M.S. THESIS

A New Approach to Binarizing Neural Networks

신경망을 이진화하는 새로운 방법에 대한 연구

FEBRUARY 2017

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

COLLEGE OF ENGINEERING

SEOUL NATIONAL UNIVERSITY

Jungwoo Seo

A New Approach to Binarizing Neural Networks

신경망을 이진화하는 새로운 방법에 대한 연구

지도 교수 최 기 영

이 논문을 공학석사 학위논문으로 제출함
2016 년 11 월

서울대학교 대학원
전기 · 정보공학부
서 정 우

서정우의 공학석사 학위논문을 인준함
2016 년 12월

위 원 장 김 태 환 (인)

부위원장 최 기 영 (인)

위 원 이 혁 재 (인)

Abstract

A New Approach to Binarizing Neural Networks

Jungwoo Seo

Department of Electrical and Computer Engineering

College of Engineering

Seoul National University

Artificial intelligence is one of the most important technologies, and deep neural network is one branch of artificial intelligence. A deep neural network consists of many neurons and synapses which mimic mammal's brain. It has attracted many interests from academia and industry in various fields including computer vision and speech recognition for the last decade.

It is well known that deep neural networks become more powerful with more layers and neurons. However, as deep neural networks grow larger, they suffer from the requirement of huge memory and computation. Therefore, reducing the overhead of handling them becomes one of key challenges in neural networks nowadays. There are many methodologies to address this issue such as weight quantization, weight pruning, and hashing.

This thesis proposes a new approach to binarizing neural networks. It prunes weights and forces remaining weights to degenerate to binary values. Experimental

results show that the proposed approach reduces the number of weights down to 5.35% in a fully connected neural network and down to 50.35% in a convolutional neural network. Compared to the floating point convolutional neural network, the proposed approach gives 98.9% reductions in computation and 93.6% reduction in power consumption without any accuracy loss.

Keywords: Deep neural network, Image recognition, Network pruning, Weight compression, Feedforward network

Student Number: 2015-20933

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Deep neural network research has made tremendous progress in the last several years. It has achieved significant improvements in various areas of artificial intelligence including computer vision, speech recognition, and even go playing. However, the use of deep neural network is sometimes very restrictive due to its large size and intensive computations. Its large size comes from a number of weights that it stores and intensive computation that comes from large number of multiplication it requires in feedforward process. Even worse is that many number of layers and neurons are essential to high performance neural network nowadays. For examples, GoogLeNet [1] which achieves state-of-art in ILSVRC2014 [2] uses more than 20 layers. To address these issues, many different techniques have been suggested including vector quantization [3], weight pruning [4], and hashing trick [5].

Binarized neural network is proposed by Hwang et al. [6] and Courbariaux et al. [7]. Including a more recent incarnation [8], they all propose using only +1 and -1 for the degenerate values of weights of a deep neural networks, which could be

considered as an oversimplification. To alleviate the problem, we let different neurons have different degenerate weight values instead of +1 and -1. For this, we first prune near-zero weights and find mean values of remaining weights in positive and negative area each. These two mean values are so small values that there's no big difference between their absolute values. So we can enforce these two mean weights to have one absolute values, $+\alpha_i$ and $-\alpha_i$. The value of α_i can be different for i^{th} neuron in a layer.

The pruning of near-zero weights reduces the number of weights. To further reduce the number of multiplications, we replace α_i to 1. To compensate for this replacement, the accumulation result of the products is multiplied by α_i. By doing so, all the multiplications of inputs and weights are replaced with simple sign changes and only one multiplication is left for each neuron.

In this paper, we present the detailed processes of binarizing weights in a feedforward neural network to significantly reduce the number of multiplications. We also show that high accuracy is maintained with much less computation and power consumption.

1.1 Thesis Organization

Chapter 2, related works, provides some previous work related to background on weight compression in neural network. It contains many kinds of compression and approximate computing neural network. This chapter gives essential technique to understand this paper and the motivation of our idea.

Chapter 3, proposed approach, provides a detailed technique to binarizing neural network that uses previous works. A method of combining two concepts,

weight pruning and binarized neural network, is demonstrated in this chapter.

Chapter 4, Implementation, presents the several algorithms and simulation environment that we use in the experiments actually. It also contains some pseudo-code that we used in experiments.

Chapter 5, Experimental result, investigates the operation of our technique and estimates its performance using real image data. Then compare the results with previous works. It also figures out the effect of other parameters to the accuracy and suggest a way to optimizing the parameters.

Finally, **chapter 6, conclusion and future work,** concludes the thesis and revisits the final goals of binarized neural network. Also, future research directions are reviewed.

Chapter 2

Related Work

2.1 Weight Pruning

Synapses are the connection between neurons which transmit information in mammal's brain. In human brain, the number of synapses increases as he ages and it improves memorial capacity. But this increase is not lasting forever due to lack of space. After few years of brain formative period, excessive synapses are destructed and removed. The reason of this process is not clear yet, however many researchers think that it is a related to energy efficiency in brain. Actually, all the process and reasons in mammal brain above are very similar to that of neural network research nowadays. The neural network also suffers from heavy memory problem for weights storing and massive energy consumption in computation. Therefore, synapse removing in brain is a kind of weight pruning in neural network.

Technique of weight pruning in neural network was firstly proposed before 1990s. Neural net pruning-why and how [9] uses weight pruning to make network fit in required size. It prunes neuron itself based on various input pattern and following neuron output. If output of a neuron is close to zero, it prunes the neuron.

Also any output sequence of neuron is same or reverse of other neuron, it can be replaced to that neuron so it pruned, too. It shows improvement in accuracy of correctly classifying noisy input patterns. Removing repetitive units also reduced the error rate as half. Although it is not experiment which was compared to recent experiments that use complex dataset MNIST [14] and ImageNet [2], but it showed the possibility of network pruning that time.

DropConnect [10] is a different form of weight pruning. It is a variation of dropout [11], but use a similar concept with weight pruning. It prunes weights rather than neuron for feed-forward process. But it is not related with network size or memory bound, because it contains all of weights and neurons entirely. It only prune some of the weights in feed-forward randomly. In short, they only concentrate on the effect of regularization in pruning. But this research also proves that pruning can be very effective to solve regularization problem the neural network.

[4] is state-of-art pruning technique in neural network. It prunes fully trained connection based on individual weight value. This scheme regards weights which close to zero do not contain important information, so prunes that kinds of weights. Besides, it uses L2 normalization in training so that makes all the weights as small as possible.

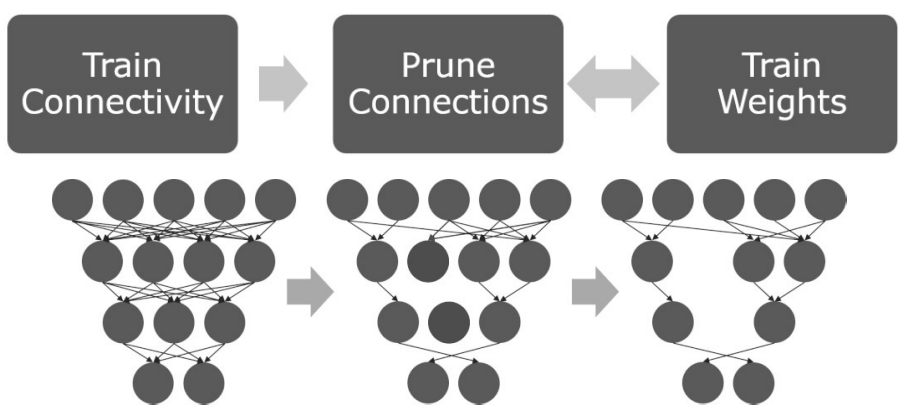


Figure 2-1. Mechanism of weight pruning.

They call it learning only important connection. But simple pruning leads to accuracy loss. Thus they retrain whole network with remaining weights only. Several steps of retraining let network more accurate and make additional weights that close to zero. They iterate this process many times. And if there's a neuron that has no connection, they also prune the neuron itself. This iteration makes the whole network sparse and accurate. The overall process is shown in Figure 2-1.

The result of [4] is so outstanding that they reduced the number of weights by a factor of 9x in AlexNet [12] and by 13x in VGG-16 [13]. Moreover, it shows better accuracy with much less weights when comparing with baseline model implemented in [12] experiment. There's could be some different reasons of accuracy rising, but regularization effect is the most convincible reason.

2.2 Binarized Neural Network

The low-precision network was used to reduce power consumption in neural network since a long ago. However, low-precision limits up to 4 bits to 8 bits due to accuracy loss. In fact, too much low-precision ruined the overall classification accuracy.

The binarized neural network is a kind of neural network which uses extremely low-precision. It uses only 2 values to represent weights and in many cases, they are +1 and -1. An example of binarized neural network is [6]. It proposes a fixed-point optimization method that reduce the length of weights. Actually it uses only ternary (+1, 0, -1) weights. It quantizes all the weights ternary values using greedy search as in Algorithm 1.

Algorithm 1 Greedy approach applied in [6]

1) Prepare a fully trained floating-point weights
2) Quantize all input data and signals of hidden layers
3) Quantize weights between first two layers by trying several step sizes
4) Choose the step size which minimize the output error
5) Iterates step 3 and 4 until it reaches the last layer

And it should be retrained to recover the original accuracy. After quantization, however, back-propagation does not work. Because the gap between quantization level is much bigger than small weight update. So, they modify the back-propagation algorithm little bit to properly accumulate the small amount of weight changes.

In experimental result, they achieve 1.08% of error with only ternary weight values and 3 bits signal word length using [14]. And at that time, about 86% of total weights are changed to zero and only 14.2% of weights remains. The original miss classification rate for the [14] test set was 0.97% with floating-point arithmetic. They show that only 0.11%-point accuracy drop in their research.

The biggest difference between [6] and our work is in the way of finding optimal quantization step size. As you can see in Algorithm 1, [6] suggests exhaustive search to find the step size. But in our approach, the step size is determined automatically after pruning. Of course, choosing a pruning threshold is another problem, and we discuss it in the chapter for experimental result.

Yet another difference is that [6] uses quantized values for the input and internal signals, whereas we use floating-point values for the signals. Quantization helps reduce the complexity of the neural network, and it would be the future work of our work.

Another famous binarizing technique is [7]. It is very similar with the work of

[6], but it shows better performance. It uses binary weight +1 and -1 in propagation process including forward and backward both. The overall algorithm is shown in Algorithm 2 below.

One remarkable thing in this paper is the weights distribution after binarization. It forms very steep bimodal distribution as shown in Figure 2-2. It is very important characteristic in this paper.

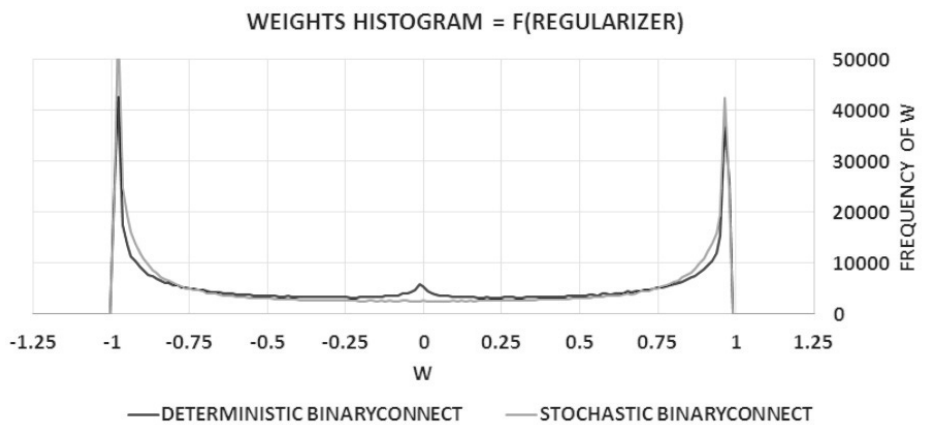


Figure 2-2. Weights distribution in BinaryConnect [7].

It also uses binary weight in forward and backward propagation. When updating weights, it requires floating-point weights and bias. It should contain both floating-point and binary weight matrix. It can be a waste of memory. Meanwhile, it shows noticeable performance. It tests the scheme in many data sets, and all of them shows near state-of-art result. And Cifar-10 [15], one of the data sets, achieves the best accuracy compared to other state-of-art papers.

There are many differences between [7] and our approach. First of all, [7] does not prune the network. In the case of [6], they use 0 weights and it is actually pruned weights. And in our approach, we prune unimportant weights at all. On the other hand, [7] keeps all the weights to +1 or -1. But the way of quantization used in [7] is similar with our approach. They deterministically decide whether quantize it to +1

or -1 based on the sign of each weights. It is very simple and effective way.

Algorithm 2 Stochastic Gradient Descent training with BinaryConnect

1) **Forward propagation:**
   Binarize all the weights
   Compute activation from the first to the last layer
2) **Backward propagation:**
   Initialize output layer's activations gradient
   Compute activation gradient from the last to the second layer
3) **Parameter update:**
   Compute weight gradient and bias gradient
   Compute the amount of weight updates and clip it
   Compute the amount of bias update

2.3 Approximate Neural Network

Stochastic Computing [16] is one of several method of approximate computing in neural network. It uses stochastic bit stream rather than floating point number to express numbers. The biggest advantage of this technique is that it requires very low computation hardware area. By using stochastic logic, they can replace multiplications with simple AND gates and adder with MUX.

Moreover, it uses stochastic bit stream to present number, so it is naturally fault tolerant. It also presents a capability to trade-off between computation time and accuracy without any hardware changes.

And recently, XNOR-Net [17] uses very similar concept with our proposal. They proposed two versions of binary convolutional neural network. One for using

binary weights only and the other for binary weights and inputs both.

First, it approximates the weights to binary number. It is shown in Formula, **I** is input matrix and **W** is weight matrix. **B** is binary value matrix and α is scaling factor. $\oplus$ is convolution without multiplications.

$$\mathbf{I} * \mathbf{W} \approx (\mathbf{I} \oplus \mathbf{B}) \cdot \alpha \tag{1}$$

The key point in the formula (1) is finding α and **B**. They estimate binary weights by minimizing error function. In the process of finding optimal values, derivative of error function should be computed. And the optimal value of α is:

$$\alpha^* = \frac{\sum |\mathrm{W_i}|}{n} = \frac{1}{n} ||\mathbf{W}||_{l1} \tag{1}$$

Parameter n is the number of weights in matrix **W**. As a result, the optimal scaling factor is the average of absolute weight values. And also the input can be binarized using same formula above.

Once the input and weights are binarized, the convolution can be replaced with XNOR operation and bitcount operations. They also proposed a way to training XNOR networks.

The result of XNOR network shows decreased accuracy compared to full precision [12]. It shows 44.2% of Top-1 and 69.2% of Top-5 in [2]. It is quite decreased one because [12] shows 56.6% of Top-1 and 80.2% of Top-5 in [2]. But it shows good result when using binary weight only. It shows 56.8% in Top-1 and 79.4% in Top-5 in [2].

The main difference between [17] and our work is the adoption of pruning. [17] does not use pruning, so it keeps many weights as in [7]. Of course, to compare [17]

and our work fairly, the possibility of applying pruning to [17] would be conducted first.

We apply the normal training algorithm to train the network. However, [17] uses a training algorithm specialized to it. It can be both advantageous and disadvantageous. It is easy to train when we use the specialized algorithm, but if we want to adjust the binarization of an already trained network, then our work will be the solution.

Chapter 3

Proposed Approach

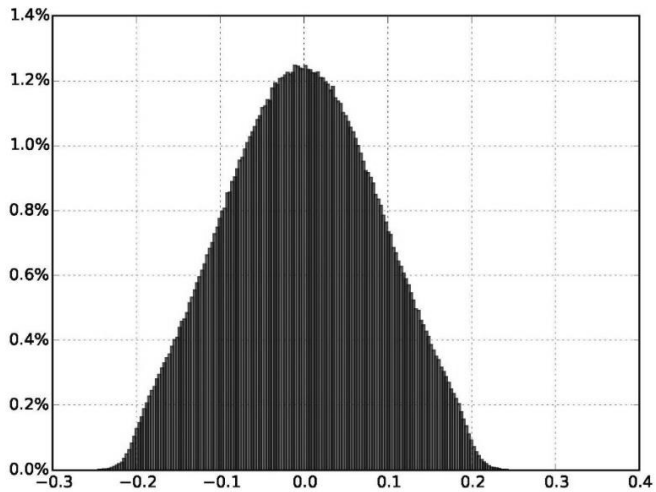
3.1 Motivational Example

There are some disadvantages in previous works. [10] and [7] use pruning as regularizer only although it can effectively reduce the number of unnecessary memory usage. And [6] requires additional searching process to find the best quantization step size. It is actually big burden when training the network. [4] gets advantages from both memory saving and regularization by using weight pruning. But, we think that we can make more compact and memory friendly neural network by combining weight pruning and binarizing technique.

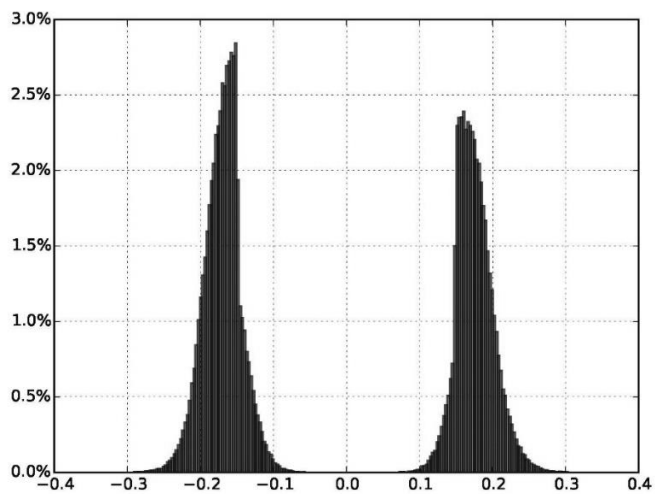
In most cases, normally trained weights forms Gaussian distribution as in Figure 3-1-(a). And weights distribution in binarized neural network forms different distribution as in Figure 2-2. But pruning in [4] makes this Gaussian distribution to bimodal distribution as in Figure 3-1-(b). Therefore, we think that if we utilize this characteristic of weight pruning, we can get a binarized neural network automatically in the process of pruning.

If we make a binarized neural network in the pruning process, we can avoid the

complex and long training process typically required in binarizing a neural network. Of course, this binarization would require other types of optimization including retraining and normalization.



(a)



(b)

Figure 3-1. (a) Normally trained weight that forms Gaussian distribution. (b) Pruned weight that forms bimodal distribution.

It looks similar to the approach of [17] since they share the same concept of changing weights to binary numbers and multiplying the weighted sum with a compensation factor. However, the key difference of our approach is in the utilization of pruning. As pruning is an essential technique in a neural network today and it also requires several times of retraining, our approach can be applied efficiently. This pruning also helps reducing the neural network size while maintaining the accuracy. On the other hand, [17] requires original weight size and its own specific process and structure.

3.2 Weight Compression

As mentioned in Chapter 3.1, we have to follow normal weight pruning process first. We call it weight compression step in this paper. Weight compression is a basic of this technique which is proposed in this paper, because binarizing process starts from pruned network.

However, before the weight compression, the neural network should be trained in a normally way. Similar to [4], it is a process of learning to see whether a connection is important or not. In the same manner, the accuracy of pruned neural network highly depends on the accuracy of original neural network. There also exists a possibility of accuracy increasing in pruning process by regularization. But we prepared a well-trained network in this paper.

When normally training, the L2 normalization should be applied to keep all the weights as small as possible. The L2 normalization is also known as weight decay in many neural network simulators. It adds a regularization term to the network's loss to compute the backpropagation gradient during training. Here is the basic formula for updating weights using gradient descent in neural network. The w_i is weight,

η is learning rate and E is error in formula (1).

$$w_i \leftarrow w_i - \eta \frac{\delta E}{\delta w_i} \tag{1}$$

The L2 normalization regularize the cost function by introducing a zero mean Gaussian prior over the weights. It changes the cost function to formula (2) below. The λ is regularization parameter.

$$\tilde{E}(w) \leftarrow E(w) + \frac{\lambda}{2} w^2 \tag{2}$$

Applying this new cost function to gradient descent, we can get a new weight updating rule as in formula (3) below.

$$w_i \leftarrow w_i - \eta \frac{\delta E}{\delta w_i} - \eta \lambda w_i \tag{3}$$

The trained weights for each neuron using L2 normalization typically form a Gaussian distribution and L2 normalization also leads weights closer to 0. Thus, it forms narrow Gaussian distribution whose weights are gathered around 0 as shown in Figure 3-1-(a). So pruning near 0 weights reduces the number of weights dramatically.

The pruning threshold is determined by multiplying proper scaling factor to the standard deviation of weights in a neuron. The pruning sensitivity in convolution layer is bigger than that in fully-connected layer and the first layer is the most sensitive to pruning. So we apply the smallest threshold to first layer and modify the threshold to fit to other layers.

Then the remaining weights after pruning form a bimodal distribution as shown in Figure 3-1-(b). The accuracy of pruned network depends on pruning threshold. If

we prune the weights severely, it would be compressed highly but the performance would be poor. In general, the pruned network shows accuracy as good as that of the original network when threshold is between 0.6 and 0.8. In the case of bigger threshold, accuracy of pruned network decreased drastically.

When we test this pruned network, accuracy dropped in most cases. The decreased accuracy could be recovered by retraining. But in this time, it uses only remaining weights. And the retraining epoch is much less than normal training epoch because all the weights are well distributed - like pre-trained model.

Then the second pruning is required. That's because retraining makes the bimodal distribution smooth. After pruning, we have to compute the mean values separately for the set of positive and negative weights in a neuron. But if a weight distribution in a neuron is too wide, there would be too much error in computing mean values. But pruning makes weight distribution steep. That's the reason why there required the second pruning. Also, when we prune the weights second time, almost all weights are pruned in the first pruning already and only a small number of weights placed to zero this time. So the second pruning helps to minimize mean error while does not decrease accuracy dramatically.

Finding the mean values is so simple that computing mean value of negative weights and positive weights each in a neuron. Let's set negative mean value as α and positive mean value as β. The n_i and n_j are the numbers of weights in each range. The formula is shown below in formula 4. We implemented this function using python.

$$\alpha \leftarrow \frac{\sum w_i}{n_i} \quad (i \ in \ negative \ range) \tag{4}$$

$$\beta \leftarrow \frac{\sum w_j}{n_j} \quad (j \ in \ positive \ range)$$

After finding the mean values, all weights in each set are degenerated to their mean value as shown in Figure 3-1-(b). It is called the quantization step. However, this simple quantization causes additional error. But in a different way from pruning, the error occurred in this step is unneglectable. The accuracy drops in quantization step range from 0.2%-point to 0.1%-point. Thus, retraining is essential to recover proper accuracy. But, before retrain the network, we have to modify weights once again.

Now we have to find one value which is equal to median value *m* of negative mean α and positive mean β. It can be computed as formula 5.

$$m \leftarrow \frac{|\alpha| + |\beta|}{2} \tag{5}$$

The value of α and β are very small due to L2 normalization and long pre-training. Therefore, there's no big difference between *m* and β. It means that there's no big accuracy drop either in this case. This process of pruning and binarizing forms a cycle of compression. This cycle is iterated to get a minimal sized network and to find optimal mean values.

3.3 Multiplication in Activation Stage

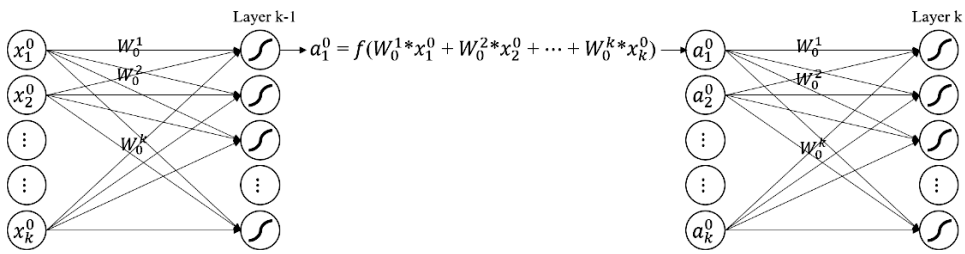
Even after pruning near-zero weights and binarizing the remaining weights, lots of multiplications are still required to obtain the products of inputs and binarized weights. Figure 3-2-(a) shows the conventional case where the input vector is multiplied with the weight matrix. Each element of the resulting vector goes through

an activation function. The total number of multiplications is the same as the number of elements in the weight matrix as in formula 6. The huge number of multiplications reduce operating speed and also consume lots of power.

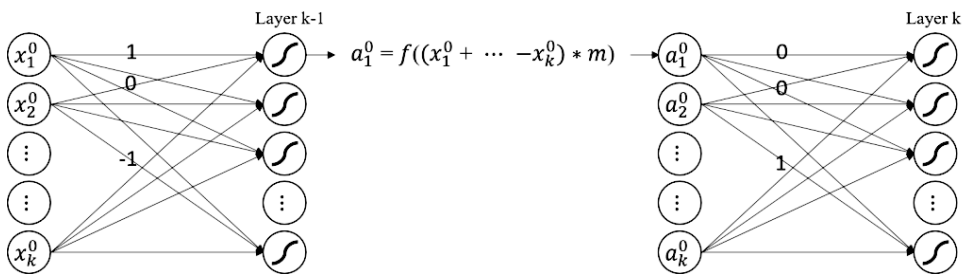
$$a_i \leftarrow f(x_1 w_1 + \cdots + x_i w_i + \cdots + x_n w_n) \tag{6}$$

$$a_i \leftarrow f((x_1 \cdot 1 + \cdots + x_i \cdot 0 + \cdots + x_n \cdot (-1))\alpha_i) \tag{7}$$

These massive multiplications can be effectively reduced in this binarized network since all the weights for a neuron have the same absolute value $|m_i|$ as shown in Figure 3-2-(b). This is simple trick that using distributive property. Then, all the weight values of $+m_i$ or $-m_i$ are changed to 1 or -1, respectively and only one m_i left in the activation function. As a result, the multiplication of m_i is placed after accumulation as in formula 7. In this way, we can reduce the number of multiplications.



(a)



(b)

Figure 3-2. Matrix multiplication in (a) normal, (b) binarized neural network.

Chapter 4

Implementation

We set up experiment environment using Caffe [18], which is a well-known deep learning simulator written in C++ and CUDA. In the [18] framework, the basic functions including back propagation and forward propagation are implemented already. But we have to implement some other functions like pruning, quantization, binarization. Also, the retraining is little bit different from normal training. Because pruned weights are not used in the retraining process. The implementation chapter consists of two branch. The first one is for pruning and the second one is for retraining.

First, we implemented a pruning process. It is written in python language using ipython notebook. The code for pruning is as below. The *val* is element in weights matrix, *std* is standard deviation of weights in a neuron. The multiplying rate can be different for each layer or each neuron. But we cannot experiment on the all possibility of different rate.

```
rate = 0.8
for x in range (# of neurons in layer n+1)
    for y in range (# of neurons in layer n)
        if (val[x, y] >= std[x] *rate) & (val[x, y] <= std[x] * rate)
            val[x, y] = 0
```

Figure 4-1. Code for pruning.

Then all weights under the threshold are pruned. The next step is mean value quantization. Before quantization, we have to find mean value of each area and standard deviation of each area. It can be computed easily in python. However, as it should be computed in each neuron, not layer, it consumes more time than expected. The code is in Figure 4-1.

```
for x in range (# of neurons in layer n+1)
    for y in range (# of neurons in layer n)
        if (val[x, y] > 0)
            val[x, y] = pm[x]
        if (val[x, y] < 0)
            val[x, y] = nm[x]
```

Figure 4-2. Code for quantization.

After the quantization, we have to find a median of two mean values and degenerates all weights to it. It is described in Figure 4-2 and Figure 4-3. In addition, when the degeneration ends, the weights are stored in *caffemodel* file to load in retraining process.

```
for x in range (# of neurons in layer n+1)
    for y in range (# of neurons in layer n)
        if (val[x, y] > 0)
```

```
                val[x, y] = (pm[x] – nm[x]) / 2
        if (val[x, y] < 0)
                val[x, y] = (nm[x] – pm[x]) / 2


net1.save('examples/mlp/zeropr/sameabs.caffemodel')
```

Figure 4-3. Code for binarization.

Second, we have to add new parameter for retraining. The pruning parameter is used in *prototxt* file that tells whether the layer is pruned or not. If the layer has pruning parameter, the layer is already pruned and we have to keep the pruned weights to 0 in the training. This retraining is implemented using mask matrix which has same size with weight matrix. It is implemented in *inner_product_layer.cpp* and it is different term of fully-connected layer. Pseudocode of implementation is shown below.

```
// pruning coefficient setting
pruning_coeff_ = this->layer.pruning_param;
pruned_ = (pruning_coeff_ == 0);


// In LayerSetUp
…


// mask matrix setting
If (pruning_coeff_ > 0)
    masks_.resize(this->size());
    masks_[weight].reset(weight_shape) // mask for weights
    caffe_set(masks_[weight], 1);
```

```
// In Forward_cpu()
…

// prune only once after loading the caffemodel
If (!pruned_)
        caffe_cpu_prune();
        pruned_ = true;

// In Backward_cpu()
…

//if (pruning_coeff_ > 0)
        if (backward propagation)
                caff_mul(mask_[weight], this->diff);
```

Figure 4-4. Code for retraining.

In the code above if it is set to prune, it copies all the weights to mask matrix and set all elements in mask matrix to 1. But if it is not set to prune, then it is same with normal training. Meanwhile, if it is already trained network, it loads *caffe_cpu_prune* function in forward propagation. It is pruning function which is implemented in *math_functions.cpp*. It gets inputs as number of elements in weights matrix, values of weight matrix and mask matrix. If weight is 0, then it passes 0 to mask matrix, too. It is shown in Figure 4-5 below.

```
void caffe_cpu_prune(){
        for (number of elements in weight matrix)
                if(weight[i] == 0)
                        mask[i] =0;
```

```
}
```

Figure 4-5. Code for mask matrix.

And then, all elements in mask matrix are multiplied with gradients of normal weights in back-propagation process. As a result, the gradients of 0 weights have gone. In this way, we can keep pruned weights from updating.

Chapter 5

Experimental Result

We demonstrate the effectiveness of the proposed approach using convolutional neural network and fully-connected neural network both of which are designed for the [14] benchmark dataset, running in [18] on Nvidia TitanX GPUs.

5.1 Convolutional Neural Network

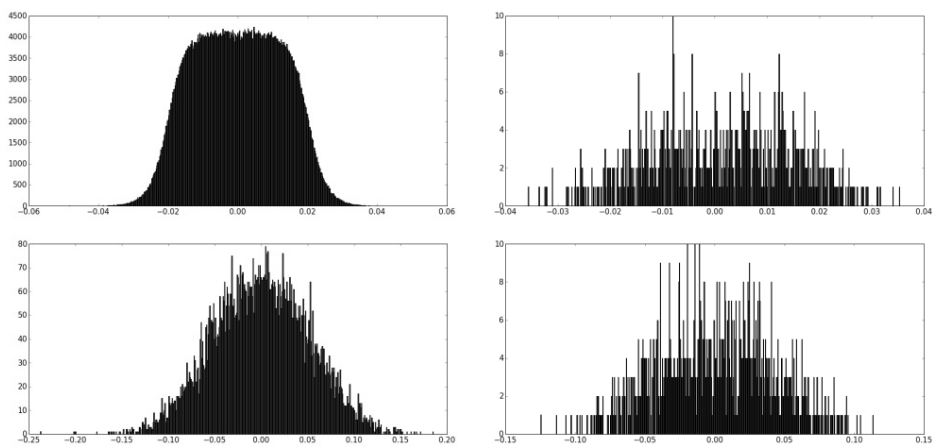
The CNN model contains two convolution layers, two pooling layers, one fully-connected layer with 1024 neurons and one output layer as shown in Table 5-1. As you can see in the Table 5-1, it is common that memory intensive problem in neural network occurs in fully-connected layer. Thus even we apply this scheme to convolution layers, overall memory deduction is fully-connected layer dominant.

Table 5-1. Topology of convolutional neural network

Layer	No Features	Filter Size	Total Weights
convolution 1	20	5 * 5	500
convolution 2	50	5 * 5	25000
fully-connected	1024	1	819200
output	10	1	10240

For details, we used max pooling layer after convolution layer and also used tangent hyperbolic as activation function. To prepare a normally trained neural network, we trained the network for about 50 epochs. Then the accuracy was 99.08% in this case. The weights distribution of normally trained neural networks are shown in Figure 5-1-(a) is graph of weights in hidden layer and output layer. As shown in upper graph of (a), there are a number of weights, it shows very dense Gaussian distribution. It is same with weights in output layer though it forms noisy Gaussian distribution.

On the other hand, in Figure 5-1-(b), the distribution of weights in a neuron is very sparse. Actually it is hard to say that it is a Gaussian distribution. But it shows similar distribution of weights in layer. In conclusion, weights in layer forms approximately Gaussian, and that in neuron forms approximately distribution of weights in layer. As it is far different from our expectation, we worried about the experimental result at first.

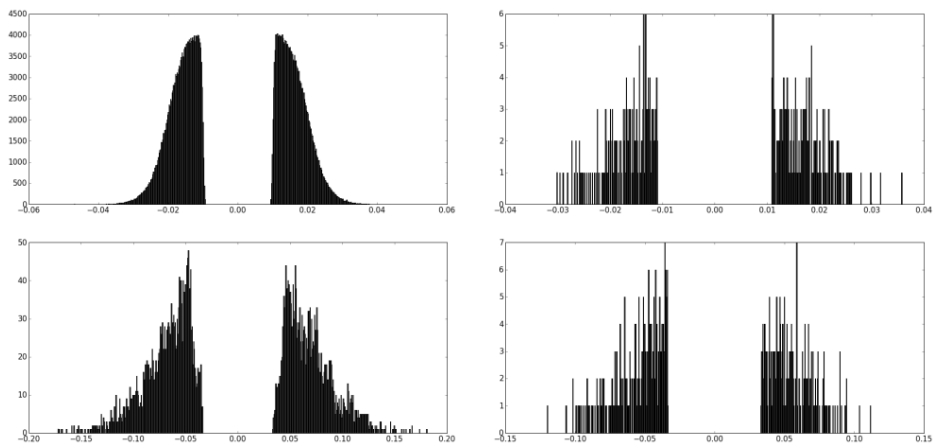


(a) (b)

Figure 5-1. Weights distribution of normally trained neural network. (a) Weights distribution in layers. (b) Weights distribution in a neuron.

The Figure 5-2 is weights distribution of pruned neural network. When pruning the weights, the threshold is determined based on the standard deviation of each

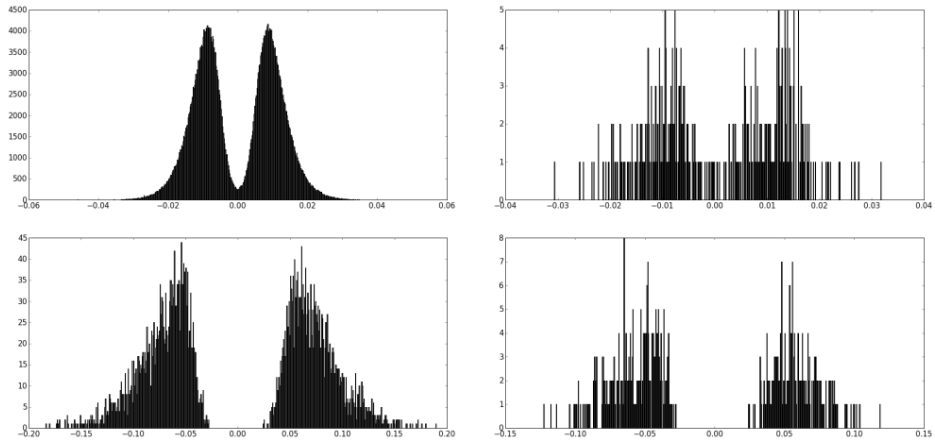
neuron, not layer. So in (a), its shape is not right angle around zero compared to original pruning result shown in [4]. It is similar in output layer. And (b) shows pruned weights in a neuron. We can check the number of weights are decreased little bit. But, there are only 800 weights and 1024 weights in a normally trained neuron. In this pruned network, the accuracy was 99.05% which is slightly decreased one compared to original accuracy. It would be due to loss of some important weights in pruning process.



(a) (b)

Figure 5-2. Weights distribution of pruned neural network. (a) Weights distribution in layers. (b) Weights distribution in a neuron.

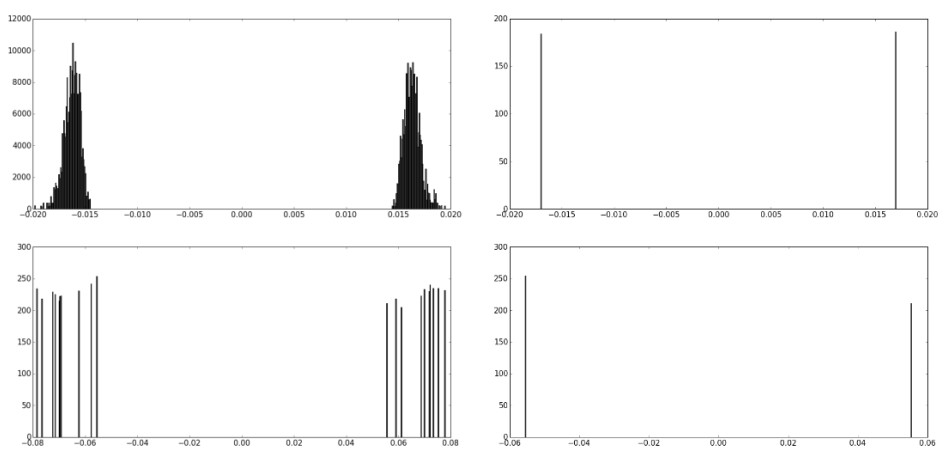
The pruning threshold for each neuron is chosen as a quality parameter multiplied by standard deviation of the weights for the neuron. The quality parameter is empirically set to 0.8 in this work, but it can be adjusted to some other number. In the experiments, α_i is calculated by taking average of the absolute of the two degenerate values (positive and negative). Since their absolute values are already very similar, there is no significant accuracy loss when replacing them with the average. After pruning, we executed retraining. Then the accuracy rises up to 99.10%. The retraining takes about 20 epochs.



(a) (b)

Figure 5-3. Weights distribution of retrained neural network. (a) Weights distribution in layers. (b) Weights distribution in a neuron.

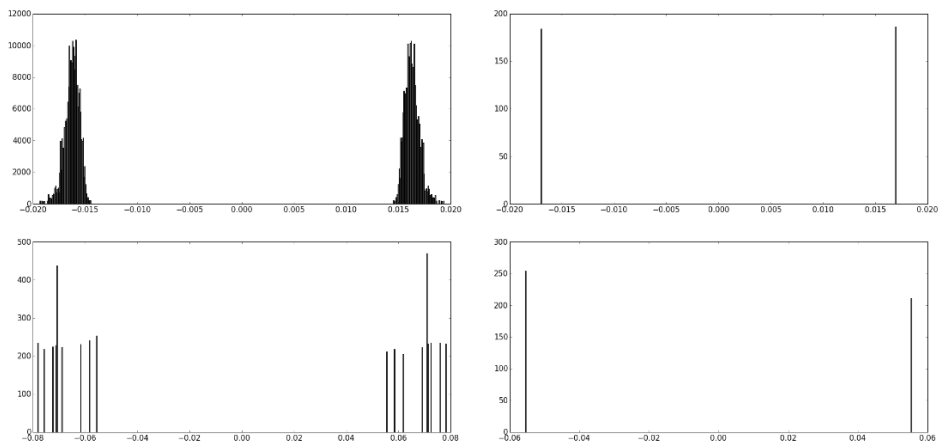
And amazingly, it does not decrease at all even after we quantize them to two mean values. It is exactly same accuracy with before. The graphs of distribution of weights in quantized network are shown in Figure 5-4 below. It looks like a retrained pruned network. The reason why the accuracy does not drop might be related to the weights shape. Because it is very similar to shape of just retrained weights. The weight distribution of a neuron in (b) also shows tendency to follow distribution of weights of layers in (a).



(a) (b)

Figure 5-4. Weights distribution of quantized neural network. (a) Weights distribution in layers. (b) Weights distribution in a neuron.

Lastly, the weights distributions of final result are shown in Figure 5-5. It is degenerated weights in all neurons to same absolute values. But there's no big difference between Figure 5-5 and Figure 5-4 above. For instance, in the first neuron in hidden layer, the mean value of positive and negative are 0.016973838 and -0.017014299 each. Therefore, the difference between each mean value and median is only 0.0000202305. It is too small to consider to effect to output of neural network.



(a) (b)

Figure 5-5. Weights distribution of binarized neural network. (a) Weights distribution in layers. (b) Weights distribution in a neuron.

Then the power consumption can be estimated by simulation. We synthesized a neuron with 1024 weights and measure its power consumption. It is a power consumption of a single neuron. So we have to multiply the number of neuron to the power consumption of individual neuron. Then, it is an approximate power consumption in neural network.

In this way, we measured power consumption of fully-connected layer in convolutional neural network. In this simulation, we used 16 bit fixed-point multiply and accumulation. So, if it is a floating point, the power reduction of binarizing would be much more powerful. The graph of comparison of power consumption between two neural network is in Figure 5-6.

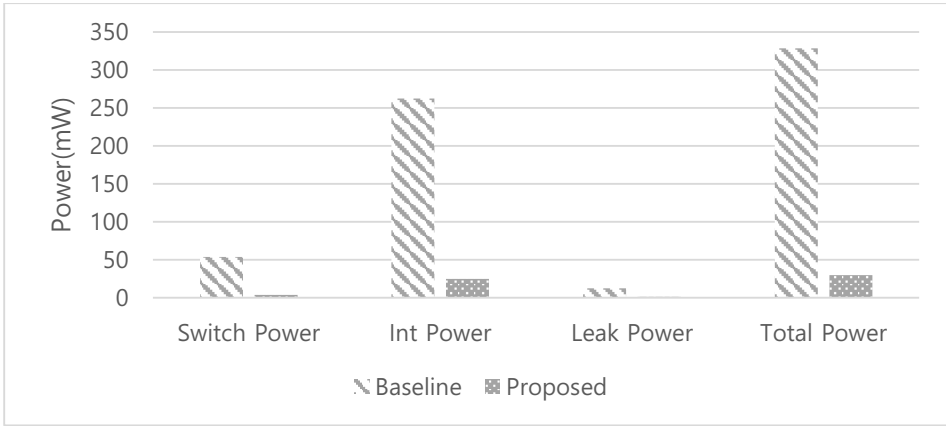


Figure 5-6. Comparison of Power consumption between fixed-point and proposed binarized neural network.

It shows four types of power consumption in a neuron. It is estimated considering the number of weights. The pruned weights are about half of original weights. So, baseline model assumes 1024 inputs and proposed model assumes 500 inputs. In both two cases, dynamic power dominants. But the quantity is very different. Overall power was 29.949 mW in proposed binarized neuron, as normal neuron consumes 328.631 mW in feed-forward. But it is assuming a power consumption in an output layer only. In fact, we have to consider other hidden layer which accepts output of convolution layer as input. As a result, experimental summary is in the Table 5-2 below.

Table 5-2. Result of convolutional neural network

Method	Accuracy	Weights	Multiplications	Power
Floating point	99.09%	100%	1058816	100%
Proposed	99.1%	50.3%	1034	6.4%

As shown in Table 5-2, the baseline model of floating point implementation shows very high accuracy, but also it has a large number of weights. It leads to a large number of multiplications in processing. However, after binarization using the

proposed technique, about 50% of the total weights are removed and the number of multiplications is reduced down to the number of neurons, while the accuracy increases slightly. It also achieves about 93.6% power reduction in the hidden layer.

The reason of accuracy increasing is not clear, but there's can be some candidates. One of the most reasonable hypothesis is regularization effect and longer training effect. As mentioned in Chapter 2, pruning has a regularization effect. It is proven in [4], either.

And the other reason is retraining. Although we cannot say that longer training time leads a higher accuracy always, generally it is true that longer trained neural network shows higher accuracy. But when we trained a network with same epochs with retrained version, it did not show higher accuracy than original accuracy. In the middle of training, it shows little bit higher accuracy sometimes. As training goes on, however, it decreased again. If we compare the highest accuracy in the normal training with result accuracy of this paper, there's a little gap about 0.03 percent point.

And then, we apply the pruning scheme to convolution layer. As in fully-connected layer, about 50% of total weights are pruned in the convolution layers. In general, convolution filter is used to extract features in original images. Therefore, the whole accuracy drops from 99.09% to 98.41% when pruning both convolution and fully-connected layers.

Another interesting thing is that once the weights are pruned, the quantization and binarization in convolution layer are not critical to accuracy as in fully-connected layer. For example, the accuracy of 98.41% above is same in pruning, quantization and binarization process. This effect is observed after retraining, too.

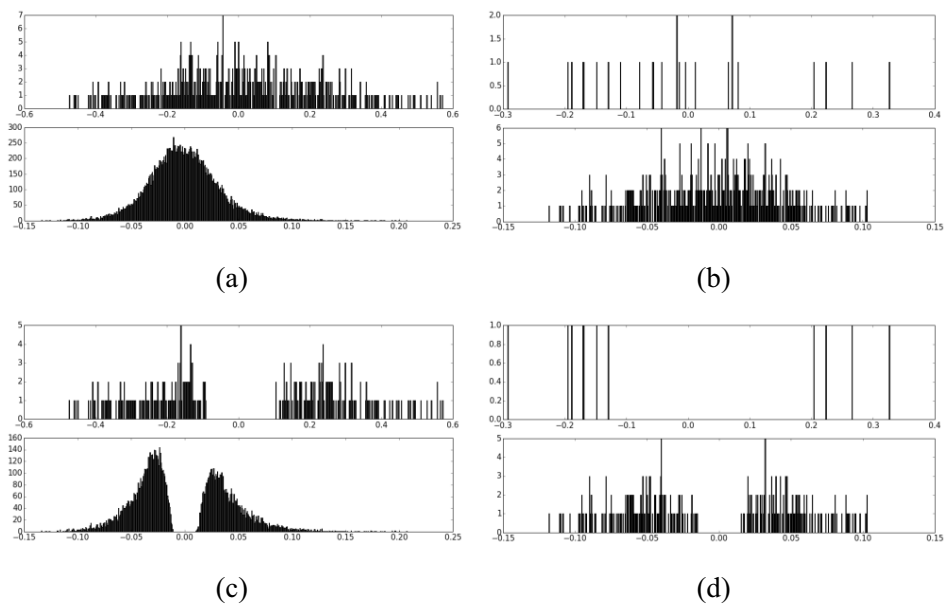
As a result, it maintains the total accuracy when both pruning fully-connected and convolution layers. The pruned weights are as shown in Table 5-3 below and the

total pruned ratio is 49.653%.

Table 5-3. Pruned weights in convolutional neural network

Layer	Num. Origin	Num. Pruned	Pruned Ratio
convolution 1	500	233	0.466
convolution 2	25000	13108	0.524
fully-connected	819200	405585	0.495
output	10240	5578	0.545

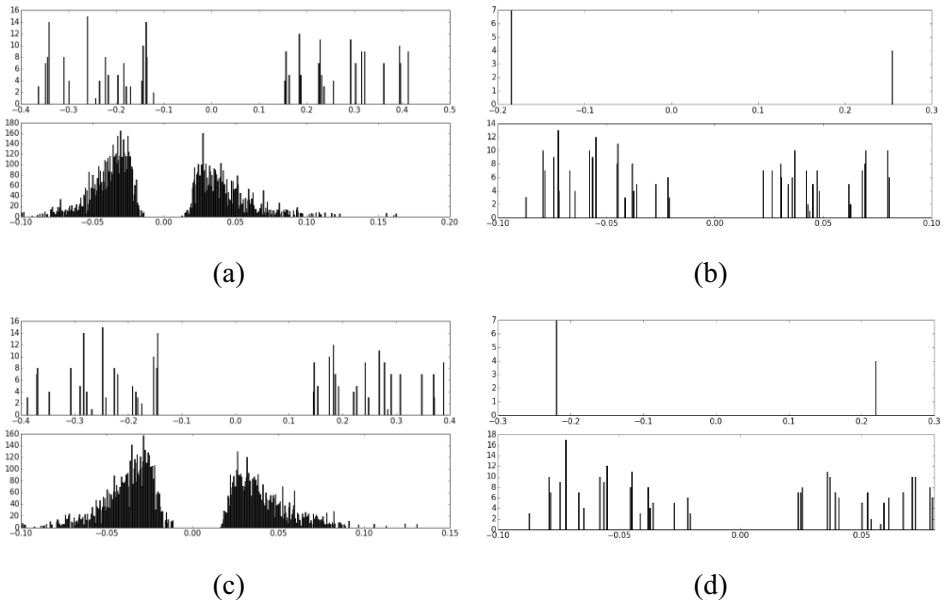
The weights distribution of convolution layer and neuron are shown in Figure 5-7 and Figure 5-8 below.



(a) (b)

(c) (d)

Figure 5-7. Weights distribution of convolution layers. (a) Weights distribution in normally trained layers. (b) Weights distribution in a normally trained neuron. (c) Weights distribution in pruned layers. (d) Weights distribution in a pruned neuron.



(a) (b)

(c) (d)

Figure 5-8. Weights distribution of convolution layers. (a) Weights distribution in quantized layers. (b) Weights distribution in a quantized neuron. (c) Weights distribution in binarized layers. (d) Weights distribution in a binarized neuron.

5.2 Fully-Connected Neural Network

For the fully-connected neural network model, we experiment on two kinds of topology to compare with previous work results. Both of [6] and [7] are composed of three fully-connected layers. Detailed topologies are shown in Table 5-3 and Table 5-4. We will use term topology 1 and 2 to describe them from now on.

Thus we implement our DNN on both topologies. Both in topology 1 and 2, we adjust [11] and batch normalization [19] and use an activation function ReLU. It is exactly same environment with [7], but little bit different from that of [6]. They used unsupervised greedy RBM learning before starting training. But in this experiments, we did not use unsupervised learning at all. Therefore, we show the accuracy drop in two cases to compare them fairly. At first, the concrete spec of topology 1 is in Table 5-3. The number of multiplication is equal to the number of total weights.

Table 5-4. Topology of Ternary FFDNN [6]

Layer	Num. Neurons	Total Weights
fully-connected 1	500	392000
fully-connected 2	500	250000
fully-connected 3	2000	1000000
Output	10	20000

The entire binarizing process and its accuracy is shown in Figure 5-9. It starts from the first iteration. Eventually, there is a baseline model. We prune the baseline model and retrain it. The baseline accuracy is 98.7% and then retrained accuracy is 98.74%. then it goes to iteration. Each iteration takes 20 epochs. At the first iteration, the accuracy of pruned network rises a bit. But after then accuracy fall drastically with quantization. The falling tendency is same in after binarizing. The first iteration shows sudden changes of accuracy according to proposed process.

It is similar in the second iteration. It also shows little increased accuracy after pruning and big accuracy drop after quantization and binarization. But the gap of increasing and decreasing is slightly decreased compare to the that of first iteration. And the result of binarization shows better accuracy then before.

The tendency of drastic change of accuracy disappears as iteration goes on. Finally, at fourth iteration, the gap between retrained accuracy and binarized accuracy is minimized in this experiment. Also as iterating the binarization, total weights are compressed more and more. As a result, it contains about only 5.3% of total weights.

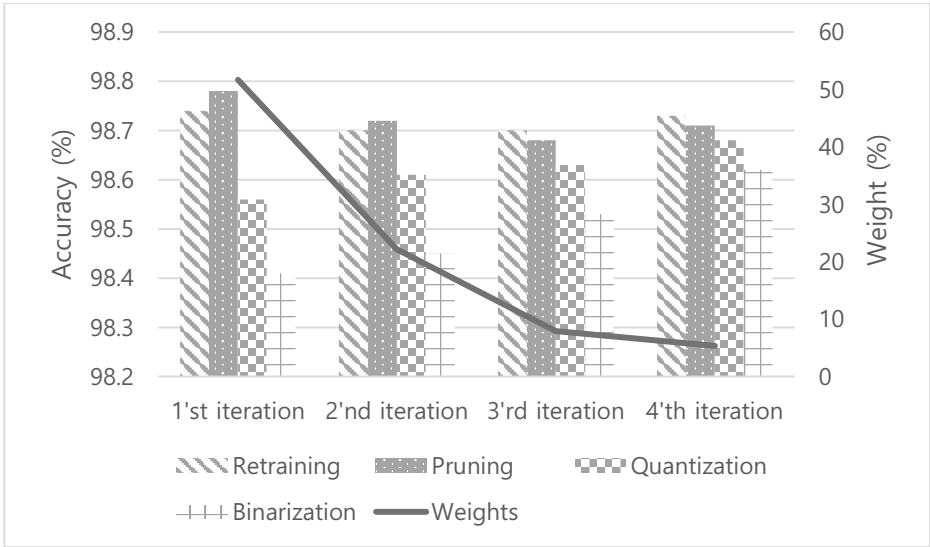


Figure 5-9. Graph of accuracy with weights ratio in topology 1.

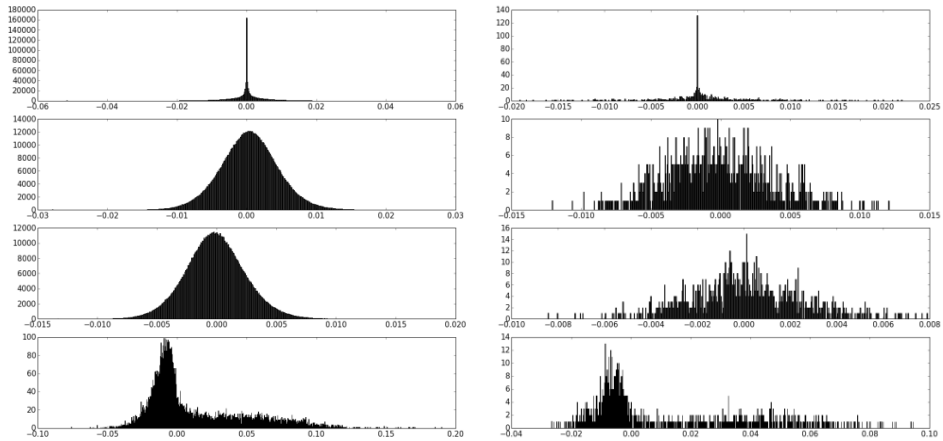
The topology 2 is shown in Table 5-4 below. The number of total weights is larger than that of topology 1. Entire experiment environment is exactly same with previous. But in this case, we can meet the baseline accuracy.

Table 5-5. Topology of BinaryConnect [7]

Layer	Num. neurons	Total Weights
fully-connected 1	1024	802816
fully-connected 2	1024	1048576
fully-onnected 3	1024	1048576
Output	10	10240

The experimental result and weights distribution are very similar to result of convolutional neural network and ternary FFDNN. But, there are some difference between them.

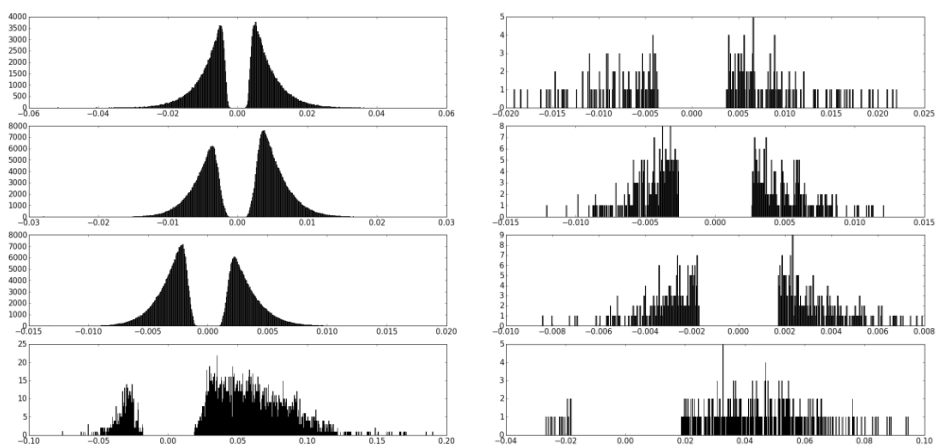
At first, if you see Figure 5-10, you can see that how many weights are concentrated on near zero in the first layer. Unlike with convolutional neural network, fully-connected neural network is more vulnerable to weight near zero.



(a) (b)

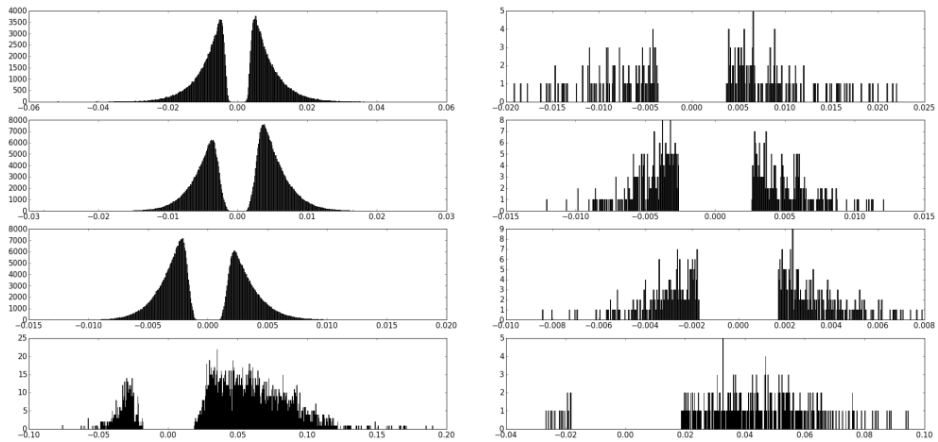
Figure 5-10. Weights distribution of baseline.

As we expected from Figure 5-10, after pruning, a number of weights are pruned at once as in Figure 5-11. The remaining weights forms bimodal distribution except for the last layer. It looks like unbalanced and has wide x range so that the distribution in right side would cause additional error. And eventually the accuracy after 1'st pruning shows 96.24%, which is bigger than 2.4%-point drop from original accuracy.



(a) (b)

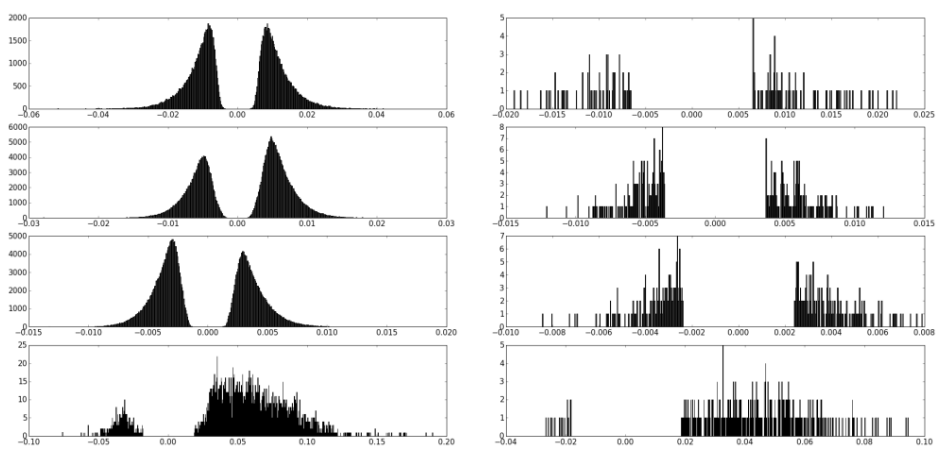
Figure 5-11. Weights distribution after 1st pruning.



(a) (b)

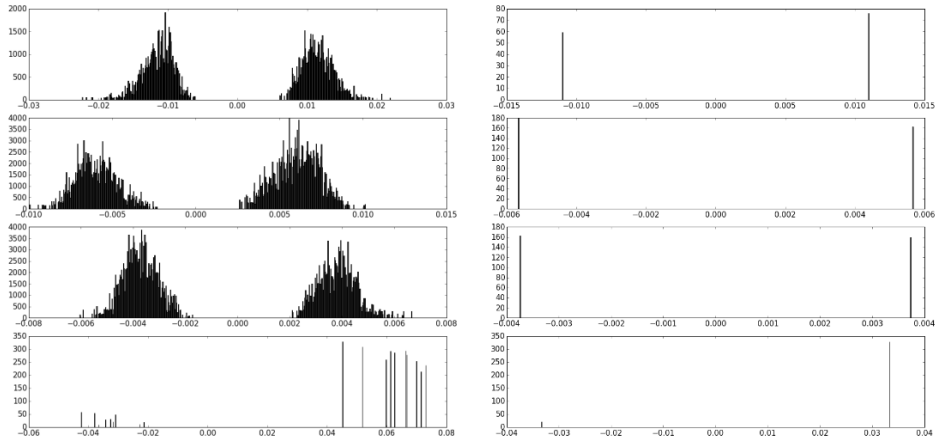
Figure 5-12. Weights distribution after retraining.

After the retraining, the accuracy rises up to 98.73%. It is higher than original accuracy. There is no special in the distribution of weights on Figure 5-12. The weights are slightly move to near zero. And 2'nd pruning changes the distribution a little. The entire number of weights are reduced. But the accuracy also decreased to 96.39%.



(a) (b)

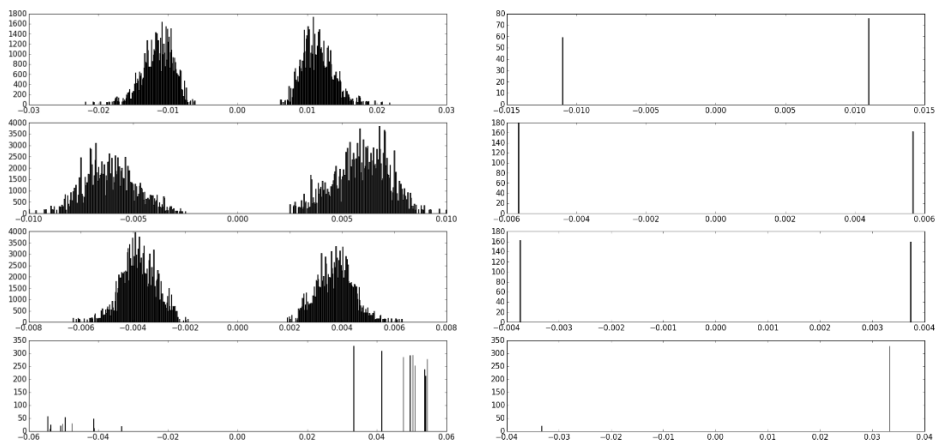
Figure 5-13. Weights distribution after 2nd pruning.



(a) (b)

Figure 5-14. Weights distribution after quantization.

After quantization, the accuracy rises a bit. It shows 96.95% of accuracy. The shapes of weights are entirely changed to discriminated to previous shape. Most of all, weights in a neuron are changed to binary values. And distribution of the last layer also changed a lot. It tends to follow the original shape, but it is very different from that. It looks very sparse now.



(a) (b)

Figure 5-15. Weights distribution after binarization.

And finally, weights after binarization is shown in Figure 5-15. The accuracy of binaized one shows 97.21%. It is much higher than pruned accuracy and quantized accuracy. Although the accuracy is not that high enough, it increased in a binarization process. Therefore, we iterate this cycle 4 to 5 times and get 98.58% of binarized accuracy at the end.

In addition, we experiment about pruning threshold. The key point of pruning and binarizing is how to determine moderate threshold or quantization step. In the previous work, [4], they do not demonstrate how to get the pruning threshold. And in [6], they use greedy search to find proper threshold from the first layer to the last layer. It consumes much resources and time either.

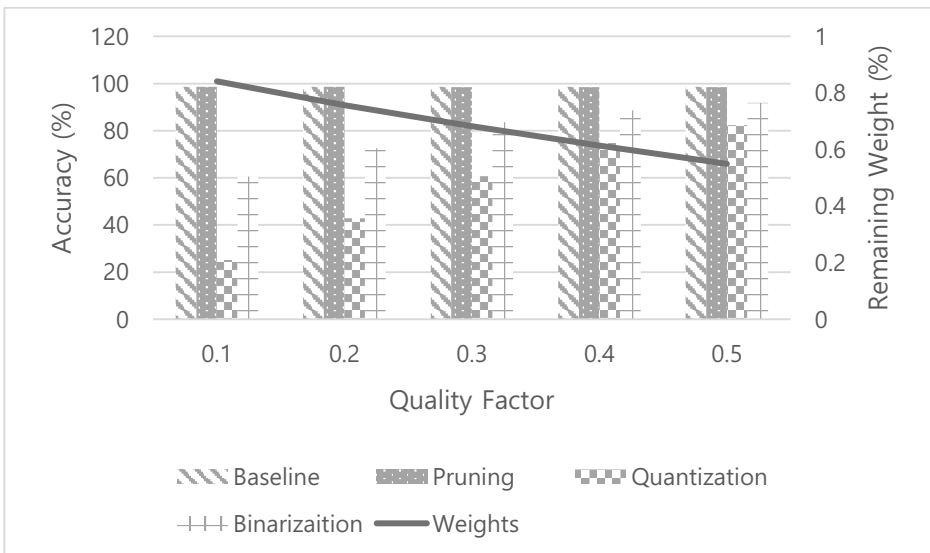


Figure 5-16. Accuracy and weights according to quality factor (1).

As shown in Figure 5-16, if the quality factor is small, like 0.1 or 0.2, there is no accuracy drop when pruning. None the less, total weights are reduced 20% at first. It looks efficient, but as we quantize the network, the situation changes. The accuracy drops to less than 40%. The reason of this dramatic decrease is the shape of

remaining weights. The removed 20% of weights are too few to make steep bimodal distribution. Thus it causes broad x range that makes big error when computing mean values. As a result, the computed mean value cannot represent other values around it. So the error increased. One thing amazing is binarized result is always better than quantized result. It might be due to a regularization effect.

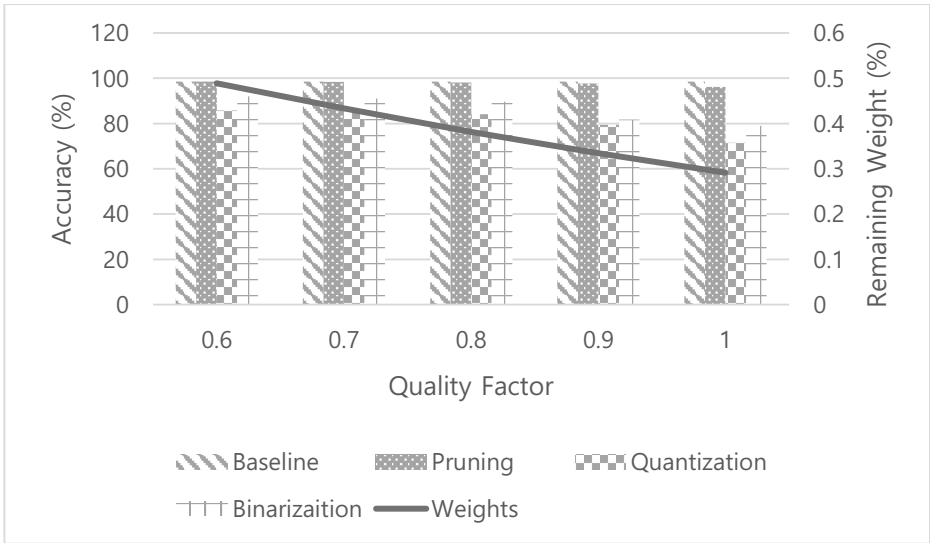


Figure 5-17. Accuracy and weights according to quality factor (2).

As quality factor grows, the accuracy gap between each process decreased as far. But after the quality factor 0.7 and 0.8, the gap increased again. If it pruned a lot, the remaining weights would be the most important weights. They are so important that approximated values cannot replace them. Thus it shows worse performance when quantized and binarized. As a result, we use 0.7 or 0.8 as quality factor in the all experiments.

Consequently, our work shows 0.41% of accuracy drop compare to [6], but achieves about 80% weights compression. And it also shows better result when comparing with [7], even though the baseline accuracy is different. In this case, the absolute accuracy is lower than [7], but the amount of drop in accuracy is slightly

better in our experiment.

Table 5-6. Comparison with previous works

Method	Accuracy			Remaining Weights
	Baseline	Binarized	Drop	
Ternary FFDNN	99.03%	98.92%	-0.11pp	14.2%
Proposed	98.70%	98.62%	-0.08pp	5.35%
Binary Connect	98.70%	98.99%	+0.29pp	100%
Proposed	98.70%	98.58%	-0.12pp	20.7%

Chapter 6

Conclusion and Future work

This thesis aimed to give a new way to binarizing feedforward neural network. We have used the concept of previous work, especially weight pruning [4] and binarized neural network [6, 7] and showed how those techniques could be combined in a neuron scale.

The experiment result shows that the proposed approach can reduce the number of weights down to 5.35% in fully-connected neural network and 50.3% in convolutional neural network. In the worst case, the accuracy drops from the first iteration, but it is recovered soon during retraining so that finally it shows accuracy which closes to original one.

Compared to previous approaches we can get much more compressed network, but the performance decreased due to trade-off between the number of weights and accuracy. In addition, we can save power consumption of computing accumulated sum in feed-forward neural network by removing multiplications. According to our experiments, the power consumption in a neuron decreased by 9.1% and in a layer decreased by 6.4%.

Finally, the main goal is training the network with binarized neural network. Many other works on training binarized neural network need both binarized weight matrix and normal weight matrix when update the values. If we can find an algorithm that does not require both weight matrix when updating and binarized convolution, it would be complete binarized neural network.

Bibliography

[1] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich; The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 1-9

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S.Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C.Berg, and L. Fei-fei. Imagenet large scale visual recognition challenge. Int. J. Comp. Vision, 115(3): 211-252, Apr. 2015.

[3] Gong, Yunchao, et al. "Compressing deep convolutional networks using vector quantization." arXiv preprint arXiv:1412.6115 (2014).

[4] Han, Song, et al. "Learning both weights and connections for efficient neural network." Advances in Neural Information Processing Systems. 2015.

[5] Chen, Wenlin, et al. "Compressing neural networks with the hashing trick." CoRR, abs/1504.04788 (2015).

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

[7] Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training deep neural networks with binary weights during propagations." Advances in Neural Information Processing Systems. 2015.

[8] Sarwar, Syed Shakib, et al. "Multiplier-less Artificial Neurons exploiting error resiliency for energy-efficient neural computing." 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2016.

[9] J. Sietsma and R. J. F. Dow, "Neural net pruning-why and how," IEEE 1988 International Conference on Neural Networks, San Diego, CA, USA, 1988, pp. 325-333 vol.1.

[10] L. Wan, et al., Regularization of neural networks using DropConnect",Proceedings of the 30th International Conference on Machine Learning (ICML-13), pp.1058–1066, 2013.

[11] Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." Journal of Machine Learning Research 15.1 (2014): 1929-1958.

[12] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems. 2012.

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

[14] LeCun, Yann, Corinna Cortes, and Christopher JC Burges. "The MNIST database of handwritten digits." (1998).

[15] Krizhevsky, Alex, and Geoffrey Hinton. "Learning multiple layers of features from tiny images." (2009).

[16] Kyounghoon, Kim, et al. "Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks." Proceedings of the 53rd Annual Design Automation Conference. ACM, 2016.

[17] Rastegari, Mohammad, et al. "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks." arXiv preprint arXiv:1603.05279 (2016).

[18] Jia, Yangqing, et al. "Caffe: Convolutional architecture for fast feature embedding." Proceedings of the 22nd ACM international conference on Multimedia. ACM, 2014.

[19] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015).

국문 초록

신경망을 이진화하는 새로운 방법에 대한 연구

오늘날 인공지능은 가장 중요한 기술이라고 해도 과언이 아니게 되었다. 그 중에서도 특히 심층 인공 신경망은 포유동물의 뇌를 모델로 개발되어 많은 수의 뉴런과 시냅스로 구성되어 있으며, 근래 몇 년 동안 음성인식과 영상처리를 비롯해 인공지능을 필요로 하는 수많은 분야에서 두각을 나타내고 있다.

심층 인공 신경망 기술은 뉴런의 수가 많고 신경망이 깊을수록 그 성능이 증가한다고 널리 알려져 있다. 그러나 이렇게 신경망이 깊어지고 커질수록 지나치게 많은 가중치를 저장할 공간 또한 더 크게 요구되며 계산량 또한 비약적으로 증가하게 된다. 따라서 가중치들과 계산량을 최대한 줄이는 것이 오늘날 심층 인공 신경망 분야의 가장 중요한 도전과제가 되었다. 실제로도 가중치 양자화 방법, 불필요한 가중치를 버리는 방법, 그리고 해시를 이용한 방법 등 이 문제를 해결하기 위해 수없이 많은 연구가 진행되고 있다.

본 논문에서는 신경망을 이진화하는 새로운 방법을 제시한다. 그 방법은 가중치들을 잘라내고 남은 가중치들을 이진화 시키도록 강제하는 것이다. 실험결과에서 본 논문이 제시하는 방법으로 가중치의 수를 다층 퍼셉트론 신경망에서는 전체의 5.35%로 줄이고 합성곱 인공 신경망에서는 50.3%로 줄일 수 있음을 보였다. 또한, 부동소수점

연산을 사용한 합성곱 인공 신경망과 비교했을 때 본 논문이 제안하는 방법을 사용하여 전체 곱셈 계산량의 98.9%를 감소시켰으며, 전력 소모 또한 성능에 어떠한 영향도 없이 93.6%를 절감시켰다.

Keywords : **심층 인공 신경망, 패턴인식, 가중치 압축, 가중치 가지치기, 순방향 전파 신경망**

Student Number : 2015-20933