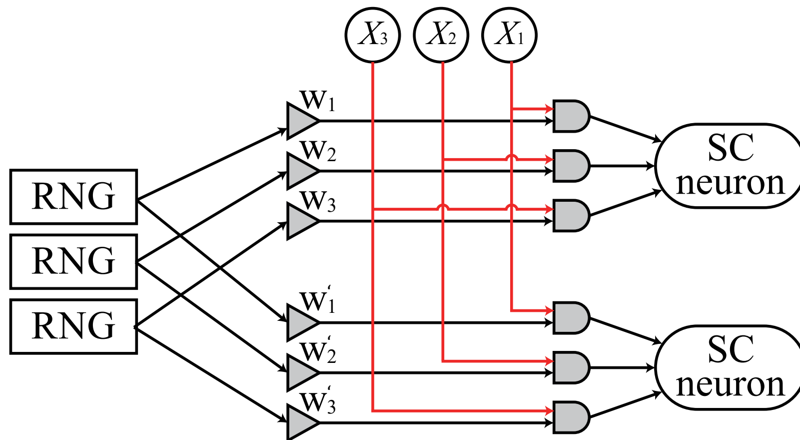


Figure 3.17: Stochastic number generator sharing method for the fully-connected layer.

generated output bitstreams are also uncorrelated with each other even though they share the same weight bistreams. This is because the inputs to individual neurons are different due to the sliding of the window and the randomness of the output bitstream is determined by the combination of the inputs as well as the weight bitstreams. Fortunately, inputs are changed according to the location of the sliding window, and thus different neurons have uncorrelated inputs (overlapped window is not a problem since the neurons will have different order of inputs) at least for the first layer (we assume that the primary inputs are uncorrelated). The uncorrelated output bitstreams are fed to the second convolutional layer, which in turn allows the neurons in the second layer to generate uncorrelated output bistreams with shared weight bitstreams. In this way, weight bitstreams can be shared among the neurons that use the same filter in a convolutional layer without accuracy drop.

In an FC layer, correlation among weight bitstreams for different neurons can affect the accuracy of the next layer since the neurons have the same input bitstreams. Figure 3.17 illustrates the RNG sharing method for the FC layer. However, its impact can be alleviated when networks are pruned, since different neurons will have connections to a different (possibly overlapped) set of neurons in the previous layer, and thus the randomness of their output bitstreams increases. Thanks to this property, RNG can also be shared in FC layers.

We apply different RNG sharing schemes to convolutional and FC layers. Because the weights are shared in a convolutional layer, the entire set of SNGs for a filter can be shared, i.e., the weight bitstreams for a sliding window are shared among all neurons, one for each placement of the sliding window. However, weights are not shared in an FC layer, so the RNG (not entire SNG) sharing scheme proposed in [106] is used. In this case, RNGs used for a neuron are shared by other neurons, but different comparators are used to generate their own weight bitstreams as shown in Figure 3.15. By applying the proposed SNG and RNG sharing, orders of magnitude reduction of the number of both SNGs and RNGs can be achieved.

3.6 Experiments

For the experiment, we use CIFAR-10 and MNIST dataset. CIFAR-10 consists of $60,000$ 32×32 RGB images of real objects in 10 classes. Among them, $50,000$ are training images and $10,000$ are test images. MNIST consists of $70,000$ 28×28 grayscale handwritten digit images. Among them $60,000$ are for training and $10,000$ are for test.

The proposed SC networks consist of SReLU and Smax modules presented in this paper, and 2^{10}-bit stream is used for both CIFAR-10 and MNIST experiments. All the neurons in the convolutional and FC layers are designed as shown in Figure 3.2. In the output layer, there is no SReLU function, so it consists of AND gates for multiplications and parallel counters for accumulations. For the accuracy simulation, software RNG is used to generate SC bitstreams, and SNG sharing is applied to every convolutional layer.

3.6.1 Accuracy of CNN using Unipolar SC

The networks used in the CIFAR-10 experiment have 2Conv-1Max-2Conv-1Max-2FC layers, and zero padding is not used. The convolutional layers have 32, 32, 64, and 64 filters, respectively, and all the filters have 3×3 sliding window with a stride of 1. The max pooling layers have a 2×2 sliding window with a stride of 2, and the FC layers consist of 512 and 10 neurons, respectively. The networks are trained and pruned with floating-point computations, and the SC hardware is used only for inference.

Figure 3.18 shows test error of both floating-point and proposed SC on CIFAR-10. The test error of SC is 23.40%, which is 7.26% point higher than that of floating-point, when max normalization is used. When 99.9% normalization is applied, its test error becomes much lower but still higher than the floating-point case by more than 1% point. The proposed 99.55% normalization lowers the test error to 16.84% and further lowers down to 16.43% with weight upscaling. The test error becomes slightly larger

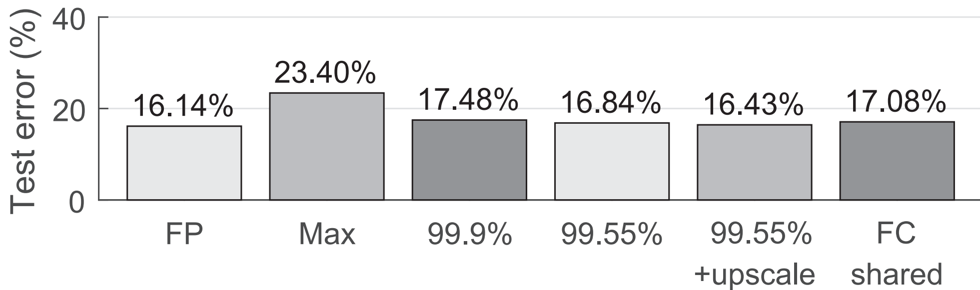


Figure 3.18: Test error for CIFAR-10 dataset. The test error of floating-point is 16.14%. In SC, test error is minimized when both 99.55% normalization and weight upscaling are used, and its test error is 16.43%. In addition, test error becomes 17.08% when RNG sharing is applied to fully-connected layer.

when RNG is shared in the FC layers. Its degeneration comes from the correlation effect mentioned in Section 3.5, but the gap with the floating-point case is lower than 1%.

Figure 3.19 shows the result of early decision termination (EDT) applied to the CIFAR-10 dataset. One EDT step consists of 32 bits, so there are 32 EDT steps because 2^{10}-bit stream is used. Many images still need 2^{10} precision for classification, but Figure 3.19 shows that more than half of the input images can be classified with lower than 2^9 precision. Compared with the baseline SC having accuracy of 17.08%, the application of EDT decreases the accuracy by only 0.09% point but the energy consumption decreases to 50.6%.

In the MNIST experiment, a LeNet-5 network with 1Conv-1Max-1Conv-1Max-2FC layers is used. The convolutional layers have 20 and 50 filters, respectively, and all the filters have a 5×5 sliding window with a stride of 1. The max pooling layers have a 2×2 sliding window with a stride of 2. There are 500 and 10 neurons in FC layers, respectively. The LeNet-5 network is also trained and pruned with floating-point computations.

The test error of the proposed SC network is 0.81%, while the test error of the

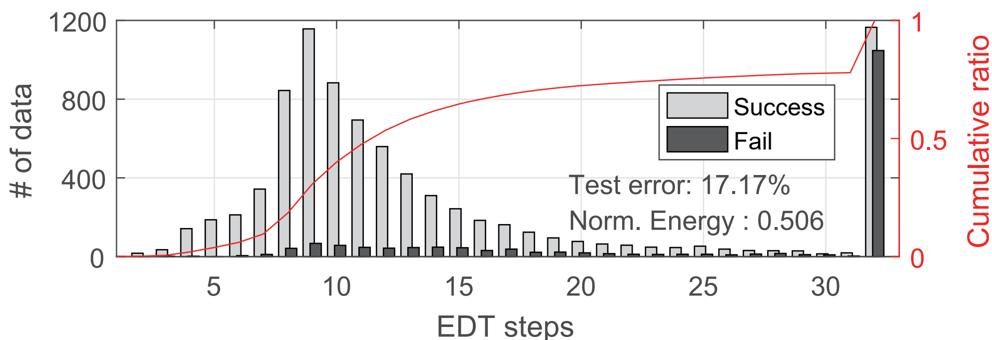


Figure 3.19: Early decision termination result on CIFAR-10 dataset. More than half of the test data can be classified with lower than 2^9-bit precision, but 20% of the data still require 2^{10}-bit precision.

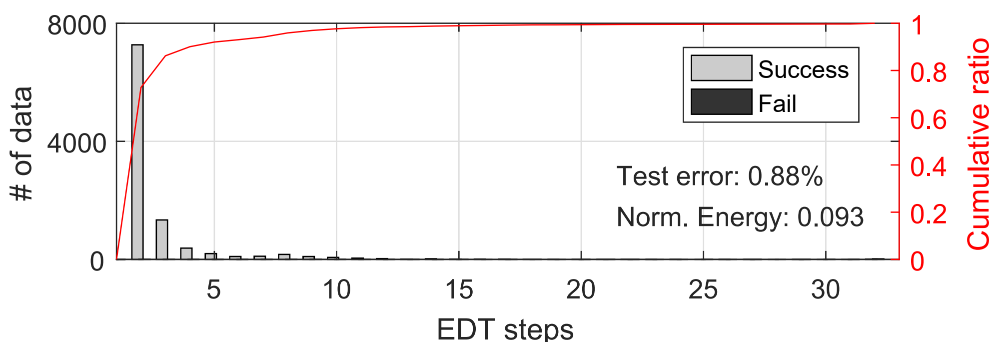


Figure 3.20: Early decision termination result on MNIST dataset. Almost test data can be correctly classified with 2^6-bit precision.

Table 3.4: Comparison with previous works in terms of configuration and test error

	Configuration			Test error	
SC Designs	Encoding	Activation	Pooling	MNIST	CIFAR-10
ICCD 16 [13]	Bipolar	Tanh	Avg	3.65%	-
ASPLOS 17 [14]	Bipolar	Tanh	Max	1.74%	-
DATE 17 [15]	Bipolar	Tanh	Max	1.06%	-
IJCNN 17 [16]	Bipolar	ReLU	Avg	1.69%	-
Proposed SC	Unipolar	ReLU	Max	0.88%	17.17%

floating-point network is 0.77%. There is only 0.04% point difference between the two networks. MNIST consists of images that have very high contrast as mentioned in Section 1.1, so it does not need high precision. Figure 3.20 shows the EDT result of MNIST, and it also shows that MNIST can be classified with only very low precision.

Table 3.4 compares the proposed approach with the previous ones. All the previous networks use LeNet-5 with bipolar encoding, and only [16] uses ReLU function. The proposed unipolar SC not only well classifies the CIFAR-10 images, but also classifies the MNIST images with the lowest test error. The accuracy improvement of the proposed SC network over the previous ones on the MNIST dataset is not big, but it can be much bigger on CIFAR-10. The reasoning is as follows. The multiplication error in bipolar encoding is 3 times higher than that of unipolar encoding as shown in Figure 3.1. Figure 3.19 and Figure 3.20 show that CIFAR-10 requires much more accurate computation than MNIST. In addition, deep networks with hyperbolic tangent cannot be well trained because of the gradient vanishing problem. Even if ReLU is used, its accuracy can be degenerated seriously because of zero multiplication error mentioned in Section 3.1.1. Thus, our approach can work better for a bigger network for more difficult classification problems.

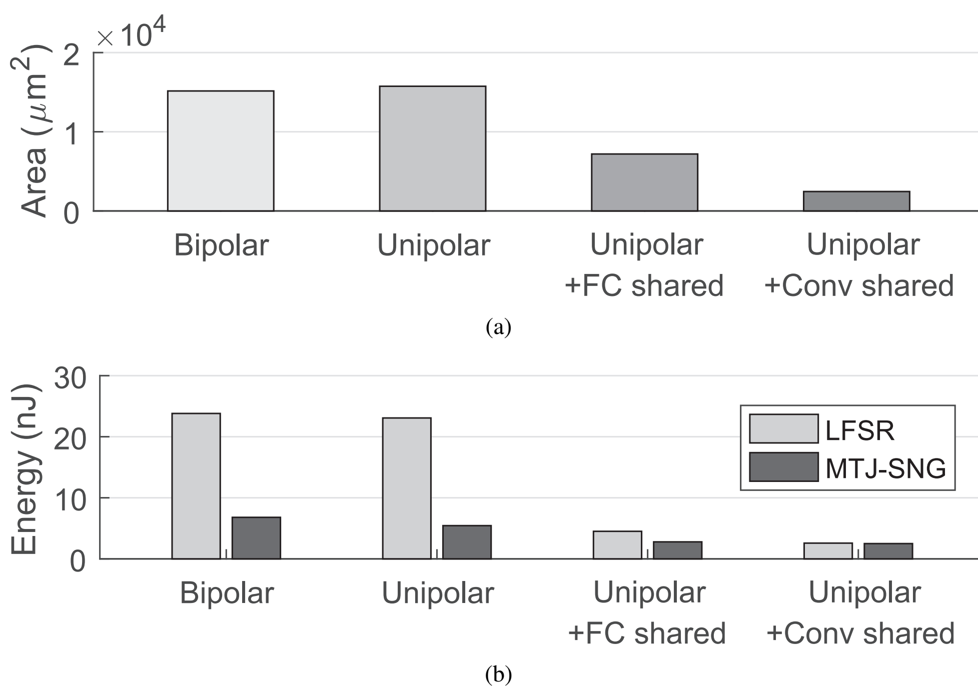


Figure 3.21: Synthesis result of conventional bipolar neuron and proposed unipolar neuron. (a) Area comparison of bipolar and unipolar neuron. LFSR is used to calculate area in all neuron. (b) Energy consumption of each neuron. All circuit execute 2^{10}(=1024)-bits.

3.6.2 Synthesis Result

We synthesize one SC neuron with 200 inputs for the conventional bipolar network as well as the proposed SC network. In unipolar encoding, the proposed SNG sharing scheme is applied to both convolutional neuron and FC neuron. The FC neuron requires its own comparators to generate its weight bitstreams, but the convolutional neuron does not need its own comparators because weights are shared. Thus, the hardware cost of the convolutional neuron is much lower than that of the FC neuron. To synthesize each type of the neurons, TSMC 45nm technology library and Synopsys Design Compiler are used. An LFSR is used to calculate the area, and LFSR and MTJ-SNG [104] are used for comparison of energy consumption.

Figure 3.21 shows the synthesis result in terms of area and energy consumption, and total delay of the entire neuron is 0.6 ns. Without SNG sharing, the unipolar neuron has similar area compared with the bipolar neuron. When SNG is shared, area of the FC and convolutional neurons is decreased to 47.5% and 16.2%, respectively, compared with the bipolar neuron. When LFSRs are used, the FC neuron and the convolutional neuron with SNG sharing respectively have $5.3\times$ and $9.2\times$ higher energy efficiency, compared with the bipolar neuron. When MTJ-SNGs are used, they still have $2.4\times$ and $2.7\times$ higher energy efficiency, compared with the bipolar neuron.

3.7 Summary

In this work, we proposed new hardware designs and methods to implement CNN hardware using unipolar SC. We proposed SReLU and Smax function, and also propose SNG sharing scheme in CNN, which can reduce the SNG overhead. In addition, we also propose 99.55% weight normalization and weight upscaling scheme to improve SNR of SC operations. Our experimental results show that the accuracy of the SC network is close to that of the floating-point network on MNIST and CIFAR-10 datasets. Our approach outperforms the previous ones on MNIST. Our synthesis result also shows the efficiency of the proposed SC network in terms of area and energy consumption.

Chapter 4

Neural Architecture Transformation

4.1 Motivation

Deeper network architectures help to achieve higher accuracy, but those have a huge amount of parameters and computation redundancies. To design a compact network architecture, the 1×1 convolution is added [2, 4, 6]. The additional 1×1 convolution reduces the number of channels of output activation. The number of parameters and multiplications in the 3×3 convolution is also reduced thanks to the 1×1 convolution. For this reason, the bottleneck block in ResNet and dense block in DenseNet use the 1×1 convolution for the parameter and multiplication reduction.

However, the bottleneck and dense block actually increase inference time even though the number of multiplications is reduced. Figure 4.1 shows the number of multiplications and actual inference time for three models of ResNet and DenseNet. ResNet-50 has a number of multiplications similar to that of ResNet-34 thanks to the 1×1 convolution, but it takes $1.8\times$ longer than ResNet-34. This is because the bottleneck block of ResNet has four times larger activation map compared with the basic residual block, so it causes four times more activation load from off-chip memory. Although DenseNet has a much smaller number of parameters and multiplications compared with ResNet, its inference time is much longer than that of ResNet. DenseNet has

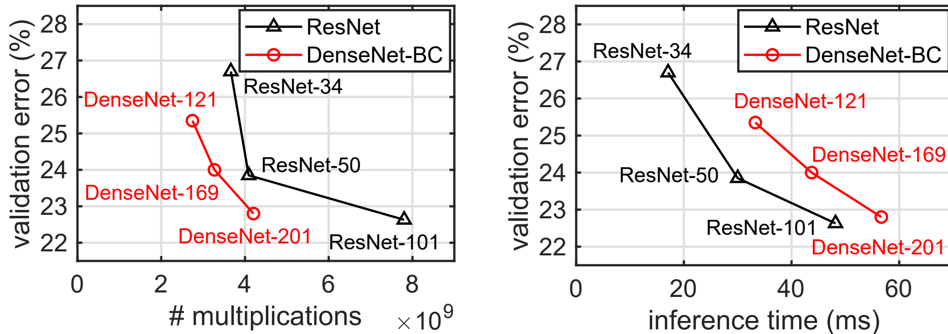


Figure 4.1: ResNet and DenseNet Top-1 validation errors for different numbers of multiplications (*left*) and inference times (*right*). To measure the inference time, single NVIDIA Titan X (Pascal) is used and batch size is set to 16. DenseNet has much fewer multiplications than ResNet, but its inference time is much longer.

much smaller total activations than ResNet, but the actual activation load of DenseNet is much larger than that of ResNet because the layers in DenseNet use output activations of all previous layers and thus actual activation load is much larger than the total activation size.

In this work, we focus on the inference time reduction rather than parameter and multiplication reduction. To reduce the inference time, we propose the *network recasting* method by transforming the network architecture for a smaller activation load. We transform the network architecture through the block-wise recasting of source blocks into target blocks. The recasting is done by training the target block to mimic the output activation of the source block, so the accuracy can be preserved after recasting. We can obtain a *mixed-architecture network* by recasting parts of the trained network. By the mixed-architecture network, we mean a network having multiple types of the block that can exploit the advantages of individual block types within a single network. In addition, we can use the network recasting method for network compression by recasting each block to a smaller one of the same type.

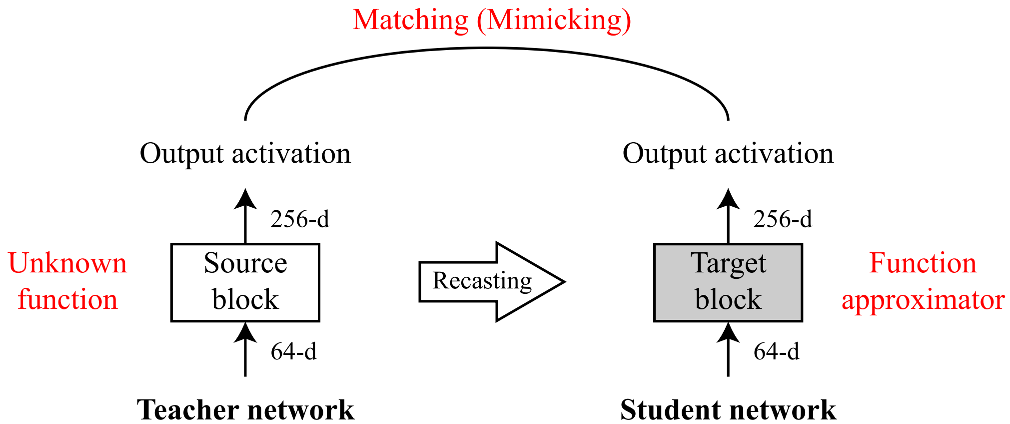


Figure 4.2: Basic concept of the network recasting. The target block of the student network is trained by mimicking the source block of the teacher network.

4.2 Network Recasting

The network recasting method recasts a pre-trained network into a network of different type and/or size. Figure 4.2 illustrates the basic concept of the proposed network recasting method. Given the pre-trained teacher network, we transform each block (*source block*) in the teacher network into a new block (*target block*) of pre-defined type and size in the student network. The transformation is done by training the target block to generate output activations similar to those of the source block. We call this process *block recasting*. In this process, the source block can be considered as an unknown function, and the target block can be considered as a functional approximator similar to a multi-layer perceptron [31]. After recasting all candidate blocks, we obtain the student network, which is faster than the teacher network while preserving the functionality or accuracy. We call the entire process *network recasting*.

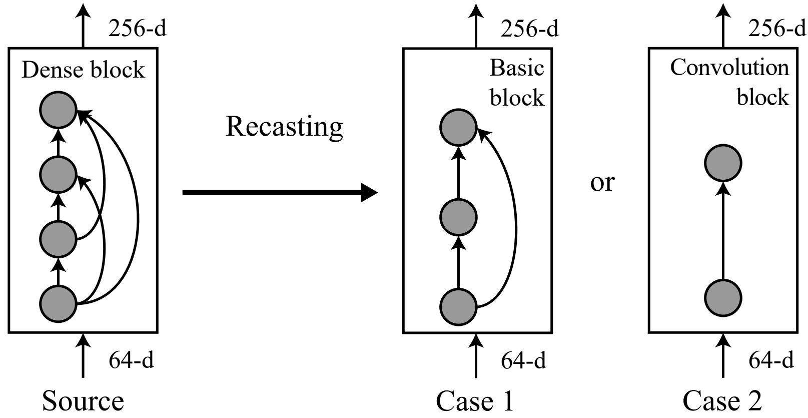


Figure 4.3: Block recasting of a dense block into a basic block (Case 1) and a convolution block (Case 2). The basic block has shorter inference time than the dense block because it has much smaller activation load. The convolution block is even faster than the basic block, but its capacity is much smaller and so it can cause accuracy loss.

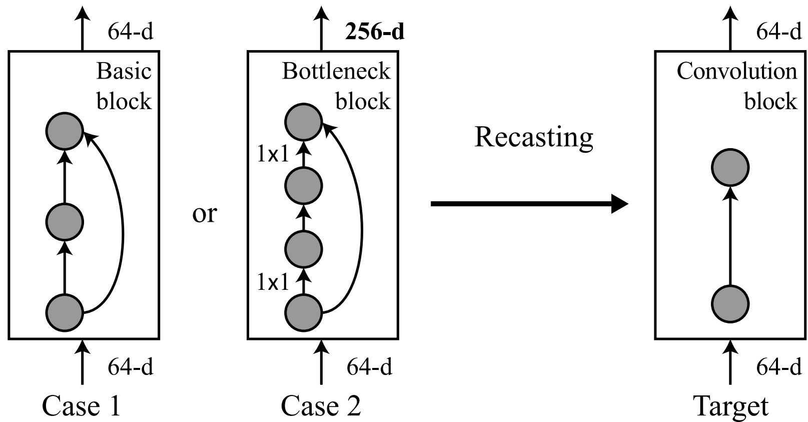


Figure 4.4: Block recasting of a residual block—basic block (Case 1) and bottleneck (Case 2)— into a convolution block. The recasting of the basic block keeps the same number of input and output channels. However, since the bottleneck block uses a smaller number of channels for the feature extraction, we recast it into a convolution block that has the same number of input and output channels as the original 3×3 convolution.

4.2.1 Recasting from DenseNet to ResNet and ConvNet

The DenseNet has a lot of activation load due to the dense connection, and by recasting the dense block into a basic residual block (we call the basic residual block as *basic block* for simplicity), we can reduce the inference time. We consider a basic block consisting of two 3×3 convolution and shortcut as shown in Figure 4.3. Even though the basic block has more parameters and multiplications than the dense block, its activation load is much smaller and thus it is much faster. For more inference time reduction, we can recast the dense block into a single convolution block, although it can cause more accuracy loss because it has a very small capacity. Figure 4.3 shows the two examples of recasting the first dense block in DenseNet-121.

4.2.2 Recasting from ResNet to ConvNet

Figure 4.4 illustrates the block recasting of a residual block into a convolution block. In the basic block, local features are extracted from the input activations using 3×3 filters, and thus, we recast the basic block into a 3×3 convolution block. Since the new convolution block has the same number of filters as the original basic block, the dimension of the output activations is not changed. However, in bottleneck block recasting, the dimension of the output activation is reduced as shown in Figure 4.4 (Case 2) for the first bottleneck block of ResNet-50. Although the output activation becomes smaller, the number of linearly independent features is not changed because the second 1×1 convolution in the source block just combines its input activations linearly to extend the dimension of output activation. Therefore, the next block in the student network still can reconstruct similar activation map.

4.2.3 Compression

The network recasting can be used to compress the large network while preserving accuracy. In this case, we assume that the network has redundancy such as ineffectual filters and redundant filters. An *ineffectual filter* denotes a filter that cannot extract any

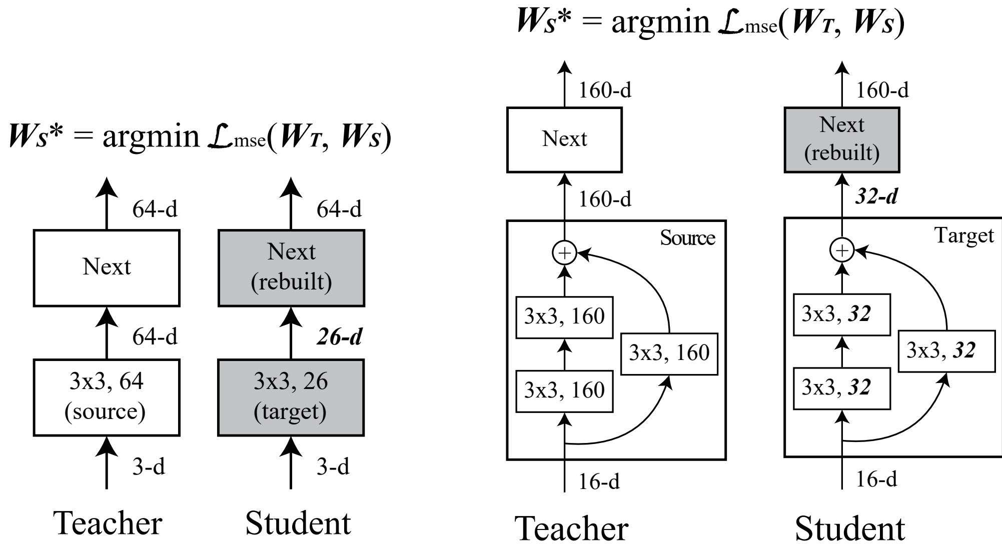


Figure 4.5: Examples of the VGG-16 and WRN-28-10 compression. Both example shows recasting of the first layer in each network.

meaningful feature, and a *redundant filter* denotes a filter that extracts a feature very similar to the one extracted by some other filter or a feature that can be obtained by combining features from other filters. To remove those filters, previous approaches use APoZ [44], sum of absolute values of a filter [19], or influence on next activations [39, 38] as the criteria, but redundant filters cannot be founded with those approaches. A possible approach is to find such redundant filters by checking the similarity between every pair of filters. However, it requires a huge amount of computations for similarity check and does not guarantee a good result. Instead, we recast a given source block into a smaller target block that has the same type as the source block. Figure 4.5 illustrates the proposed compression examples for the convolutional block and residual block. Then we train the target block and the next block to reconstruct the output activation of the next block with smaller number of filters. If the next block can reconstruct a similar output activation, the new target block can extract effective features for reconstruction. A convolution block can recast into another convolution block that has a smaller number of filters as shown in Figure 4.5 (*left*). Then we train both the

new convolution block and the next block to reconstruct the original activation map of the source next block. After training, we can obtain a more effective filter set without any similarity or effectiveness check criteria. We also apply the same method to residual network. In this case, we reduce the number of the convolutional filters inside the residual block. Especially, the reduction block that convolution with stride 2 has more layer, so we also reduce its channels.

4.2.4 Block Training

For the target block to work properly, it should be trained with the source block as the teacher. We can easily train the target block by approximating the output activations to those of the source block if both blocks have the same dimension of output activations. However, dimension mismatch happens in the block recasting especially when we reduce the number of channels for network size reduction. Table 4.1 shows the recasting cases that we handle in this paper. To avoid the dimension mismatch problem, when training a target block, we train the target block together with the next block by approximating the output activations of the next block as shown in Figure 4.6. The next block is rebuilt from the corresponding source block by reducing the filter size when the target block has a smaller number of channels. Both the target block and the next block are initialized randomly and trained to minimize the loss of mean-square error (MSE) between teacher's and student's activations given by,

$$\mathcal{L}_{mse}(W_T, W_S) = \frac{1}{N}\|A(x; W_T) - A(x; W_S)\|_2^2, \tag{4.1}$$

where A means the activation of the next block, and x is the input data. W_T and W_S indicate parameters of teacher network and student network, respectively. N denotes the size of an output activation of the next block.

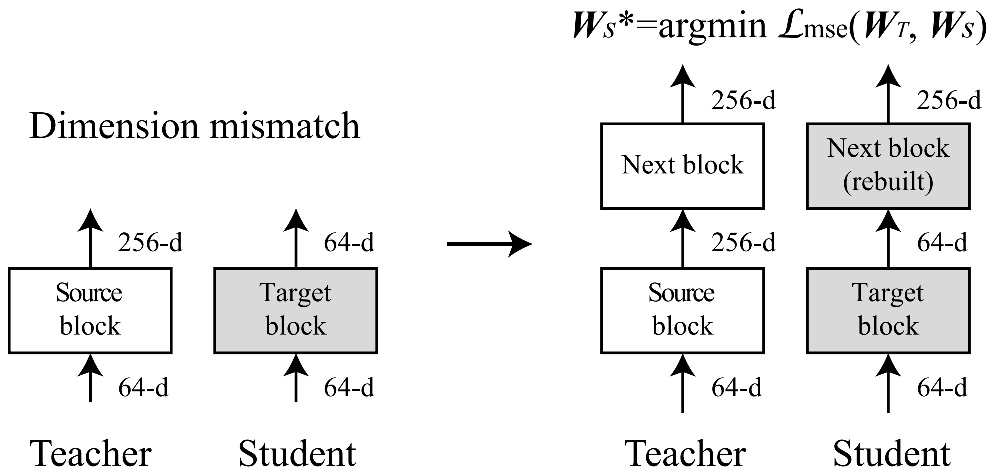


Figure 4.6: Dimension mismatch and proposed block training method. The dimension mismatch happens when the source block is recast into a smaller target block. The next block is used to match the dimension of output activation. After rebuilding the next block, both blocks are trained by minimizing $\mathcal{L}_{mse}(W_T, W_S)$.

Table 4.1: Candidates for the network recasting

Recasting Type	Source	Target	Dimension
Transformation	Dense	Basic	Preserved
	Dense	Convolution	Preserved
	Basic	Convolution	Preserved
	Bottleneck	Convolution	Reduced
Compression	Basic	Basic	Reduced
	Convolution	Convolution	Reduced

4.2.5 Sequential Recasting and Fine-tuning

To recast the entire network, we apply the block recasting method sequentially. Figure 4.7 shows an example of sequential recasting method. The type and dimension of the first (target) block of the student network are determined, and then the second block is rebuilt from the second block of the teacher network; if there is no dimension mismatch, the second block will be the same as that of the teacher network. The two blocks are initialized randomly and trained by minimizing $\mathcal{L}_{mse}(W_T, W_S)$. Now, the second block becomes the target. Thus, its type and dimension are determined, the third block is rebuilt, and both blocks are initialized randomly. To train the second and third blocks, we reuse the trained first block. The first block is already trained in the previous step, but it still has approximation errors. We can reduce the effect of its errors by training both the previous and current blocks. Therefore, three blocks are trained in the second step by minimizing $\mathcal{L}_{mse}(W_T, W_S)$. This process is continued for the following blocks until the last block is recast as a new block. We can select arbitrary blocks as candidates for recasting so that the student network can consist of multiple types of block.

For example, the student network can have both residual and dense blocks when only the first dense block is recast into a residual block. We call the network that has multiple types of block as *mixed-architecture network*, and Figure 4.8 shows an example of the mixed-architecture network. The mixed-architecture network can have advantages of both blocks. For example, by mixing dense blocks and residual blocks, we can obtain a mixed-architecture network that is faster than DenseNet and has fewer parameters than ResNet.

The block-by-block sequential recasting has two advantages. First, the functionality of each block is much simpler than that of the whole network. Thus, it is easier to approximate the functionality of each block. By approximating each of easier subfunctions, we can finally obtain the student network with smaller approximation error. Secondly, sequential recasting can alleviate the vanishing-gradient problem. When the

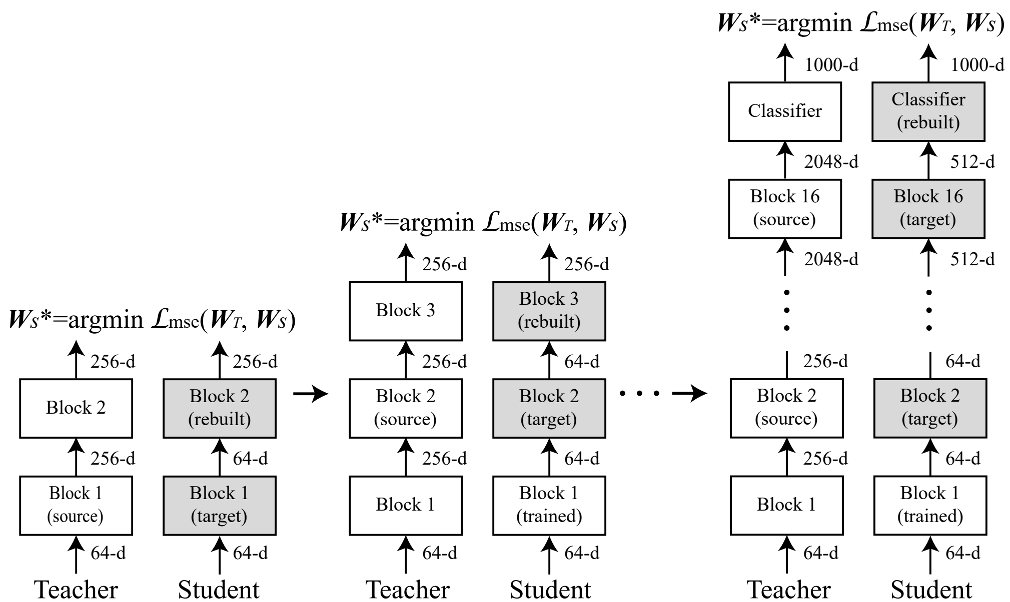


Figure 4.7: Example of sequential recasting for ResNet-50. All blocks are recast in this example. In each step, the target block and the next block (shaded blocks) are initialized randomly and trained by minimizing $\mathcal{L}_{mse}(W_T, W_S)$.

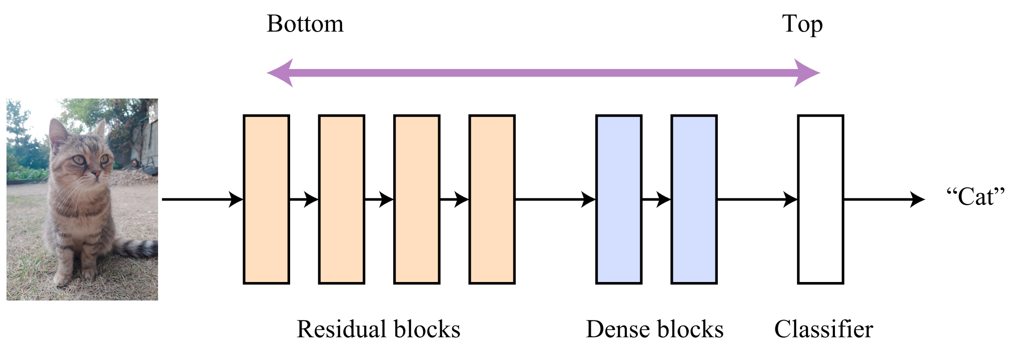


Figure 4.8: Example of the mixed-architecture network. It has both residual and dense block.

source block is recast as a convolution block, the student network cannot be trained well due to the gradient vanishing. However, sequential recasting has very short gradient paths from the output activation to the target block, so it can be trained well. Therefore, we can obtain the student network with higher accuracy using sequential recasting.

After finishing sequential recasting, we use the knowledge distillation approach to fine-tune the student network. There are approximation errors after sequential recasting, and we can reduce the effect of those errors by training the whole network. We train the student network with logits of the teacher network and ground truth. Thus, our knowledge distillation (KD) loss is defined by

$$\mathcal{L}_{kd}(W_T, W_S) = \mathcal{L}_{mse_logit}(W_T, W_S) + \mathcal{L}_{ce}(y_{true}, W_S), \tag{4.2}$$

where $\mathcal{L}_{mse_logit}$ is the MSE loss for the logits, and $\mathcal{L}_{ce}$ is the cross-entropy loss between the given label y_{true} and softmax output of the student network that is parameterized by W_s.

4.3 Experiments

We conducted several experiments for the network recasting. For the experiments, we used CIFAR and ILSVRC2012 dataset and four kinds of network architectures; ResNet [4], Wide ResNet (WRN) [5], DenseNet [6], and VGG-16 [108]. We adopted batch normalization [109] for all networks, because it was also effective for block-wise training. The network recasting was implemented on the *PyTorch* framework. We used the Xavier initializer [110] in all experiments. We used SGD with Nesterov momentum [111] to train the teacher network and used Adam optimizer [112] for the network recasting. In addition, we trained the student network with KD and back propagation from scratch using SGD with Nesterov momentum for the comparison.

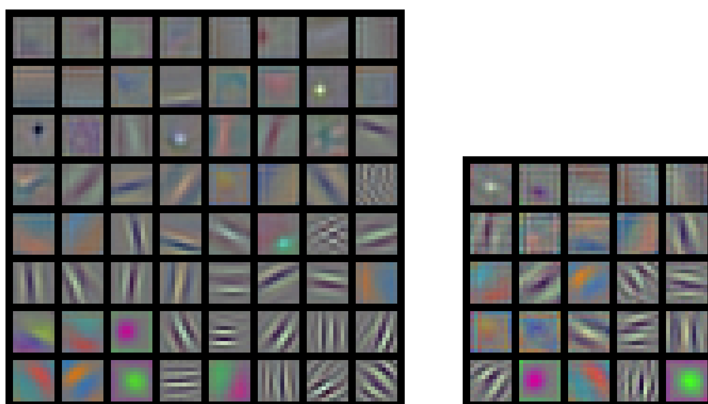


Figure 4.9: Visualization of filters in the first layer of AlexNet (*left*) and a student network (*right*). Redundant filters are removed after network recasting.

4.3.1 Visualization of Filter Reduction

The network recasting can be used for network compression; it can remove redundant filters as well as ineffectual filters. To show the filter reduction, we compressed only the first layer of AlexNet and visualized the filter set in Figure 4.9. The first layer of the original AlexNet had 64 filters, but we decreased the number to 25 in the student network. Then we trained the first block of the student network for eight epochs, and fine-tuned the entire student network; the learning rates for the recasting of the first block and the fine-tuning were 0.0005 and 0.0001, respectively. Every five epochs, the running rates were divided by 10. Figure 4.9 shows filters extracted from the first layers of the teacher and student networks. Filters of the teacher network consist of many ineffectual and redundant filters, but those are eliminated as shown in Figure 4.9. In addition, the student network achieves the top-1 error of 44.20% and the top-5 error of 21.54%. The top-1 and the top-5 errors increase by only 0.72% and 0.61%, respectively. Note that it is hard to remove many filters without accuracy loss because AlexNet has a relatively large (11×11) filters. The filter size is related to the dimension of filter vector, and many more filters are required to span the vector space as the filter size increases. As expected, we could remove many more filters on both VGG-16 and

ResNet, which have only 3×3 filters.

4.3.2 CIFAR

For CIFAR dataset, we used ResNet-56, ResNet-83, WRN-28-10, DenseNet-100, and VGG-16. Especially, ResNet-83 has the same number of blocks with ResNet-56, but consists of bottleneck blocks. In addition, we used a modified version of VGG16, which has only one hidden fully-connected layer with 512 neurons. Teacher networks were trained from scratch using back propagation. We used CIFAR-10 and 100 dataset with the standard data augmentation, which consists of four pixel zero-padding and random cropping, and horizontal flipping with 0.5 probability.

In CIFAR experiments, we recast all blocks of teacher networks, so there is no mixed-architecture result. We counted the number of parameters, multiplications, and activation loads for the convolution operation. Especially, we reported the activation load of a single image in Table 4.2. Table 4.2 shows the architecture transformation results. The network recasting achieved similar accuracy with the teacher network, and activation access is reduced significantly. It shows lower test error compared to other methods in all network architectures. When networks were recast into a plain convolutional network, the network recasting achieved much lower test error compared with both KD and back propagation. The sequential recasting can alleviate the vanishing-gradient problem, so its results outperformed the others.

We also compressed the VGG-16 and WRN-28-10 using the network recasting. In this experiment, source blocks were recast into $2.5\times$ and $5\times$ smaller blocks in VGG-16 and WRN-28-10, respectively. Table 4.3 shows compression results of both networks. The network recasting achieved the smallest accuracy loss compared with other methods. Especially, network recasting achieved 1.58% and 3.57% lower test error compared with KD and back propagation in VGG-16 compression on CIFAR-100.

The born again network (BAN) proposed by [84] also trains ResNet student using

Table 4.2: Error rates (%) of architecture transform results on CIFAR datasets (B/M: billion/million)

Method	Type	C10+	C100+	Params	Mults	Acts/image
			ResNet-56			
Baseline		7.02	30.89	0.85M (1.0×)	125.75M (1.0×)	0.56M(1.0×)
Recasting	Conv	**6.75**	**32.14**	0.41M (2.1×)	61.78M (2.0×)	0.27M(2.0×)
KD	Conv	9.43	33.22	0.41M (2.1×)	61.78M (2.0×)	0.27M(2.0×)
Backprop	Conv	10.61	37.85	0.41M (2.1×)	61.78M (2.0×)	0.27M(2.0×)
			ResNet-83			
Baseline		6.34	28.13	0.83M (1.0×)	125.09M (1.0×)	1.69M(1.0×)
Recasting	Conv	**6.90**	**31.04**	0.41M (2.0×)	61.78M (2.0×)	0.27M(6.2×)
KD	Conv	8.95	32.75	0.41M (2.0×)	61.78M (2.0×)	0.27M(6.2×)
Backprop	Conv	9.77	37.14	0.41M (2.0×)	61.78M (2.0×)	0.27M(6.2×)
			WRN-28-10			
Baseline		4.06	19.54	36.45M (1.0×)	5.24B (1.0×)	2.52M(1.0×)
Recasting	Conv	**4.11**	**19.74**	4.86M (7.5×)	1.17B (4.5×)	0.90M(2.8×)
KD	Conv	4.40	19.94	4.86M (7.5×)	1.17B (4.5×)	0.90M(2.8×)
Backprop	Conv	4.67	20.90	4.86M (7.5×)	1.17B (4.5×)	0.90M(2.8×)
			DenseNet-100			
Baseline		5.11	23.62	0.74M (1.0×)	0.29B (1.0×)	4.41M(1.0×)
Recasting	Basic	4.91	**22.39**	2.53M (0.3×)	0.77B (0.4×)	0.89M(4.9×)
KD	Basic	**4.71**	22.71	2.53M (0.3×)	0.77B (0.4×)	0.89M(4.9×)
Backprop	Basic	5.39	24.57	2.53M (0.3×)	0.77B (0.4×)	0.89M(4.9×)
Recasting	Conv	6.82	**25.60**	0.87M (0.9×)	0.19B (1.5×)	0.51M(8.6×)
KD	Conv	**6.75**	26.52	0.87M (0.9×)	0.19B (1.5×)	0.51M(8.6×)
Backprop	Conv	8.11	30.05	0.87M (0.9×)	0.19B (1.5×)	0.51M(8.6×)

Table 4.3: Error rates (%) of compression results on CIFAR datasets (B/M: billion/million)

Method	Type	C10+	C100+	Params	Mults	Acts/image
			VGG-16			
Baseline		6.85	28.80	14.71M(1.0×)	313.20M(1.0×)	0.31M(1.0×)
Recasting	Conv	**8.31**	**31.56**	2.36M(6.2×)	50.63M(6.2×)	0.13M(2.4×)
KD	Conv	9.24	33.14	2.36M(6.2×)	50.63M(6.2×)	0.13M(2.4×)
Backprop	Conv	8.71	35.13	2.36M(6.2×)	50.63M(6.2×)	0.13M(2.4×)
			WRN-28-10			
Baseline		4.06	19.54	36.45M(1.0×)	5.24B(1.0×)	2.52M(1.0×)
Recasting	Basic	**5.18**	**24.13**	1.46M(24.9×)	0.21B(24.5×)	0.52M(4.9×)
KD	Basic	5.48	25.28	1.46M(24.9×)	0.21B(24.5×)	0.52M(4.9×)
Backprop	Basic	5.39	25.78	1.46M(24.9×)	0.21B(24.5×)	0.52M(4.9×)

logits of DenseNet teacher. However, they proposed only switching DenseNet with ResNet, and the test error of BAN will be higher than that of network recasting because BAN only uses the KD method as shown in Table 4.2 and 4.3. We propose any to any architecture transformation, and deep student networks that have only convolution blocks can also be trained well by applying sequential recasting because it can alleviate the vanishing-gradient problem. In addition, we also propose mixed-architecture network, which can also be trained well by using the proposed network recasting.

4.3.3 ILSVRC2012

For ILSVRC2012 dataset, we used the pre-trained ResNet-50, DenseNet-121, and VGG-16 available from *torchvision* which is one of the *PyTorch* packages. These pre-trained networks were used as the teacher networks. We recast the blocks of ResNet-50 into convolution blocks, and the blocks of DenseNet-121 into basic blocks. In addition, we recast only parts of these networks to obtain mixed-architecture networks. In Table 4.4, *Recasting(C)* indicates that the student network only has convolution blocks, and *Recasting(C+R_{bt})* denotes that the student network has both convolution and bottleneck blocks. In the same way, *Recasting(R_{bs})* and *Recasting(R_{bs}+D)* denotes that

Table 4.4: Error rate (%) of network recasting results on ILSVRC2012 (B/M: billion/million, I/B: image/batch)

Method	Top1	Top5	Params	Mults	Acts/I	Time/I	Time/B
			ResNet-50				
Baseline	23.85	7.13	25.50M	4.09B	11.57M	6.16ms	107.17ms
Recasting(C)	30.74	10.39	10.29M	1.71B	2.53M	**2.12ms**	**37.21ms**
Recasting(C+R_{bt})	**25.00**	**7.71**	21.72M	2.40B	3.69M	3.79ms	49.97ms
KD(C+R_{bt})	27.00	8.30	21.72M	2.40B	3.69M	3.79ms	49.97ms
			DenseNet-121				
Baseline	25.57	8.03	7.89M	2.75B	16.52M	12.73ms	111.31ms
Recasting(R_{bs})	26.42	8.25	32.23M	8.15B	5.32M	**3.95ms**	**81.17ms**
Recasting(R_{bs}+D)	**24.87**	**7.59**	10.42M	5.72B	9.15M	9.40ms	88.94ms
KD(R_{bs}+D)	24.90	7.65	10.42M	5.72B	9.15M	9.40ms	88.94ms
			VGG-16				
Baseline	26.63	8.50	138.34M	15.47B	15.09M	6.17ms	200.47ms
Recasting(C_P)	**28.25**	**9.41**	81.93M	4.73B	8.27M	**3.45ms**	116.45ms
Recasting(C_A)	30.05	10.38	120.61M	3.12B	3.30M	3.61ms	**63.52ms**

the student networks consist of only basic blocks and both basic blocks and dense blocks, respectively. KD(C+R_{bt}) and KD(R_{bs}+D) have the same network architecture as Recasting(C+R_{bt}) and Recasting(R_{bs}+D) respectively, but those are trained with only KD method. For the VGG-16 compression, we used two criteria: higher parameter reduction (*Recasting(C_P)*) and higher activation reduction (*Recasting(C_A)*). In addition, we measured the actual inference time for all networks on an NVIDIA Titan X (Pascal) GPU, and batch sizes were set to 1 and 64.

We measured the training time for Recasiting(C+R_{bt}), KD(C+R_{bt}), Recasting(R_{bs} +D) and KD(R_{bs}+D) to compare the training time and accuracy. Recasting(C+R_{bt}) took 7.6 days, and KD(C+R_{bt}) took 6.3 days on a GPU. Compared to KD(C+R_{bt}), Recasting(C+R_{bt}) took 20% longer, but achieved 2.00%p and 0.59%p improvement in top-1 and top-5 accuracy, respectively. On the other hand, Recasting(R_{bs}+D) took 3.9 days, while KD(R_{bs}+D) took 8.9 days with similar accuracy. Those results show that network recasting can achieve higher accuracy with slightly longer training time for a deep network and shorter training time with similar accuracy for a shallow network.

Table 4.5: Comparison of error rate (%) with previous works on ILSVRC2012 (B/M: billion/million)

Method	Top1	Top5	Params	Mults	Acts/batch	Speed-up
ResNet-50						
Recasting(C+R$_{bt}$)	**25.00**	**7.71**	21.72M	2.40B	236.16M	**2.1**×
ThiNet-30 [39]	31.58	11.7	8.66M	1.10B	-	1.3×
AutoPruner ($r = 0.3$) [41]	27.47	8.89	-	1.32B	-	-
VGG-16						
Recasting(C_A)	**30.05**	**10.38**	120.61M	3.12B	220.61M	**3.2**×
ThiNet-Conv [39]	30.20	10.47	131.44M	4.79B	-	2.5×
RNP (3×) [40]	-	12.42	-	-	-	2.3×
Channel Pruning (3×) [38]	-	11.10	-	-	-	2.5×
AutoPruner ($r = 0.4$) [41]	31.57	11.57	-	4.09B	-	-

As shown in Table 4.4, the network recasting significantly reduced the inference time in all experiments. Recasting(C) and Recasting(R$_{bs}$) achieved 2.9× and 3.2× inference time reduction for a single image compared with original ResNet-50 and DenseNet- 121, respectively. Moreover, mixed-architecture networks also achieved significant inference time reduction with smaller accuracy loss. For the batch processing, Recasting(C+R$_{bt}$) achieved 2.1× time reduction with 0.58% top-5 accuracy loss compared to Baseline, and Recasting(R$_{bs}$+D) achieved 1.3× time reduction even with 0.44% higher top-5 accuracy. In particular, Recasting(R$_{bs}$+D) achieved similar accuracy and inference time with 3.1× fewer parameters compared to Recasting(R$_{bs}$). In VGG-16 compression, Recasting(C_P) and Recasting(C_A) achieved 1.7× parameter reduction and 4.6× activation reduction with 0.91% and 2.05% top-5 accuracy loss, respectively. Recasting(C_A) achieved 3.2× inference time reduction compared to the baseline.

We compared our results with several previous approaches [39, 38, 40, 41]. For the comparison, we used batch inference time because previous approaches have reported inference time only for the batch processing. Table 4.5 shows that the network recasting achieved much higher inference time reduction. In ResNet-50, Recasting(C+R$_{bt}$) achieved lower error rate and much higher actual speedup compared with ThiNet [39].

ThiNet only reduced filters and multiplications in 3×3 convolution of bottleneck blocks, so it cannot accelerate the inference effectively because activation load is still large. However, the network recasting can reduce the activation load effectively, so it achieved $2.1\times$ actual speedup with smaller accuracy loss. AutoPruner does not mention actual-speedup, but we can guess that our network recasting result is much faster than their AutoPruner result because they cannot remove the 1×1 convolution [41] . For the VGG-16 compression, the network recasting also achieves much higher speedup with lower error rate compared to previous approaches. It also achieves higher parameter and multiplication reduction with similar accuracy compared to others.

4.4 Summary

In this paper, we proposed network recasting as a universal method for network architecture transformation. This method can accelerate network inference by transforming the network (teacher) to a more efficient one (student). We could recast residual and dense blocks into convolution and residual blocks, respectively, to achieve much higher actual speedup at small accuracy loss. By recasting blocks sequentially, the student network can be trained well even though there is no shortcut or dense connection. In addition, our method can recast arbitrary blocks, thereby producing a mixed-architecture network. The mixed-architecture networks produced as such achieved $2.1\times$ inference time with 0.58% top-5 accuracy loss compared to original ResNet-50, and also achieved $1.3\times$ inference time reduction with 0.44% higher top-5 accuracy on DenseNet-121 recasting. We also applied the network recasting for the purpose of compression and achieved higher compression ratio and speedup compared to previous approaches. Our method can be applied to various kinds of network architecture to transform it into various kinds of target network architecture.

Chapter 5

Fine-Grained Neural Architecture Search

5.1 Motivation

5.1.1 Search Space Reduction Versus Diversity

To find neural network architecture automatically, several NAS algorithms are proposed. In NAS research, the most critical problem is that search space is tragically large. For example, when we want to find 10 layer network architecture and each layer has 8 candidates, the search space becomes more than 1 billion (8^{10}). To solve those problems, the cell-level design method is introduced [87, 88]. The cell is the basic building block of network architecture, and it is repeated several times. It can dramatically reduce the search space because we only have to find one cell structure. Thanks to this idea, several kinds of network search can be used for the NAS. Liu et al. [90] propose AmoebaNet, and they find two kinds of cells (normal and reduction) using the evolutionary algorithm. The gradient-based network search is also proposed [22], and it finds not only network architecture but also weight parameters. Especially, the gradient-based network search only takes several GPU days to find network architecture, so many researches are proposed to improve the performance of the gradient-based search.

The cell-based repeating structure cannot reflect positional information. According

to the position, activation size and the number of filters are changed, so block characteristic is totally different. The bottom layers (close to the input data) have large input activation and small filters, but the top layers (close to logits) has small input activation and large filters. In addition, the filter also extracts different abstraction level of features according to the position. For the bottom layers, filter extract low-level features (edges, colors, spiral, etc), and the filter of top layers extract high-level features (flamingo, pelican school bus, etc) [18]. For this reason, we have to free from cell level search to build more efficient network architecture.

To reduce the search space preserving the diversity of each block, a simpler structure is used. ProxylessNAS [23] and FBNet [24] use the inverted residual block [113], and vary the kernel size and expansion ratio. The inverted residual block is much simpler than the DARTS cell, so they achieve a layer-wise architecture search. Moreover, the final architecture consists of several inverted residual blocks, and those are cascaded. Thanks to this characteristic, it is easy to measure and estimate the latency of the final network. They measure the latency of each operation and use the weighted average value of the measured latency for the latency estimation. However, a more complex search space is needed to find a much better architecture.

5.1.2 Hardware-Aware Optimization

To design efficient network architectures, many architectures are proposed. ResNet is adapt the 1×1 convolutions to reduce FLOPs [4], and DenseNet achieves further reduction by using dense connections [6]. In addition, several mobile target architectures are also proposed for the FLOPs reduction [114, 113, 115]. However, FLOP count cannot reflect actual latency as we mentioned in Section 4.1. Yang et al. [116] propose the energy-aware pruning method, and they focus on not only weights and MACs but also feature map. Memory access requires much higher energy compared with arithmetic operations [57], and it is related to both the number of weights and feature map size. By reducing the feature map size and layer, the pruned network has

much higher energy efficiency. Chen et al. [117] also proposed another energy-aware pruning method, and they use layer-wise pruning. The energy consumption of each layer is totally different, so they apply a different pruning ratios for each layer. For example, many previous works focus on the weight compression ratio, so it is highly biased for the fully-connected layer. However, convolutional layers consume most of the energy, and it is also not evenly distributed. Those researches show that efficient network architecture cannot be found with only one component. Therefore, a more sophisticated estimation method is needed for efficient network architecture.

5.2 InheritedNAS

We propose the novel fine-grained differentiable neural architecture search (InheritedNAS), and it is a secondary searching method from the coarse-grained searched architecture. Without any search space reduction, it is impossible to search the network that has a layer-wise search space, so we reduce search space by using cell-based architecture. First, we find cell-based network architecture with previous approaches such as DARTS, and then we find new network architecture with a layer-wise search space by using hints from the searched cell-based architecture. Figure 5.1 illustrates the overall process of InheritedNAS. In this paper, we call the cell-based architecture as *the coarse-grained architecture*, and layer-wise searched architecture as *the fine-grained architecture*.

5.2.1 Stage Independent Search

The convergence of architecture search depends on the complexity of the search space. If the given search space is too large, the gradient-based NAS algorithm is diverged, so it is very important to design search space. To search the fine-grained architecture, we divide the network according to the activation size (stage) to reduce the complexity of the search space. We build and train searched cell-based architecture, and we build

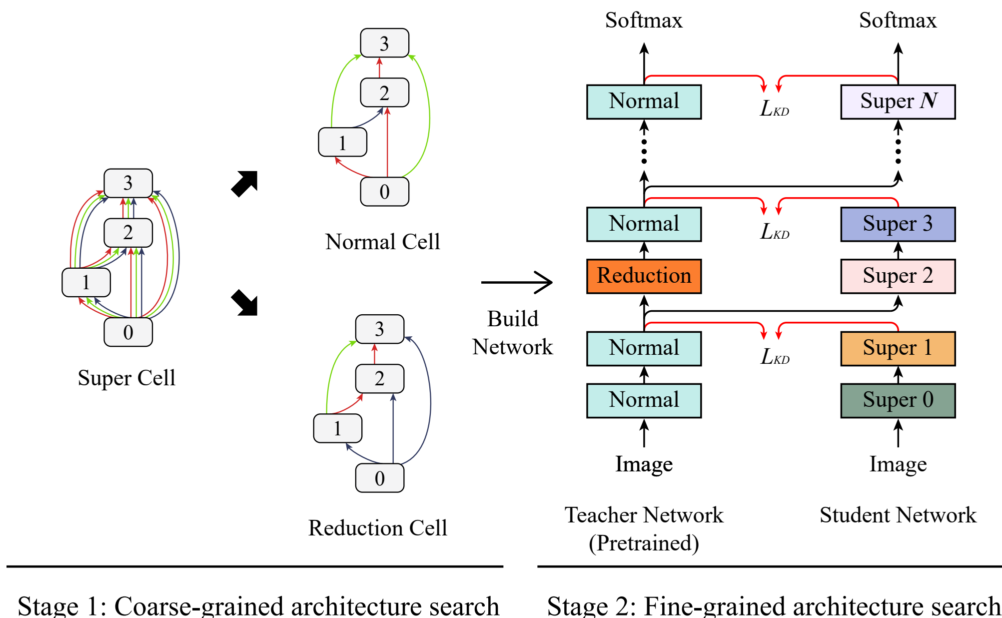


Figure 5.1: The overall process of InheritedNAS. First, coarse-grain architecture is searched. After then the fine-grain architecture is searched with pretrained coarse-grain architecture. To reduce the search space, we divide and train the network using the knowledge distillation, and the teacher network gives layer-wise/stage-wise hints to the student network for the fine-grain architecture search. Each super cell has its own architecture parameters, so each block has intrinsic architectures after search.

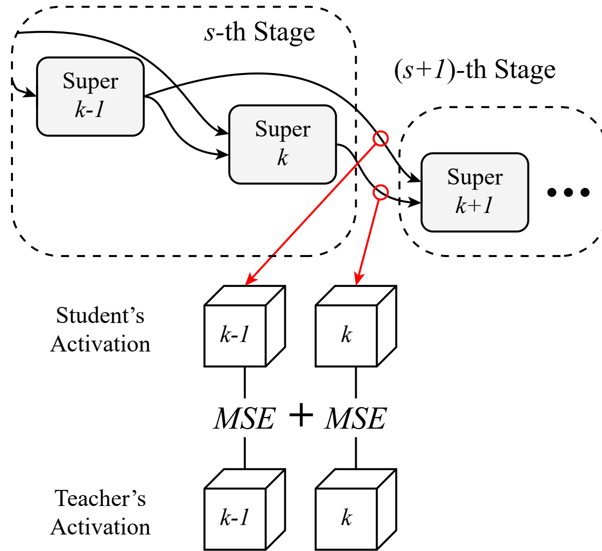


Figure 5.2: The two-point matching distillation. This method can break the dependency from s-th to (s+1)-th stage, so each stage can be trained independently.

the fine-grained architecture with the super cell rather than the searched normal or reduction cell. After then, we find each network architectures of the divided stage as shown in Figure 5.1. Each divided stage has its own search space, and search space is independent from each other because those are trained independently. For this reason, the convergence of the NAS is related *the maximum complexity of given search spaces* rather than that of the entire search space. For instance, there are eight super cells in the network, and it is divided into three stages ($[2, 2, 4]$). The maximum complexity of the search is determined by the last stage that has four super cells. The other stages have two super cell, and it can be searched successfully when the NAS for the last stage is converged. Therefore, we only have to consider the maximum complexity of search spaces.

In the fine-grained architecture search stage, the coarse-grained architecture is considered as a teacher network. The teacher network gives the input data and targets (intermediate activations) to the fine-grained architecture (student network) as shown in

Figure 5.1. Looking into the super cell architecture, it requires two input activations from previous and before previous cell, and it gives an output activation to next and after next cell. To train each stage independently, this input and output dependency has to be modified. For this reason, we give two intermediate activations of the teacher network to the student network as the inputs and train each stage by minimizing the MSE loss for two output activations as shown in Figure 5.2. We call this training method as *the two-point matching distillation*. For the more stable training, we match the size of two output activation, because the scale of MSE loss can be changed according to the activation size. The sizes of two output activation are the same when the network is divided according to the stage. If another division method is used, the activation sizes have to be concerned carefully.

5.2.2 Operation Pruning

To determine the final architecture, the previous work retains the top-k operations among all non-zero operation for each node [22]. For instance, each node chooses top-2 strongest operations, so there are eight edges per cell in the experiments of previous work. However, this method eliminates the chance to find more diverse network architecture, and zero operation (disconnection) never appears. For further optimization, we propose *the operation pruning*, and it removes less important operation similar to previous parameter pruning. To the best of our knowledge, this is the first work pruning operation rather than weights or filters. After training architecture parameters is finished, the most probable operation is selected for each edge, and network architecture is determined. And then, we train the weight parameters from scratch and also train *the connectivity parameters*. The connectivity parameter is introduced to determine the connection of each edge, and it is converted to the connection probability. Figure 5.3 illustrates the forward propagation of the connectivity parameter. To relax the discrete characteristic, we use the connection probability, and it is multiplied to the output of the operation. The connection probability is calculated by using the sigmoid

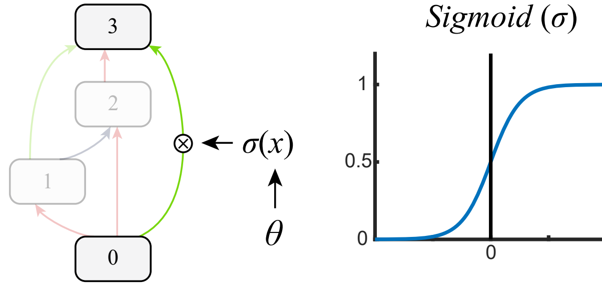


Figure 5.3: Forward propagation for the connectivity parameters θ. The probability of connection is calculated with the sigmoid function, and it works as the scaling factor of each operation.

function that is used to convert parameters to probabilities. After training, we remove the less important filters that have a smaller connection probabilities. To give more diversity, we apply the block-wise pruning that considers all candidates in the block. In this work, we remove the six operations in each block according to the connection probabilities. Thanks to the block-wise pruning, various architectures can be found. For example, some node does not have any input edges, so it always generates zero activation. The previous work cannot remove the nodes, but the proposed method can remove them. After connectivity of each edge is determined, we remove the operations according to the connectivity parameters, and then also remove *the redundant operations*, which has zeros as its inputs because every input edge is disconnected. In this case, it never affects prediction, so it can be removed without any accuracy loss.

5.2.3 Entire Search Procedure

The InheritedNAS consists of two training stages as we mentioned in Section 5.2. The first training stage is the coarse-grained architecture search, and it is the same with previous work [22]. The second training stage is the fine-grained architecture search, and it consists of two sub-stages. We train the architecture parameter to find proper operations for each edge and then remove unimportant operations. Algorithm 2 shows the

Algorithm 2 InheritedNAS

Create super network with weights w_c and *coarse-grained* architecture parameter α_c
while *not converged* **do**
  Train weight parameter w_c by descending the $\nabla_{w_c}\mathcal{L}_{CE}(w_c, \alpha_c)$
  Train architecture parameter α_c by descending the $\nabla_{\alpha_c}\mathcal{L}_{CE}(w_c, \alpha_c)$
Build coarse-grained architecture N_c based on the trained α_c

Train N_c by descending the $\nabla_{w_c}\mathcal{L}_{CE}(w_c)$

Create super network with weights w_f and *fine-grained* architecture parameter α_f
while *not converged* **do**
  Train weight parameter w_f by descending the $\nabla_{w_f}\mathcal{L}_{PKD}(w_f, \alpha_f)$
  Train architecture parameter α_f by descending the $\nabla_{\alpha_f}\mathcal{L}_{PKD}(w_f, \alpha_f)$
Build fine-grained architecture N_f based on the trained α_f
Initialize w_f and add connectivity parameter θ
while *not converged* **do**
  Train weight parameter w_f by descending the $\nabla_{w_f}\mathcal{L}_{KD}(w_f, \theta)$
  Train connectivity parameter θ by descending the $\nabla_{\theta}\mathcal{L}_{KD}(w_f, \theta)$
Remove operations based on the trained θ and redundant operations

detail of InheritedNAS. Weight and architecture parameters are separately trained for stable training. To search the coarse-grained architecture, conventional cross-entropy loss ($\mathcal{L}_{CE}$) is used. And, we use parallel knowledge distillation loss ($\mathcal{L}_{PKD}$) for the stage independent search. The stage-wise training is applied with two point matching distillation ($\mathcal{L}_{TPD}$), so our parallel knowledge distillation loss is defined by,

$$\mathcal{L}_{PKD}(w_f, \alpha_f) = \sum_{i=1}^{s} \mathcal{L}_{TPD}(\mathcal{S}_c^i, \mathcal{S}_f^i) \tag{5.1}$$

$$= \sum_{i=1}^{s} \sum\nolimits_{(A_c, A_f)\in(\mathcal{S}_c^i, \mathcal{S}_f^i)} \frac{1}{N} \|A_c - A_f\|_2^2, \tag{5.2}$$

where w_f indicates the the weight and α_f denotes architecture parameter of the fine-grained architecture. s is the number of stages, and $\mathcal{S}_c^i$, $\mathcal{S}_f^i$ are outputs of the i-th stage for coarse-grained (c) and fine-grained (f) architecture, respectively. A_c, A_f also means the activation of coarse-grained and fine-grained architecture, and N is

the number of elements in the activation. After stage-wise training, the fine-grained architecture is built based on the trained architecture parameters, and it also contains a new connectivity parameter θ. The next sub-stage is operation pruning, the fine-grained architecture is trained by minimizing the conventional knowledge distillation loss ($\mathcal{L}_{KD}$), and it defines as:

$$\mathcal{L}_{KD}(w_f, \theta) = \mathcal{L}_{MSE}(o(X; w_c), o(X; w_f, \theta)) + \mathcal{L}_{CE}(Y_{true}, O(X; w_f, \theta)), \quad (5.3)$$

where $\mathcal{L}_{MSE}$ is the mean square error loss. o is the output logits, and X and $Y_t rue$ denote the input data and its target, respectively. After operation pruning, the architecture search is finished.

5.3 Hardware-aware Penalty Design

The fine-grained architecture search gives further optimization chance in terms of inference accuracy, time, and energy consumption. As we mentioned Section 5.1, the operation can have different characteristics according to its position, so choosing proper operation is very important. For this reason, we use the hardware-aware penalty to retain proper operations, and it is applied during both the stage independent search and the operation pruning. In this work, we focus on the FLOPs and memory access to obtain the hardware friendly architecture.

FLOPs Penalty

Applying penalty to total FLOPs is the most easiest way to improve the performance of the hardware. Inference time is naturally reduced by reducing the number of multiplication. To obtain the FLOPS, we only consider the weighted operation; convolution and linear (fully-connected layer). Batch normalization (BN) also has an arithmetic operation, but it can be merged to weighted operation because it is frozen in inference [103]. Each edge has several operations during the architecture search, so exact FLOPs

value is not determined. For this reason, we use the expected FLOPs values, and it is calculated the weight average of operations.

$$E[\text{FLOPs}] = \sum_i p_i \times F_f(\text{op}_i). \tag{5.4}$$

p_i is the probability of the i-th architecture parameter, and it is considered as the weigh value for i-th operation (op_i). F_f indicates the flops calculation function. This method is the same as the latency prediction model in ProxylessNAS [23].

Memory Access Penalty

As the complexity of network architecture increases, FLOPs based optimization research show very small inference time reduction [39]. Previous work shows that the number of multiplication is not directly related to inference time [118]. Due to FLOPs cannot reflect hardware performance properly [118], we add an additional penalty term to reflect the effect of memory access. As we mentioned in Section 4.1, memory access is very important for the actual hardware. There are also many researches that report memory access takes most of the energy and computation time both inference and training [58, 119, 120]. For this reason, we also propose a new penalty that considers memory access to find hardware friendly network architecture. Basically, weight parameter and activation cause memory access, so those are used to estimate the total memory access. In CPU and GPU, operations run layer-by-layer, and both activation function and batch normalization are also considered as a single layer. However, activation function and batch normalization can be merged to the convolutional layer in the inference accelerator as we mentioned before. To support both hardware characteristic, we design the two expectation mode; *individual* and *unified*. Similar to expected FLOPs, the weighted average is used for the expected memory access.

$$E[\text{Memory}] = \sum_i p_i \times F_m(\text{op}_i). \tag{5.5}$$

Mixed Penalty

We also propose a mixed penalty that consists of both FLOPs and memory access. The mixed penalty is defined as:

$$E[\text{Mix}] = \alpha \cdot E[\text{FLOPs}] + \beta \cdot E[\text{Memory}], \tag{5.6}$$

where α and β are hyperparameter, and it decide the optimization direction. Each hardware has a different characteristic, so For example, CPU has few arithmetic logic units (ALUs), so FLOPs reduction can be a better choice than memory access reduction. Conversely, GPU has many ALUs, and memory access reduction can be more important when the average utilization of ALUs is low. Therefore, hardware-specific network architecture can be searched by adjusting α and β.

5.4 Experiments

We conducted several experiments to see the convergence and performance of the InheritedNAS, and applied several kinds of hardware-aware penalty to check the performance on the hardware. The InheritedNAS was implemented on the *PyTorch* framework. For the experiments, we used the CIFAR-10 and CIFAR-100 dataset, and cutout [121] and the standard data augmentation, which consists of four pixel zero-padding and random cropping, and horizontal flipping with 0.5 probability. We initialize the network with Xavier initializer [110]. we use the SGD with Nesterov momentum [111] for the weight parameter training, and use the Adam optimizer [112] for the architecture and connectivity parameter training. We used the network architecture that was found in DARTS [22] as the coarse-grained architecture. For the architecture search, we use the operation set; 3×3 and 5×5 depthwise separable convolution, 3×3 and 5×5 dilated separable convolution, 3×3 max pooling, 3×3 average pooling, identity, and *zero*. This operation set is the same as that of the previous work [22]. However,

in this set, memory-intensive operations also have huge amount of computations. The depthwise separable convolutions have much larger FLOPs and memory access compared with the dilated separable convolutions. Actually, FLOPs penalty is a more harsh constraint than the memory penalty. Therefore, FLOPs penalty shows the best result on every item; FLOPs, parameters, activations, and inference time. For this reason, we introduce new operation set; 3×3 and 5×5 depthwise separable convolution, 3×3 and 5×5 standard convolution, 3×3 max pooling, 3×3 average pooling, identity, and *zero*. The depthwise separable convolution has smaller FLOPs compared with standard convolution, but it has much larger memory access due to the activations. We call the former set as OS1, and the latter set is OS2.

5.4.1 Fine-Grained Architecture Search

For the fine-grained architecture search, we used the network that consists of eight blocks. The coarse-grained architecture is consists of normal and reduction cell that were found in DARTS, and the fine-grained architecture search has eight super cell. We trained the fine-grained architecture for 200 epochs in both stage independent search and operation pruning. The learning rates of the SGD were starting 0.1, and it was divided by 5 at the 60, 120, and 150 epoch. For the architecture parameter, the learning rates were starting at 0.001, and there was no learning rate drop. Three kinds of penalties (None, FLOPs, and Memory) are used to check the maximum accuracy of the searched network and the effect of the penalty.

Table 5.1 shows the our experimental results. In OS1 The fine-grained architecture shows the 92.31% accuracy, and it achieved the 0.8%p higher accuracy compared with the coarse-grained architecture. The performance improvement comes from block diversity as we mentioned before. By applying the penalty term, accuracy slightly decreases but it still comparable. In addition, hardware characteristics are much more improved compared with DART and non penalized architecture. For the FLOPS penalty result shows the smallest FLOPs, parameter, and activation.

Table 5.1: Experimental results of InheritedNAS

Model	Penalty	Accuracy	FLOPs	Parameter	Activation
DARTS[22]	-	91.56	42.95 M	0.30 M	1.63 M
Ours (OS1)	-	**92.31**	49.89 M	0.38 M	1.46 M
	FLOPs	91.82	**33.43** M	**0.21** M	**1.03** M
	Memory	91.13	38.38 M	0.25 M	1.07 M
Ours (OS2)	-	**92.67**	257.65 M	2.33 M	0.63 M
	FLOPs	92.02	**34.10** M	**0.18** M	0.86 M
	Memory	92.03	225.46 M	1.89 M	**0.60** M

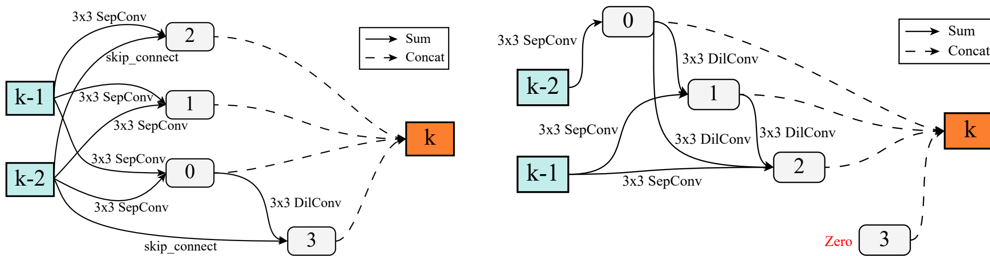


Figure 5.4: Comparison of searched architecture. (*left*) The normal cell of DARTS. (*right*) The first block of our searched network (OS1).

In OS2, all searched network shows a higher accuracy compared with previous work result. Without the penalty, the number of FLOPs is much higher because standard convolution is much better for the accuracy compared with separable convolution. For this reason, it achieved the highest accuracy, and its accuracy is the 1.1% higher than previous work. The other penalized networks also achieved a higher accuracy than previous work and OS1.

Figure 5.4 illustrates the searched architecture of DARTS and our searched network. DARTS uses the node-wise operation selection, so it has a regular pattern for each edge. However, our method has more diversity for each node, and several nodes

Table 5.2: Architecture search results through the hardware penalties

α, β	Accuracy	FLOPs	Parameter	Activation
Operation Set: OS1				
$1.00, 0.00$	91.82	**33.43** M	**0.21** M	**1.03** M
$0.75, 0.25$	92.05	39.44 M	0.25 M	1.21 M
$0.50, 0.50$	91.89	33.13 M	0.21 M	0.97 M
$0.25, 0.75$	**92.09**	39.21 M	0.25 M	1.20 M
$0.00, 1.00$	91.13	38.38 M	0.25 M	1.07 M
Operation Set: OS2				
$1.00, 0.00$	92.02	**34.10** M	**0.18** M	0.86 M
$0.75, 0.25$	**92.21**	68.82 M	0.38 M	0.74 M
$0.50, 0.50$	91.99	91.12 M	0.52 M	0.72 M
$0.25, 0.75$	92.08	143.00 M	0.90 M	0.77 M
$0.00, 1.00$	92.03	225.46 M	1.89 M	**0.60** M

can be discarded. Figure 5.4 (*right*) shows the the first block of our searched network, and the third node has no input operations. In this case, there is no operation for the third node, and it only generates the activation that consists of only zero value. We also can remove those node to further optimization, so it is our future work.

5.4.2 Penalty Analysis

We conducted several experiments to check the diversity of searched architectures according to the penalty. In this experiment, FLOPs penalty, memory access penalty, and three kinds of mixed penalties are used. We divided the penalty range to the 0.25 granularity, so there are three mixed penalties. Each expected penalties are normalized to FLOPs and memory access of the coarse-grained architecture, and it makes training more stable.

Table 5.3: Latency on the CPU and GPU

α, β	CPU image	CPU batch	GPU image	GPU batch
Operation Set: OS1				
$1.00, 0.00$	**31.26** ms	**1157.91** ms	**13.80** ms	**14.63** ms
$0.75, 0.25$	34.90 ms	1273.25 ms	15.44 ms	16.27 ms
$0.50, 0.50$	31.57 ms	1183.96 ms	13.54 ms	14.43 ms
$0.25, 0.75$	35.83 ms	1353.46 ms	15.31 ms	16.31 ms
$0.00, 1.00$	35.29 ms	1394.95 ms	14.88 ms	15.86 ms
Operation Set: OS2				
$1.00, 0.00$	17.09 ms	397.76 ms	13.48 ms	14.17 ms
$0.75, 0.25$	17.16 ms	**341.10** ms	14.45 ms	14.70 ms
$0.50, 0.50$	16.72 ms	353.26 ms	13.70 ms	14.19 ms
$0.25, 0.75$	19.12 ms	393.10 ms	15.66 ms	16.06 ms
$0.00, 1.00$	**16.63** ms	393.19 ms	**13.28** ms	**13.43** ms

Table 5.2 shows the InheritedNAS results for the given penalties. α and β mean the coefficient for the memory and FLOPs in 5.6. In OS1, every searched architecture show the very similar number of FLOPs and memory footprint for all penalties. The number of FLOPs is related to the parameter and activation, so the network has the smallest FLOPs and memory when α and β are 1 and 0 respectively. In OS2 the searched architectures show much more differences according to the penalty. When α becomes larger, FLOPs decreases, and activation increases. The number of parameters is directly related to the FLOPs in this operation set. For this reason, hardware penalty shows much obvious characteristics in OS2.

In addition, we measured the latency of the searched network on the Intel Xeon

E5-1680 v4 CPU and NVIDIA Titan X (Pascal) GPU, and batch size was set to 1 (image) and 128 (batch). Table 5.3 shows the latency of each architecture. In OS1, the searched architecture with FLOPs penalty has the smallest FLOPs, memory access, so it shows the best result on both CPU and GPU. FLOPs penalty still shows the best result in the batch processing.

However, in OS2, inference time trends of single image and batch processing are totally different. For the single image processing, memory access is more important because the arithmetic operation is too small to hide memory access. In batch processing, CPU and GPU show a different result. FLOPs reduction shows a better result on the CPU, and this result comes from parallelism. CPU has much smaller parallelism, so it requires much more time for the arithmetic operations. Memory access time can be hidden, so FLOPs much affect the inference time. Conversely, GPU has much more parallelism, so memory access time still requires much more time. Therefore, the penalty has to be designed according to the hardware parallelism.

5.5 Summary

In this work, we proposed the InheritedNAS, which is the fine-grained architecture search method. We propose the stage independent search method that can reduce the search space effectively. To break the dependency of each stage, two-point matching distillation method is proposed, and it helps to stabilize the training dynamic. By using those methods, we can obtain the fine-grained architecture, which has higher accuracy compared with previous coarse-grained architecture. Moreover, we also proposed the hardware-aware penalty to find efficient network architecture for the target hardware. Experimental results show that the inference time of searched architecture can be reduced effectively with the proposed penalty.

Chapter 6

Conclusion

In this dissertation, we proposed the designing technique for both neural network accelerator and network architecture. Conventional designs have several limitation and inefficiency for the inference. The neural network has a huge amount of arithmetic operations, so it is very inefficient on conventional hardware such as CPU and GPU. And, previous neural network compression methods do not concern the actual hardware characteristic, so its inference is still inefficient after the compression. Moreover, previous neural architecture search algorithms find the network that has a regular pattern, so the positional characteristic of each layer is not a concern. The proposed techniques alleviate those limitations and help to achieve efficient inference.

First, we proposed the neural network accelerator based on the stochastic computing. The stochastic computing has very small multiplication hardware, but it has computation errors. This characteristic prevents to support of the deeper network and complex dataset because its multiplication error is amplified according to the number of layers. To solve this problem, the unipolar encoding is used, and it can reduce the multiplication error effectively. In addition, we proposed the stochastic ReLU function and new stochastic max function, which occupy about half of the previous max function area. The weight modulation method is proposed to increase the SNR of SC hardware, and it can improve the inference accuracy dramatically. Moreover, we also

proposed the random number generator sharing technique to reduce the area overhead. Our experiments show that the accuracy of the SC network becomes close to that of the floating-point network on MNIST and CIFAR-10 datasets, and the proposed hardware outperforms the previous SC-based hardware in terms of the accuracy, prediction time, area, and energy consumption.

Second, we proposed the network recasting that enables the network architecture transformation. Several previous works achieve significant network compression, but its actual speedup is much smaller than the compression ratio. This problem is caused by the network architecture, so it cannot be solved if architecture is maintained. Our proposed method changes the network architecture itself, so further improvement can be achieved. The proposed method can accelerate inference by transforming the network into a more efficient one. To train the efficient network, we proposed the block training and sequential recasting method. In addition, our method can recast the arbitrary blocks, so a mixed-architecture can be found. The mixed-architecture networks produced as such achieved $2.1\times$ inference time with 0.58% top-5 accuracy loss compared to original ResNet-50, and also achieved $1.3\times$ inference time reduction with 0.44% higher top-5 accuracy on DenseNet-121 recasting. The proposed method also can be used for the network compression, and we also achieved a higher compression ratio and speedup compared to previous compression approaches.

Third, we propose InheritedNAS, which is the fine-grained network architecture search method. To automate the network architecture design, the neural architecture search method is proposed. However, the previous NAS method only finds the normal and reduction cell, so its diversity is significantly small. To enlarge the search space preserving the convergence, we proposed the fine-grained architecture search method. The proposed method gives a chance to find network architecture on the more large search space, and it also preserves the convergence. The stage independent search can reduce the search space effectively, and it also helps to stabilize the training dynamics. To break the dependency of each stage, we also proposed the two-point matching dis-

tillation method. In addition, we also propose the hardware-aware penalty, and it helps to choose the operation for efficient inference on the hardware. Our experiments show that our result gives a further optimization chance, and latency is effectively reduced.

Bibliography

[1] M. Lin, Q. Chen, and S. Yan, "Network in network," in *Proceedings of the International Conference for Learning Representations (ICLR)*, Apr. 2014.

[2] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015, pp. 1–9.

[3] G. Larsson, M. Maire, and G. Shakhnarovich, "Fractalnet: Ultra-deep neural networks without residuals," in *Proceedings of the International Conference for Learning Representations (ICLR)*, Apr. 2017.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778.

[5] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *Proceedings of the British Machine Vision Conference (BMVC)*, Sep. 2016, pp. 87.1–87.12.

[6] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 4700–4708.

[7] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*. MIT press Cambridge, 2016, vol. 1.

[8] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, Dec. 2015, pp. 1135–1143.

[9] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient inference engine on compressed deep neural network," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2016, pp. 243–254.

[10] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2016, pp. 14–26.

[11] K. Kim, J. Kim, J. Yu, J. Seo, J. Lee, and K. Choi, "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks," in *Proceedings of the Annual Design Automation Conference (DAC)*, Jun. 2016, p. 124.

[12] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Transactions on Embedded computing systems (TECS)*, vol. 12, no. 2, p. 92, May 2013.

[13] Z. Li, A. Ren, J. Li, Q. Qiu, Y. Wang, and B. Yuan, "DSCNN: hardware-oriented optimization for stochastic computing based deep convolutional neural networks," in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, Oct. 2016, pp. 678–681.

[14] A. Ren, J. Li, Z. Li, C. Ding, X. Qian, Q. Qiu, B. Yuan, and Y. Wang, "SC-DCNN: highly-scalable deep convolutional neural network using stochastic computing," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2017.

[15] Z. Li, A. Ren, J. Li, Q. Qiu, B. Yuan, J. Draper, and Y. Wang, "Structural design optimization for deep convolutional neural networks using stochastic computing," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2017, pp. 250–253.

[16] J. Li, Z. Yuan, Z. Li, C. Ding, A. Ren, Q. Qiu, J. Draper, and Y. Wang, "Hardware-driven nonlinear activation for stochastic computing based deep convolutional neural networks," in *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, May 2017.

[17] V. T. Lee, A. Alaghi, J. P. Hayes, V. Sathe, and L. Ceze, "Energy-efficient hybrid stochastic-binary neural networks for near-sensor computing," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2017, pp. 13–18.

[18] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," in *Proceedings of the International Conference on Machine Learning Workshop (ICML Workshop)*, Jul. 2015.

[19] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2016.

[20] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2017.

[21] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the Thirty-Third AAAI conference on artificial intelligence (AAAI)*, vol. 33, 2019, pp. 4780–4789.

[22] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2019.

[23] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2019.

[24] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 10 734–10 742.

[25] A. Alaghi, C. Li, and J. P. Hayes, "Stochastic circuits for real-time image-processing applications," in *Proceedings of the Annual Design Automation Conference (DAC)*, 2013, pp. 1–6.

[26] W. Qian, X. Li, M. D. Riedel, K. Bazargan, and D. J. Lilja, "An architecture for fault-tolerant computation with stochastic logic," *IEEE transactions on computers (TC)*, vol. 60, no. 1, pp. 93–105, 2010.

[27] M. H. Najafi and M. E. Salehi, "A fast fault-tolerant architecture for sauvola local image thresholding algorithm using stochastic computing," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 24, no. 2, pp. 808–812, 2015.

[28] A. Alaghi and J. P. Hayes, "Fast and accurate computation using stochastic circuits," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–4.

[29] P. Li and D. J. Lilja, "Using stochastic computing to implement digital image processing algorithms," in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, Oct. 2011, pp. 154–161.

[30] S. KLEENE, "Representations of events in nerve nets and finite automata," *Automata Studies [Annals of Math. Studies 34]*, 1956.

[31] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural networks*, vol. 4, no. 2, pp. 251–257, 1991.

[32] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[33] M. Denil, B. Shakibi, L. Dinh, M. Ranzato, and N. De Freitas, "Predicting parameters in deep learning," in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, 2013, pp. 2148–2156.

[34] K. Hwang and W. Sung, "Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1," in *Proceedings of IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2014, pp. 1–6.

[35] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, 2014, pp. 1269–1277.

[36] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, Dec. 2016, pp. 1379–1387.

[37] S. Han, J. Pool, S. Narang, H. Mao, E. Gong, S. Tang, E. Elsen, P. Vajda, M. Paluri, J. Tran *et al.*, "Dsd: Dense-sparse-dense training for deep neural networks," in *Proceedings of the International Conference for Learning Representations (ICLR)*, Apr. 2017.

[38] Y. He, X. Zhang, and J. Sun, “Channel pruning for accelerating very deep neural networks,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017, pp. 1389–1397.

[39] J.-H. Luo, J. Wu, and W. Lin, “ThiNet: A filter level pruning method for deep neural network compression,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017, pp. 5058–5066.

[40] J. Lin, Y. Rao, J. Lu, and J. Zhou, “Runtime neural pruning,” in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, Dec. 2017, pp. 2181–2191.

[41] J.-H. Luo and J. Wu, “AutoPruner: An end-to-end trainable filter pruning method for efficient deep model inference,” *arXiv preprint arXiv: 1805.08941*, 2018.

[42] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks,” in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, 2016, pp. 2074–2082.

[43] S. Scardapane, D. Comminiello, A. Hussain, and A. Uncini, “Group sparse regularization for deep neural networks,” *Neurocomputing*, vol. 241, pp. 81–89, 2017.

[44] H. Hu, R. Peng, Y.-W. Tai, and C.-K. Tang, “Network trimming: A data-driven neuron pruning approach towards efficient deep architectures,” *arXiv preprint arXiv: 1607.03250*, 2016.

[45] G. Dundar and K. Rose, “The effects of quantization on multilayer neural networks,” *IEEE Transactions on Neural Networks*, vol. 6, no. 6, pp. 1446–1451, 1995.

[46] Y. Gong, L. Liu, M. Yang, and L. Bourdev, “Compressing deep convolutional networks using vector quantization,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, Apr. 2015.

[47] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2015, pp. 1737–1746.

[48] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, Apr. 2016.

[49] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “Xnor-net: Imagenet classification using binary convolutional neural networks,” in *Proceedings of the European Conference on Computer Vision (ECCV)*. Springer, 2016, pp. 525–542.

[50] E. Park, J. Ahn, and S. Yoo, “Weighted-entropy-based quantization for deep neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5456–5464.

[51] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks,” in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, 2016, pp. 4107–4115.

[52] ——, “Quantized neural networks: Training neural networks with low precision weights and activations,” *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[53] A. Zhou, A. Yao, K. Wang, and Y. Chen, “Explicit loss-error-aware quantization for low-bit deep neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 9426–9435.

[54] S. Jung, C. Son, S. Lee, J. Son, J.-J. Han, Y. Kwak, S. J. Hwang, and C. Choi, "Learning to quantize deep networks by optimizing quantization intervals with task loss," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 4350–4359.

[55] K. Wang, Z. Liu, Y. Lin, J. Lin, and S. Han, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 8612–8620.

[56] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *Proceedings of the International Conference for Learning Representations (ICLR)*, Apr. 2015.

[57] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Apr. 2014, pp. 269–284.

[58] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[59] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[60] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: ineffectual-neuron-free deep neural network comput-

ing," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 1–13.

[61] D. Kim, J. Ahn, and S. Yoo, "A novel zero weight/activation-aware hardware architecture of convolutional neural network," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2017, pp. 1462–1467.

[62] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[63] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 764–775.

[64] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019, pp. 749–763.

[65] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "Snapea: Predictive early activation for reducing computation in deep convolutional neural networks," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 662–673.

[66] D. Lee, S. Kang, and K. Choi, "Compend: Computation pruning through early negative detection for relu in a deep neural network accelerator," in *Proceedings of the International Conference on Supercomputing (ICS)*, 2018, pp. 139–148.

[67] G. Shomron and U. Weiser, "Spatial correlation and value prediction in convolutional neural networks," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 10–13, 2018.

[68] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 27–39.

[69] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2017, pp. 541–552.

[70] B. D. Brown and H. C. Card, "Stochastic neural computation. I. computational elements," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 891–905, Aug. 2002.

[71] ——, "Stochastic neural computation. ii. soft competitive learning," *IEEE Transactions on Computers*, vol. 50, no. 9, pp. 906–920, 2001.

[72] H. Sim and J. Lee, "A new stochastic computing multiplier with application to deep convolutional neural networks," in *Proceedings of the Annual Design Automation Conference (DAC)*, 2017, pp. 1–6.

[73] H. Sim, S. Kenzhegulov, and J. Lee, "Dps: dynamic precision scaling for stochastic computing-based deep neural networks," in *Proceedings of the Annual Design Automation Conference (DAC)*, 2018, pp. 1–6.

[74] R. Hojabr, K. Givaki, S. R. Tayaranian, P. Esfahanian, A. Khonsari, D. Rahmati, and M. H. Najafi, "Skippynn: An embedded stochastic-computing accelerator for convolutional neural networks," in *Proceedings of the Annual Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[75] J. Ba and R. Caruana, “Do deep nets really need to be deep?” in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, Dec. 2014, pp. 2654–2662.

[76] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” in *Proceedings of the Advances in Neural Information Processing Systems Workshop(NeuIPS Workshop)*, Dec. 2014.

[77] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, “FitNets: Hints for thin deep nets,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2015.

[78] P. Luo, Z. Zhu, Z. Liu, X. Wang, and X. Tang, “Face model compression by distilling knowledge from neurons.” in *Proceedings of the Thirtieth AAAI conference on artificial intelligence (AAAI)*, Feb. 2016, pp. 3560–3566.

[79] S. Zagoruyko and N. Komodakis, “Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, Apr. 2017.

[80] J. Yim, D. Joo, J. Bae, and J. Kim, “A gift from knowledge distillation: Fast optimization, network minimization and transfer learning,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jul. 2017, pp. 4133–4141.

[81] A. Mishra and D. Marr, “Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, Apr. 2017.

[82] A. Polino, R. Pascanu, and D. Alistarh, “Model compression via distillation and quantization,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2018.

[83] J. Kim, Y. Bhalgat, J. Lee, C. Patel, and N. Kwak, “Qkd: Quantization-aware knowledge distillation,” *arXiv preprint arXiv:1911.12491*, 2019.

[84] T. Furlanello, Z. C. Lipton, A. Amazon, L. Itti, and A. Anandkumar, “Born again neural networks,” in *Proceedings of the International Conference on Machine Learning (ICML)*, Jul. 2018, pp. 1607–1616.

[85] B. Heo, J. Kim, S. Yun, H. Park, N. Kwak, and J. Y. Choi, “A comprehensive overhaul of feature distillation,” in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2019, pp. 1921–1930.

[86] C. Li, J. Peng, L. Yuan, G. Wang, X. Liang, L. Lin, and X. Chang, “Blockwisely supervised neural architecture search with knowledge distillation,” *arXiv preprint arXiv:1911.13053*, 2019.

[87] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 8697–8710.

[88] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, “Practical block-wise neural network architecture generation,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2423–2432.

[89] K. O. Stanley and R. Miikkulainen, “Evolving neural networks through augmenting topologies,” *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[90] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, “Hierarchical representations for efficient architecture search,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2018.

[91] F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1251–1258.

[92] T. Elsken, J. H. Metzen, and F. Hutter, “Efficient multi-objective neural architecture search via lamarckian evolution,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2019.

[93] S. Saxena and J. Verbeek, “Convolutional neural fabrics,” in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, Dec. 2016, pp. 4053–4061.

[94] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, “Smash: one-shot model architecture search through hypernetworks,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2018.

[95] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, “Efficient neural architecture search via parameters sharing,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2018, pp. 4095–4104.

[96] S. Xie, H. Zheng, C. Liu, and L. Lin, “Snas: stochastic neural architecture search,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2019.

[97] M. Courbariaux, Y. Bengio, and J.-P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Proceedings of the Advances in Neural Information Processing Systems (NeuIPS)*, 2015, pp. 3123–3131.

[98] A. Wan, X. Dai, P. Zhang, Z. He, Y. Tian, S. Xie, B. Wu, M. Yu, T. Xu, K. Chen *et al.*, “Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[99] Y. Xu, L. Xie, X. Zhang, X. Chen, B. Shi, Q. Tian, and H. Xiong, "Latency-aware differentiable neural architecture search," *arXiv preprint arXiv:2001.06392*, 2020.

[100] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, Apr. 2011, pp. 315–323.

[101] Y. Cao, Y. Chen, and D. Khosla, "Spiking deep convolutional neural networks for energy-efficient object recognition," *International Journal of Computer Vision*, vol. 113, no. 1, pp. 54–66, 2015.

[102] P. U. Diehl, D. Neil, J. Binas, M. Cook, S.-C. Liu, and M. Pfeiffer, "Fast-classifying, high-accuracy spiking deep networks through weight and threshold balancing," in *Proceedings of International Joint Conference on Neural Networks (IJCNN)*, Jul. 2015, pp. 1–8.

[103] B. Rueckauer, I.-A. Lungu, Y. Hu, and M. Pfeiffer, "Theory and tools for the conversion of analog to spiking convolutional neural networks," in *Proceedings of the Advances in Neural Information Processing Systems Workshop(NeuIPS Workshop)*, Dec. 2016.

[104] R. Venkatesan, S. Venkataramani, X. Fong, K. Roy, and A. Raghunathan, "Spin-tastic: Spin-based stochastic logic for energy-efficient computing," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2015, pp. 1575–1578.

[105] X. Chen, L. Wang, B. Li, Y. Wang, X. Li, Y. Liu, and H. Yang, "Modeling random telegraph noise as a randomness source and its application in true random number generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 35, no. 9, pp. 1435–1448, Dec. 2015.

[106] H. Ichihara, S. Ishii, D. Sunamori, T. Iwagaki, and T. Inoue, “Compact and accurate stochastic circuits with shared random number sources,” in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, Oct. 2014, pp. 361–366.

[107] A. Alaghi and J. P. Hayes, “Exploiting correlation in stochastic circuit design,” in *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, Oct. 2013, pp. 39–46.

[108] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2015.

[109] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the International Conference on Machine Learning (ICML)*, Jul. 2015, pp. 448–456.

[110] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, May 2010, pp. 249–256.

[111] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, “On the importance of initialization and momentum in deep learning,” in *Proceedings of the International Conference on Machine Learning (ICML)*, Jun. 2013, pp. 1139–1147.

[112] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proceedings of the International Conference for Learning Representations (ICLR)*, May 2015.

[113] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520.

[114] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *arXiv preprint arXiv:1704.04861*, 2017.

[115] X. Zhang, X. Zhou, M. Lin, and J. Sun, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 6848–6856.

[116] T.-J. Yang, Y.-H. Chen, and V. Sze, “Designing energy-efficient convolutional neural networks using energy-aware pruning,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 5687–5695.

[117] Y. Chen, T.-J. Yang, J. Emer, and V. Sze, “Understanding the limitations of existing energy-efficient design approaches for deep neural networks,” in *Proceedings of the Conference on Systems and Machine Learning (SysML)*, 2018.

[118] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, “Shufflenet-v2: Practical guidelines for efficient cnn architecture design,” in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 116–131.

[119] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, “DaDianNao: A machine-learning supercomputer,” in *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2014, pp. 609–622.

[120] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, “Tetris: Scalable and efficient neural network acceleration with 3d memory,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 751–764.

[121] T. DeVries and G. W. Taylor, “Improved regularization of convolutional neural networks with cutout,” *arXiv preprint arXiv:1708.04552*, 2017.

// 초록

머신 러닝 (Machine Learning) 방법 중 현재 가장 주목받고 있는 딥러닝(Deep Learning)에 관한 연구들이 하드웨어와 소프트웨어 두 측면에서 모두 활발하게 진행되고 있다. 높은 성능을 유지하면서도 효율적으로 추론을 하기 위하여 모바일용 신경망 구조(Neural Network Architecture) 설계 및 학습된 모델 압축 등 소프트웨어 측면에서의 최적화 방법들이 연구되고 있으며, 이미 학습된 딥러닝 모델이 주어졌을 때 빠른 추론과 높은 에너지효율성을 갖는 가속기를 설계하는 하드웨어 측면에서의 연구가 동시에 진행되고 있다. 이러한 기존의 최적화 및 설계 방법에서 더 나아가 본 논문에서는 새로운 하드웨어 설계 기술과 모델 변환 방법 등을 적용하여 더 효율적인 추론 시스템을 만드는 것을 목표로 한다.

첫 번째, 새로운 하드웨어 설계 방법인 확률 컴퓨팅(Stochastic computing)을 도입하여 더 효율적인 딥러닝 가속 하드웨어를 설계하였다. 확률 컴퓨팅은 확률 연산에 기반을 둔 새로운 회로 설계 방법으로 기존의 이진 연산 회로(Binary system)보다 훨씬 더 적은 트랜지스터를 사용하여 동일한 연산 회로를 구현할 수 있다는 장점이 있다. 특히, 딥러닝에서 가장 많이 사용되는 곱셈 연산을 위하여 이진 연산 회로에서는 배열 승산기(Array Multiplier)를 필요로 하지만 확률 컴퓨팅에서는 AND 게이트 하나로 구현이 가능하다. 선행 연구들이 확률 컴퓨팅 회로를 기반한 딥러닝 가속기들을 설계하고 있는데, 인식률이 이진 연산 회로에 비하여 많이 뒤쳐지는 결과를 보여주었다. 이러한 문제들을 해결하기 위하여 본 논문에서는 연산의 정확도를 더 높일 수 있도록 단극성 부호화(Unipolar encoding) 방법을 활용하여 가속기를 설계하였고, 확률 컴퓨팅 숫자 생성기 (Stochastic number generator)의 오버헤드를 줄이기 위하여 확률 컴퓨팅 숫자 생성기를 여러 개의 뉴런이 공유하는 방법을 제안하였다.

두 번째, 더 높은 추론 속도 향상을 위하여 학습된 딥러닝 모델을 압축하는 방법 대신에 신경망 구조를 변환 하는 방법을 제시하였다. 선행 연구들의 결과를 보면, 학습된 모델을 압축하는 방법을 최신 구조들에 적용하게 되면 가중치 파라미터 (Weight Parameter)에는 높은 압축률을 보여주지만 실제 추론 속도 향상에는 미미한 효과를 보여주었다. 실질적인 속도 향상이 미흡한 것은 신경망 구조가 가지고 있는 구조상의 한계에서 발생하는 문제이고, 이것을 해결하려면 신경망 구조를 바꾸는

것이 가장 근본적인 해결책이다. 이러한 관찰 결과를 토대로 본 논문에서는 선행연구보다 더 높은 속도 향상을 위하여 신경망 구조를 변환하는 방법을 제안하였다.

마지막으로, 각 층마다 서로 다른 구조를 가질 수 있도록 탐색 범위를 더 확장시키면서도 학습을 가능하게 하는 신경망 구조 탐색 방법을 제시하였다. 선행 연구에서의 신경망 구조 탐색은 기본 단위인 셀(Cell)의 구조를 탐색하고, 그 결과를 복사하여 하나의 큰 신경망으로 만드는 방법을 이용한다. 해당 방법은 하나의 셀 구조만 사용되기 때문에 위치에 따른 입력 특성맵(Input Feature Map)의 크기나 가중치 파라미터의 크기 등에 관한 정보는 무시하게 된다. 본 논문은 이러한 문제점들을 해결하면서도 안정적으로 학습을 시킬 수 있는 방법을 제시하였다. 또한, 연산량 뿐만아니라 메모리 접근 횟수의 제약을 주어 더 효율적인 구조를 찾을 수 있도록 도와주는 페널티(Penalty)를 새로이 고안하였다.

주요어: 확률 컴퓨팅, 심층신경망 알고리즘 가속기, 신경망 압축, 신경망 구조 변환, 신경망 구조 탐색

학번: 2015-20950