Log-or-Trig: Towards Efficient Learning in Deep Neural Networks

CAI JINGYONG 19834311

Graduate School of Engineering
Tokyo University of Agriculture and Technology

Supervisor: Prof. Nakajo Hironori

Jan. 2022

Contents

List of Figures

List of Tables

Abstract

Based on the fact that parameters of pre-trained neural networks naturally have non-uniform distributions, logarithmic quantization of network parameters can achieve better classification results than linear quantization of the same resolution. In our practice, we found that the logarithmic quantization suffers huge accuracy decrease on small size neural networks. This is because the parameters of trained small neural networks are not highly concentrated around 0. In this research, we analyse in depth the attributes of logarithmic quantization. In addition, existing compression algorithms highly rely on retraining which requires heavy computational power. In such a situation, we propose a new logarithmic quantization algorithm to mitigate the deterioration on neural networks which contain layers of small size. As the result, our method achieves the minimum accuracy loss on GoogLeNet after direct quantization compared to quantized counterparts.

Moreover, many deep neural network quantization algorithms use the mean square quantization error (MSE) as the optimization objective, which we found is not suitable: MSE can easily lead the quantization algorithm to assign more precision to weights that are near zero because they dominate in quantity. Instead, we present a weight-aware optimization objective for deep neural network quantization. Our method quantifies the importance of each weight by its magnitude. This method requires no sorting during quantization; therefore, the computation cost is very low and can be easily combined with existing quantization algorithms. We tested our method using existing state-of-the-art quantization methods. On Additive Powers-of-Two (APoT) quantization, our method can largely reduce the retraining time and on Piecewise Linear Quantization (PWLQ), our method improves the image classification accuracy.

In addition to the quantization algorithms, we also propose a new learning algorithm. Despite being heavily used in the training of deep neural networks, multipliers are resource-intensive and insufficient in many different scenarios. Previous discoveries have revealed the superiority when activation functions, such as the sigmoid, are calculated by shift-and-add operations, although they fail to remove multiplications in training altogether. Based on the aforementioned facts, we propose an innovative approach that can convert all multiplications in the forward and backward inferences of deep neural networks into shift-and-add operations.

Because the model parameters and backpropagated errors of a large deep neural network model are typically clustered around zero, these values can be approximated by their sine values. Multiplications between the weights and error signals are transferred to multiplications of their sine values, which are replaceable with simpler operations with the help of the product to sum formula. A rectified sine activation function is also proposed for further converting layer inputs into sine values. In this way, the original multiplication-intensive operations can be computed through simple add-and-shift operations.

This trigonometric approximation method provides an efficient training and inference alternative for devices with insufficient hardware multipliers. Experimental results demonstrate that this method is able to obtain a performance close to that of classical training algorithms. The approach we propose sheds new light on future hardware customization research for machine learning.

Chapter 1

Introduction

1.0.1 Background

Deep neural networks show their incomparable performance in computer vision, natural language processing, visual art processing and so forth. However, on one hand, since the training of deep neural networks is extremely time and resource consuming, many researchers are committed to reducing the amount of computation so as to train neural networks fast and easily. On the other hand, the huge storage cost of trained neural networks hinders their deployments on mobile or IoT devices.

Applications and Challenges

Although neural networks have been used in a wide range of industries, there are still some challenges that make it difficult to apply neural networks in some areas. The size of neural networks and the difficulty of training them are among the reasons that prevent them from being more widely used. For instance, neural network inference tasks on mobile devices face are very difficult for the limitation of the computational performance on mobile devices. Many mobile applications require the application of neural networks. For example, today's smartphones can do portrait recognition and segmentation by neural networks to simulate the depth-of-field effect of an SLR (single-lens reflex camera). Neural networks are even used in real time by mobile devices for computationally intensive video quality optimization tasks, such as Hdrnet ([4]), which is used by Google's pixel phone for video tasks. Not only that, now some smartphones can be waked up by voice or used to translate in real time without network, etc. also need a lot of machine learning involved.

Our demand for smartphone intelligence continues to increase. Today, smartwatches can do sleep monitoring, disease prevention, etc., and street lights on the highway even have intelligence (Figure 1.1), these devices can not provide powerful computing power. We can expect to delegate more and more deep learning tasks to mobile, which is limited by processor performance and battery life. Therefore, we need to reduce the computation burden and model complexity of neural networks as much as possible when we do neural network inference on mobile.

The training of neural networks is also facing some challenges. It often takes a lot of computational resources when training neural networks, and some complex tasks take a long time to train. Today's neural networks rely heavily on matrix multiplication, which in turn relies heavily on multipliers, leading to a particularly high importance of hardware resources in neural network training and research.

Figure 1.1: Some scenarios that require efficient neural network inference.

Composition of this thesis

In this paper, we approach the above problem from two main aspects. One is neural network quantization methods, and the other is an attempt to construct more efficient neural network learning algorithms.

In the next part of this chapter, we will introduce some basics of machine learning. In Chapter 2 we will present related previous research. In Chapter 3 we will explain our work on neural network quantization, and Chapter 4 introduces a neural network learning algorithm that does not rely on multipliers. We conclude our work and bring discussions in Chapter 5.

1.0.2 Deep Neural Networks

The power of a deep neural network comes from its fitting capabilities. For instance, we can assume an unknown objective function or probability distribution that reveals the relation between samples and their categories. Since it is difficult to find this objective function directly, we use a neural network to fit it. After the training stage and all inputs and outputs are consistent, we assume that the neural network successfully fits the objective function.

As shown in Figure 1.2, a fully connected neural network is composed of neurons layer by layer. All connections between layers are weighted and usually initialized to random variables. When training a neural network, the training dataset is first batched and then input into the neural network. Accumulated multiplication and addition operations are then calculated between the inputs and weights in the first layer. The output of the first layer becomes the input to the next layer and the same computation forwards recursively until the last layer.

The output of the last layer is compared with the output of the objective function. The difference can be propagated back to the hidden layer of the neural network as the error signal. Through backpropagation the weights are updated to minimize the error. In addition, there is an activation function between each layer. The purpose of the activation function is to introduce non-linearity so as to enhance the fitting ability of neural networks. Sometimes, in order to further enhance the fitting ability, we add an offset term to each connection. In this case, we need to add a bias term after the multiplication and addition operation. Equation 1.1 gives the forward computation between each layer in a fully connected neural network.

$$y(\mathbf{x}) = \sigma\left(\sum_{i=0}^{N-1} \mathbf{W_i j x_i} + b\right) \tag{1.1}$$

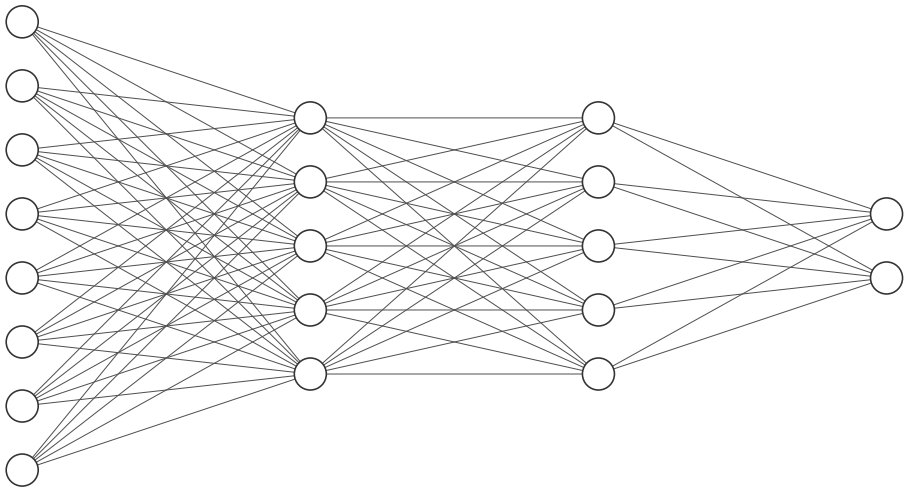


Figure 1.2: A simple fully connected neural network.

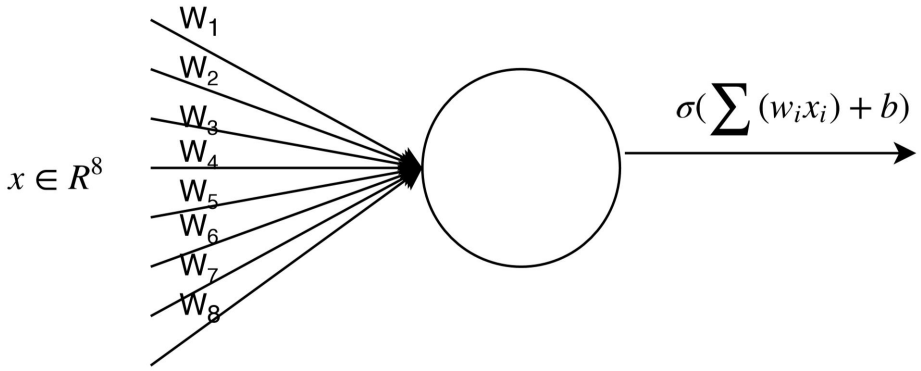


Figure 1.3: A single node from the first hidden layer.

1.0.3 Network Training

As introduced in Section 1.0.2, we assume a neural network can fit the latent objective function. In practice, we adjust the weight of every connection during training so that the output of the neural network can be consistent with the target output as much as possible. Usually, we package multiple inputs into a batch so that the neural network can generalize better (a theoretical analysis can be found in [5]). Backpropagation is a mechanism to minimize the error between objective outputs and neural network outputs (Figure 1.4).

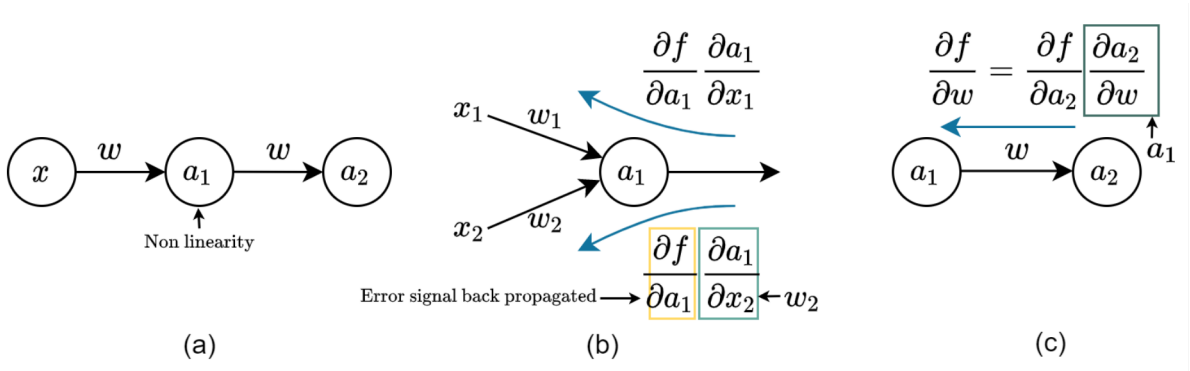


Figure 1.4: A simple illustration of forward and backward inference in neural network training. Subfigure (a) denotes the forward inference of neural networks, where non-linearity is introduced during the activation stage. Subfigure (b) and (c) represent the error backpropagation of neural networks. As indicated by the equations in (b) and (c), the error signals are backpropagated by computing the gradients of each weight by the chain rule.

When training a target neural network, we first feed the input batches into the neural network and compute predictions. For a simple fully connected network this step is performed via dot products layer by layer. A loss function is then used to derive the differences (errors) between the predictions and labels. Note that the labels are usually encoded in the same form as the output of the target network. Note in this step, non-linear activation functions play an important role in the neural network's fitting ability. The activation step is performed after dot products layer by layer. There are various forms of activation functions and at the core of all these methods is the introduction of non-linearity for neural networks. Subfigure (a) in Figure 1.5 is a representative case (sigmoid: $(\frac{1}{1+e^{-x}})$) of traditional activation functions. Subfigure (b), ReLU (rectified linear unit: $max(x, 0)$)), is widely used recently since it is highly efficient and avoids the vanishing gradient problem.

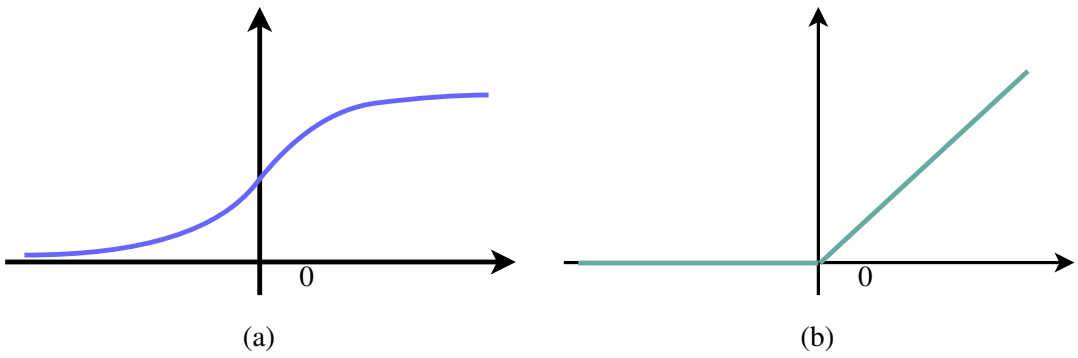


Figure 1.5: Non-linearity. (a): A traditional non-linear activation function (sigmoid). (b): A modern non-linear activatioin function (ReLU).

Secondly, the error signal is backpropagated to compute the gradient of each weight through recursive application of the chain rule. The gradient is used to perform a gradient descent algorithm on all weight parameters so that the final error could be minimized. Generally, many epochs of training need to be performed to achieve a good result (one epoch is one iteration over all training samples). When the dataset and neural network size are huge, the training cost can be prohibitive. Some large models even require large GPU clusters for training. For instance, in 2017 at Facebook, 256 GPUs was used for fast

training on the ImageNet dataset ([6]).

Gradient descent methods are used to find the set of parameters that can minimize the error of the loss function. Since the weights always point in the opposite direction of the steepest descent, gradient descent algorithms take the repeated steps toward the opposite direction of the gradients. When adjusting the weights during training, we always chose a step size (learning rate) so that the weight will not "overstep" (Equation 1.2).

$$w_j := w_j - \alpha \frac{\partial}{\partial w_j} J(w) \tag{1.2}$$

In practice, the direction adjustment and step size might be computed in a more complicated approach that requires more information. For instance, one might simulate a physical way that accumulates precious momentum so that a better direction can be obtained [7]. Many gradient descent optimization methods are studied and it is still an important direction of tuning neural network learning ability [8–11].

In recent years, neural networks are getting deeper and more complex. Along with more layers, there are much more model parameters. Convolutional neural networks (Figure 1.6) use filters to extract information from inputs or former layers. Some convolutional networks use big filters such as filters of 11 by 11. In this case, only one filter itself requires 11^2 parameters. If many filters are used for convolution, the network size may get large which hinders the implementation on small devices.

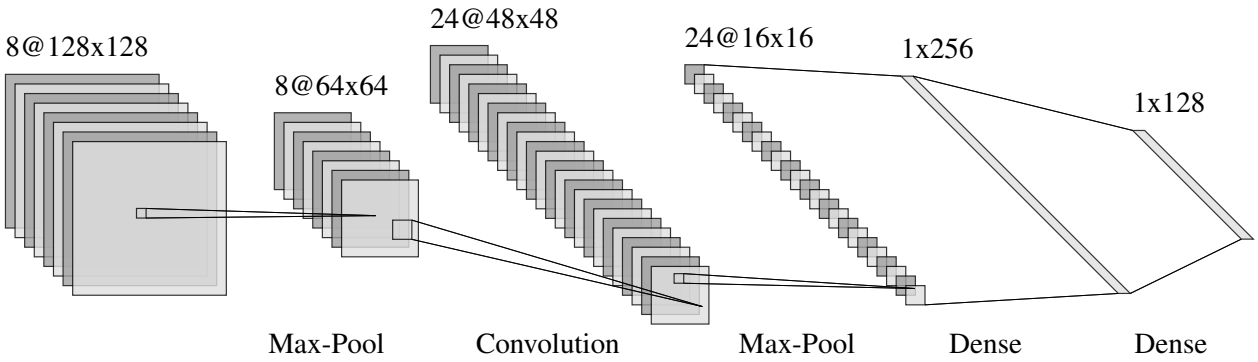


Figure 1.6: Filters in convolutional neural networks.

1.0.4 Computation Cost Reduction

Some of the difficulties mentioned above can be effectively alleviated with the compression of neural networks. There are several common means of neural network compression as follows.

Network Pruning

As depicted by Figure 1.7, network pruning is to reduce the computational cost by cutting off some of the connections of the neural network. According to NVIDIA's pruning test [12], 40% FLOPS reduction on ResNet-101 was achieved by removing 30% of the parameters, with almost no loss in the top-1 accuracy on ImageNet.

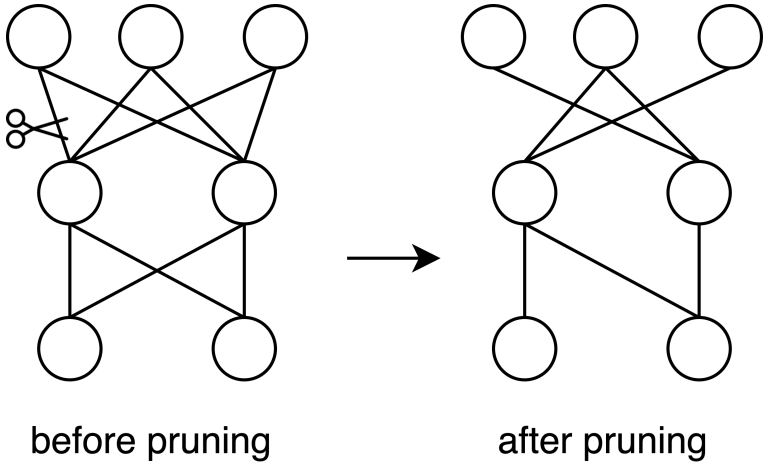


Figure 1.7: Network pruning.

However, current deep learning frameworks do not support pruning very well, and the operation of pruning is basically done by a 0-setting operation, which means that the weights pruned will still be involved in the matrix multiplication operation and will not significantly accelerate the operation.

```
def layer_mask(weights, pruning_rate):

    weight_abs = abs(weights.data)
    weight_data = sort(xp.ndarray.flatten(weight_abs))
    num_prune = int(len(weight_data) * pruning_rate)
    idx_prune = min(num_prune, len(weight_data)-1)
    threshould = data[idx_prune]

    mask = weight_abs
    mask[mask < threshould] = 0
    mask[mask >= threshould] = 1
    return mask

# The pruned weights are still involved in matrix multiplications afterwards.
def prune_weight(weight, mask):

    return weight * mask
```

Listing 1.1: forward and backward pass of the AdderNet.

Network Quantization.

Network quantization techniques quantize the original 32 bits model parameters into smaller bitwidth (for instance 4 bits or 8 bits). The most simple quantization method is linear quantization, which projects the model parameters to smaller quantization levels linearly.

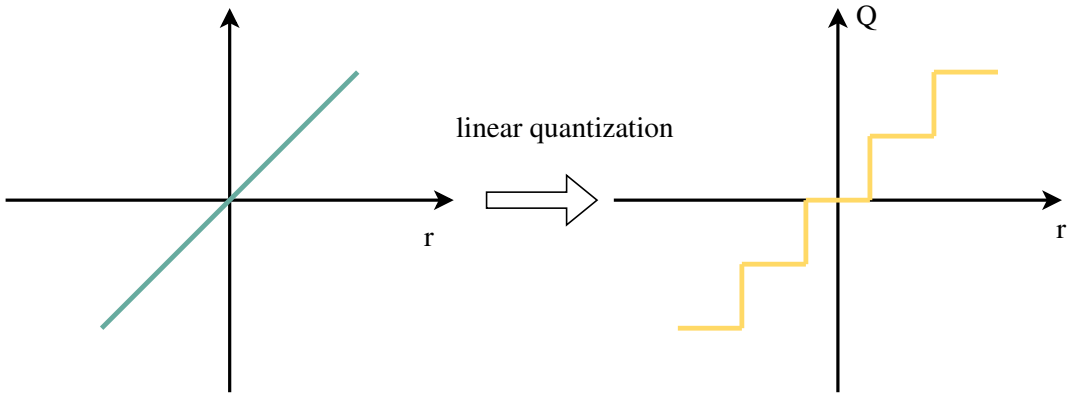


Figure 1.8: Linear quantization.

As shown in Figure 1.8, model parameters with similar numerical values are quantified to the same interval. It can be seen that neural network quantization is lossy, and how to minimize the loss of accuracy of the quantized neural network is a popular research topic. The benefits of neural network quantization are obvious. If we quantize 32 bits model parameters to 4 bits, the size of the neural network is reduced by a factor of four (e.g., from 40M to just 10M), and the computational effort is also greatly reduced.

As we can see from the Figure, parameters close to zero are quantized to zero. As we will show later, in today's deep neural networks, the weights close to zero account for most of the weights, and such a one-size-fits-all quantization is inappropriate.

Efficient Learning Methods

By changing the computational method of the neural network or the basic architecture of the neural network, the computational reduction can also be obtained. For example, a segmentation task requires many convolution operations. For modern encode-decode structure, there are often many 0-valued blocks in the output feature maps of each layer, and the convolution output of these 0-valued blocks is still 0-valued (Figure 1.9).

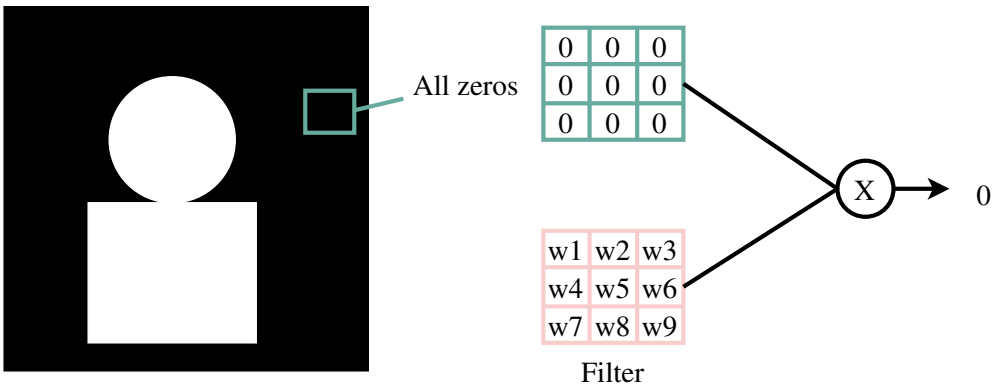


Figure 1.9: A special case that the convolution operation can be omitted.

Rather than convolving these blocks layer by layer, we can add 0-activation prediction to the forward and backward inference and the convolution computation can be omitted when the output of a convolu-

tion operation is predicted to be 0. By predicting the 0-value activation to reduce computation, Shomron et al. achieved over 30% savings in MAC operations on ResNet-18 [13].

Chapter 2

Related Works

2.1 Feature Extraction

Feature extraction is used to reduce the input complexity before the samples are fed into the neural network. Ali et al. used CNN model OverFeat [14] as a feature extractor on different recognition tasks and achieved superior results [15]. In practice, feature extraction and model compression could be combined to further reduce memory size. The following example shows a feature extraction experiment on MNIST which reduces the input size from 784 to 64.

2.1.1 An Feature Extraction Example on MNIST

Here we use the MNIST as an example owing to its simplicity. Handwritten digital images usually have clear contours and large portions of blank background which allows for considerable dimension reduction. Among several classifiers surveyed by Patel et al. [16], feature extraction combined with MLP (multi-layer perceptron) gives the highest recognition rate for handwritten digits. Four View Projection Profiles algorithm [17] with MLP, the best method in the survey, reaches a recognition rate of 98.73%. However, their samples are normalized to 100 by 100 sizes which are vastly more dimensions than MNIST samples. To fill the gap, we have made some modifications to the Four View Projection Profiles and the modified algorithm outputs four peak sums in each direction. The process is explained as follows:

- Divide each (28,28) matrix to 16 zones (each of 7 rows $\times$ 7 columns).
- Find the horizontally, vertically, left diagonal and right diagonal sums in each line of 16 zones.
- Store 4 peak values of each direction.
- Generate feature vectors of 64 elements.

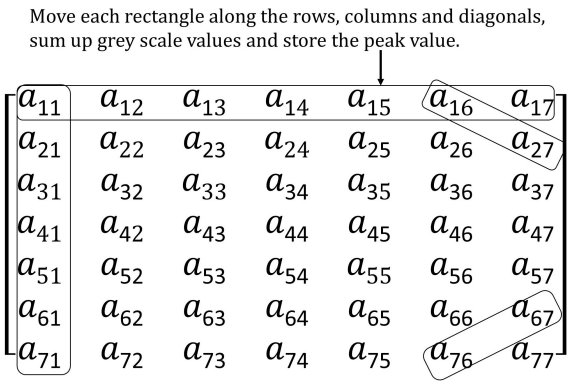


Figure 2.1: Process of the algorithm.

Figure 2.1 illustrates the specific process of the proposed algorithm. Compared with the Four View Projection Profiles algorithm, we firstly change one of the horizontal projections to left diagonal projection. Adding the left diagonal projection enables our algorithm to detect curves in four directions. Secondly, instead of generating an average of four sums in each zone, our algorithm outputs four peak sums as four different features. This step ensures more detailed information in each zone is retained.

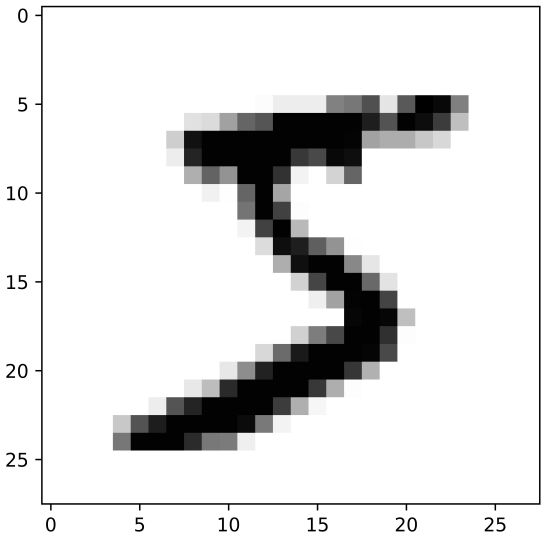


Figure 2.2: An MNIST sample.

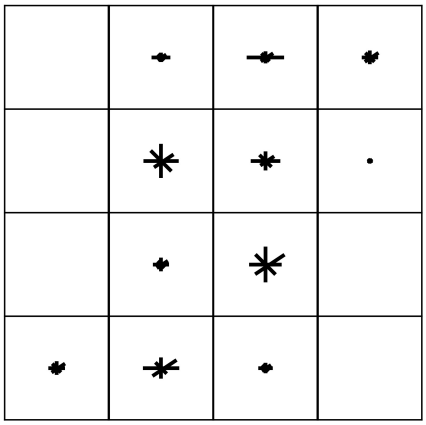


Figure 2.3: Visualization of a sample from feature vectors (16 zones).

Figure 2.2 and Figure 2.3 show the comparison between the original data sample and extracted feature vectors. Figure 2.2 is a sample from MNIST, its greyscale has been reversed to be better compared with Figure 2.3. In Figure 2.3, each zone contains 4 feature vectors. To better visualize it, different vector values are expressed through variations in length. In addition, we retained feature vectors' original orientation when drawing these line segments.

As can be seen from Figure 2.3, although we have compressed the original image to a great extent, the information of the original image is still well preserved. Moreover, background noise is largely taken away from the original sample. Although it might be hard to recognize with human eyes, neural networks should be able to abstract characteristics and differences between samples like these.

2.2 Toward Efficient Inference: Quantization and Additional Neural Networks

With the widespread use of deep learning, a growing number of scenarios require the deployment of neural networks to low-performance devices. Neural networks are increasingly being deployed on wearable devices and IoT devices [18]. For instance, on some wearable devices, neural networks might be used for sleep monitoring which requires efficiency and low energy consumption [19]. Multiplications between full-precision float point numbers are computationally expensive which hinders deployments of deep learning on low-performance devices. Many researches are devoted to optimizing the resource-intensive computations during training and forward inferences.

Neural Network Compression

To compress neural networks, Courbariaux et al. introduced BNN (binary neural network) [20, 21] which constrains all weights into +1 and -1. By retraining, BNN achieves nearly state-of-the-art accuracy on CIFAR-10, MNIST and SVHN though deteriorates drastically on ImageNet [22] dataset (22.1% of Top-5 accuracy decrease on GoogLeNet) [23]. Either Binary or Ternary (+1, 0, -1) neural networks, at cost of huge quantization noises, rely deeply on retraining [24].

Hubara et al. also extended their experiments using Miyashita et al.s logarithmic quantization algorithm which narrowed the accuracy gap to 8.1%. In another way, Deep Compression [25] clusters weights via K-means. Weights grouped into the same category will share the same quantized weight value. Specifically, Deep Compression quantizes convolution layers weights to 8 bits and fully connected layers weights to 5 bits. After retraining, they realized lossless accuracy. Combined with pruning and Huffman coding, Han et al. further reduced parameter size. Compared with Deep Compression which performs pruning and quantization in separate steps, CLIP-Q [26] carries out these 2 approaches in parallel during training. Googles team implements integer arithmetic only inference (see [27]) which quantizes all weights into 8 bits integers. Integer only inference is realized via affine mapping, and now it is the quantization scheme adopted by TensorFlow Lite [28].

Modifying the existing architecture to reduce parameter size is also studied. Iandola et al. created SqueezeNet [29] which achieves AlexNet-level accuracy with 50Œ fewer parameters. SqueezeNet decreases the number of input channels to 3Œ3 filters. Besides, it also replaces 3 Œ 3 filters with 1 Œ 1 filters. SimpleNet [30] is another case, Hasanpour et al. released their 13 layers convolutional network in 2016 outperforming deeper and heavier networks. Interestingly SimpleNet avoids using 1 Œ 1 filters in early layers to preserve locality information.

Either pruning or quantization might cause a huge influence on the final accuracy. To restore the original accuracy, retraining is usually performed after network compression. Some quantization or pruning strategies are applied during the training process [31–33] therefore no retraining is needed. The convergence of training using coarsely quantized weights was analysed by Li et al. [34]. On the other hand, direct quantization on pre-trained neural networks relies deeply on retraining. Typically, retraining algorithms still keep a copy of full-precision weights.

Sung et al.'s retraining method [24] applies weight update only to full-precision weights and then generates quantized weights. They also assert that this approach takes less time to converge. Other efforts such as INQ [32] quantizes networks incrementally. In each iteration, it quantizes only part of the network and adapts update to the left part. Although retraining algorithms can restore the original accuracy, they usually require big computational power since backpropagation is involved.

All compression methods mentioned above are able to improve the efficiency during inference. In this section, we focus on quantization methods. Further we elaborate on some commonly used quantization.

2.2.1 Neural Network Quantization

Logarithmic Quantization

Most operations in a neural network are matrix multiplications, which are basically dot products $y = \sum W^T x$. In order to save computational resources, this multiplication can also be performed in the log-domain [35]. As shown in Figure 2.4, computations in conventional neural networks require full-precision input which is memory-intensive. In addition, multiplication between full-precision numbers is expensive for hardware. Fortunately, in log domain the above problems can be largely relieved. Quantizing the input x to log domain can reduce the memory consumption. For instance, an input element 0.12 will be quantized to 2^{-3} and only the integer -3 will be kept in the memory. In this way, the original 32 bits inputs are quantized to integers with lower bitwidth. Also, the costly multiplication operations are transferred to bit shifts operations which are extremely hardware friendly.

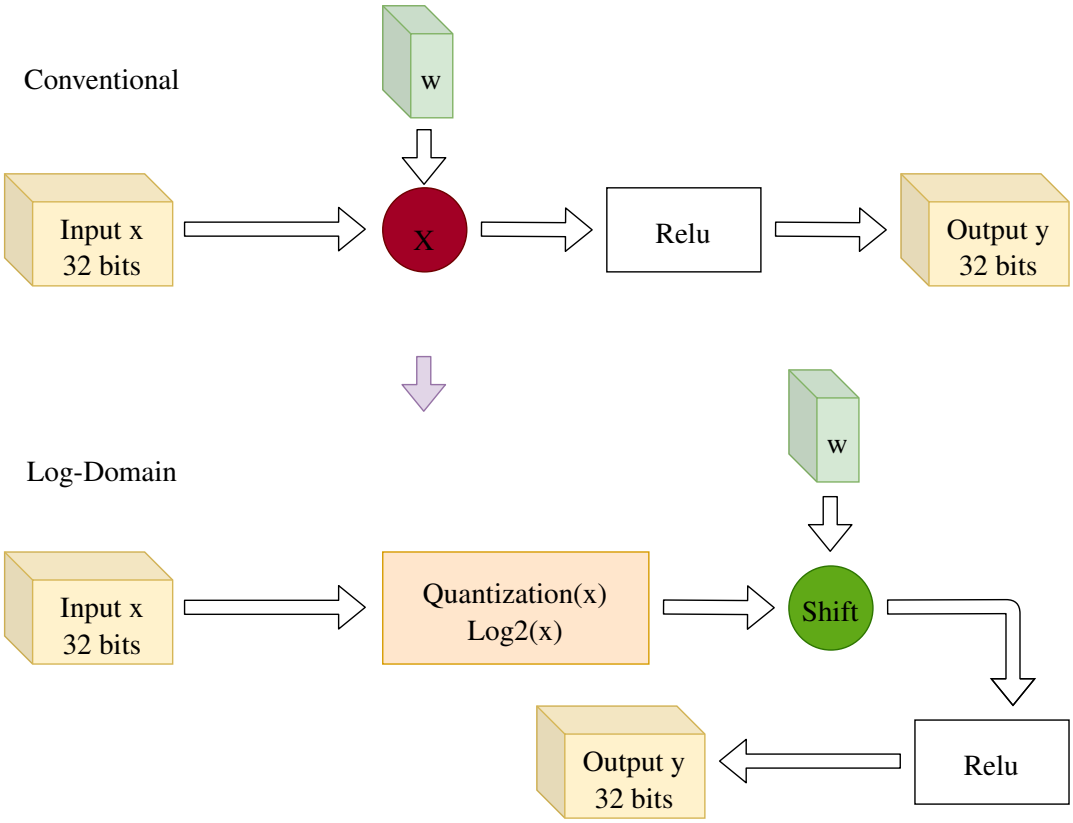


Figure 2.4: Logarithmic quantization is used to quantize model parameters to log-domain so that dot product between weights and activations can be performed via bit shift operations.

Miyashita et al. proposed two approaches to optimize neural network training [35]. One is to quantize layer inputs only so that all multiplications are performed through bit shifts (Equation 2.1). Another method is to quantize both inputs and weight parameters so that dot product can be obtained via integer additions (Equation 2.2).

$$\begin{aligned} w^T x &\simeq \sum_{i=1}^{n} w_i \times 2^{\tilde{x}_i} \\ &= \sum_{i=1}^{n} \text{Bitshift}\left(w_i, \tilde{x}_i\right) \end{aligned} \tag{2.1}$$

$$\begin{aligned} w^T x &\simeq \sum_{i=1}^{n} 2^{\text{Quantize }(\log_2(w_i))+ \text{ Quantize }(\log_2(x_i))} \\ &= \sum_{i=1}^{n} \text{Bitshift}\left(1, \tilde{w}_i + \tilde{x}_i\right) \end{aligned} \tag{2.2}$$

Miyashita et al. also used a FSR (full scale range) parameter to control the output scale during quantization. The complete quantization process can refer to Equation 2.3.

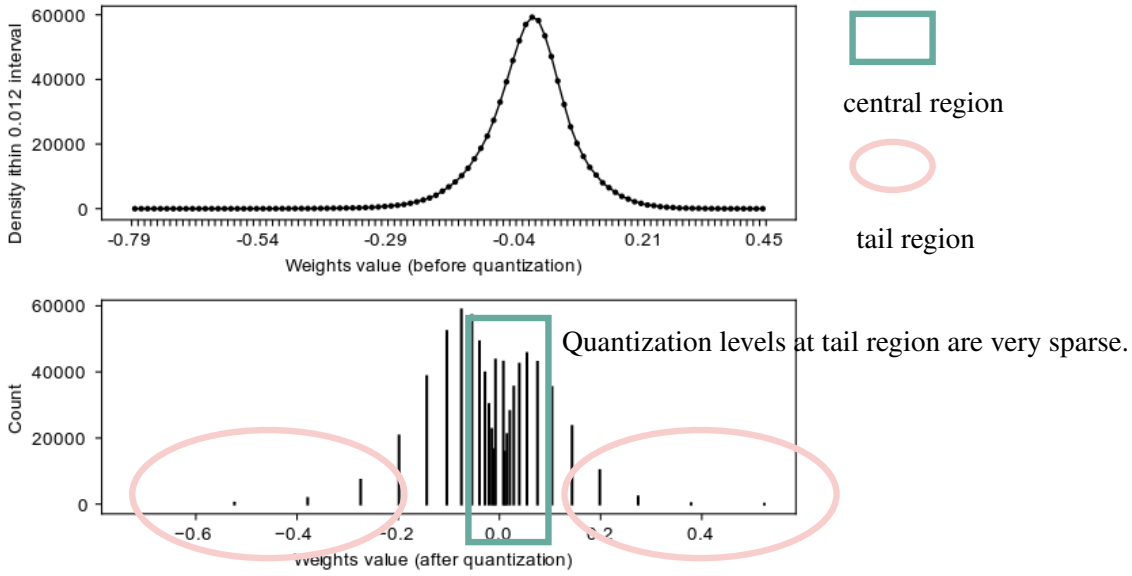


Figure 2.5: Sparsity of quantization levels at tail region (4 bits quantization with LogQuant). Note: the sparsity at tail region problem cannot be solved with increasing the quantization bitwidth.

$$\text{LogQuant}\ (x,\ \text{bitwidth, FSR}\ ) = \begin{cases} 0 & x = 0 \\ 2^{\tilde{x}} & \text{otherwise} \end{cases}$$

where

$$\tilde{x} = \text{Clip}\ \left(\ \text{Round}\ \left(\log_2(|x|)\right),\ \text{FSR}\ - 2^{\text{bitwidth}}, FSR\right) \tag{2.3}$$

$$\text{Clip}(x, \min, \max) = \begin{cases} 0 & x \leq \min \\ \max - 1 & x \geq \max \\ x & \text{otherwise} \end{cases}$$

Paper [35] showed that with 3 bits quantization, the final accuracy can be close to the original on AlexNet and VGG16. Besides, the model size can be largely reduced thanks to the logarithmic quantization. Elhoushi et al. [36] implemented a shift-based GPU kernel that trains a neural network by shift from the start. DeepShift removes all multiplications during training. Instead, only shift and sign reverse operations are used. They reported about 25% reduction in latency time on modern GPUs. With 6-bits shift format, training ResNet-18 from scratch using DeepShift on ILSVRC-2012 reported a 4.48% accuracy loss.

Although LogQuant reduces the computational effort and the size of the model, there is a problem that the method does not overcome, that is, after quantizing the neural network with this method, the accuracy of the model decreases significantly and thus needs to be retrained to make up for the accuracy.

Here we briefly analyze the reason why the method decreases a lot in accuracy. We used the LogQuant to quantize a hidden layer of a simple fully connected neural network trained on MNIST. The quantized results are illustrated in Figure 2.5. As shown in the Figure, the problem with LogQuant is that neural network quantization with this method will be more focused on smaller model parameters and very sparse in larger parameter regions (tail regions). We will discuss the importance of large parameters in detail in Chapter 3, and we can find experimentally that large parameters are much more important

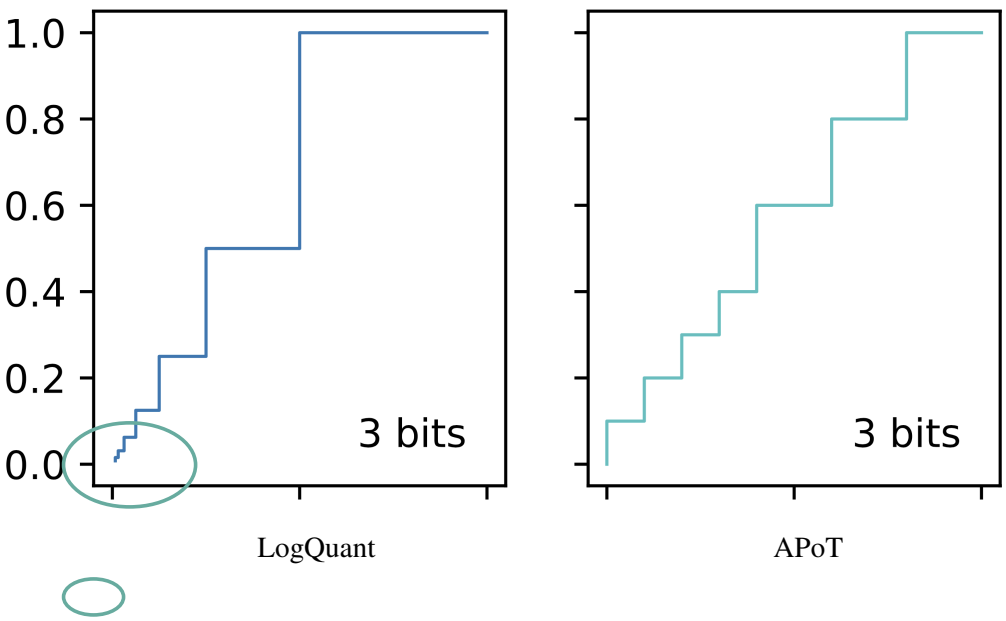


Figure 2.6: 3-bits quantization comparison (left: LogQuant, right: APoT). Compared with LogQuant, APoT is more balanced and assigns more quantization levels to weights on tail regions, therefore can obtain better accuracy without retraining.

than small model parameters. This tail region sparse quantization method ignores the importance of the large parameters and instead enhances the resolution of the small parameter part.

Li et al. adopted a method to solving the sparsity on edges (larger weights) after logarithmic quantization. In [37] the APoT (additive powers of two quantization) quantization approach was proposed. They referred to the phenomenon that larger weights cannot benefit from increasing the quantization bit-width as *rigid resolution*. As Equation 2.4 indicates, APoT introduces extra additive terms for logarithmic quantization. Note γ is a scaling factor and α denotes the maximum of the quantization.

$$\mathcal{Q}^a(\alpha, kn) = \gamma \times \left\{ \sum_{i=0}^{n-1} p_i \right\} \text{ where } p_i \in \left\{ 0, \frac{1}{2^i}, \frac{1}{2^{i+n}}, \ldots, \frac{1}{2^{i+(2^k-2)n}} \right\} \tag{2.4}$$

As shown in Figure 2.6, APoT can resolve the sparsity on edges and, at the same time, preserve the merit of logarithmic quantization which is the efficient shift based computation. Although a scaling factor is introduced and can cause the decrease in efficiency, the multiplication with the factor is performed only once after the multiply-accumulate step. Li et al. reported 22% faster computation compared with uniform quantization.

Linear Quantization

Linear quantization is another quantization research direction that produces neural networks with efficient inference. The simplest linear quantization method is ULQ (uniform linear quantization). ULQ linearly maps full-precision parameters into low-precision representations. However, direct ULQ has been reported to cause decreases in accuracy. Baluja et al. [38] presented an approach that adaptively

cluster the weight throughout the training process. Their quantization algorithm clusters weights in every 1000 steps during training with a one dimension K-means method [39].

To achieve inference without multiplication operations of float point numbers, Baluja et al. also quantized activations of each layer so that they can make a multiplication table. In this way all possible entries are pre-computed therefore can be stored in memory. The multiplication operations in a neural networks are then transferred to table look-up operations.

Fang et al. [40] proposed PWLQ (piecewise linear quantization) that can compress the neural network without retraining. Previous linear quantization methods suffer large accuracy losses, therefore a post-training linear quantization method is desirable. To overcome this drawback, Fang et al. designed an algorithm that aims to quantize the neural network with minimal quantization error.

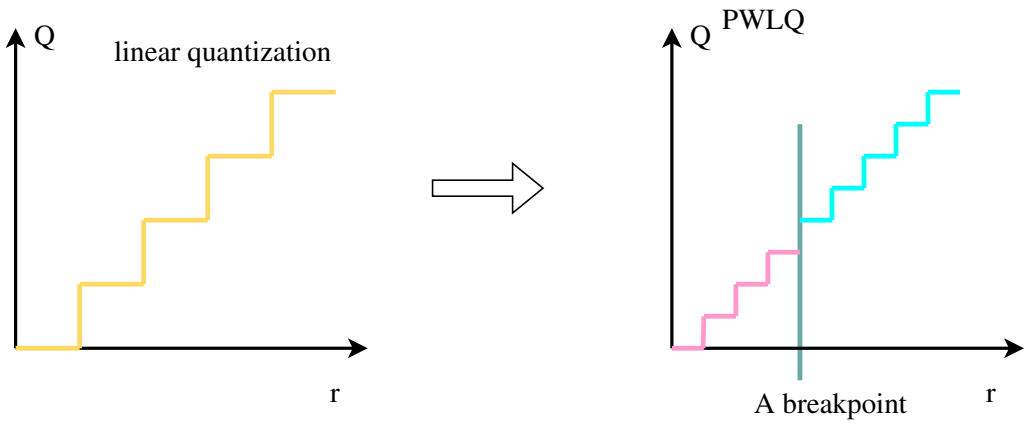


Figure 2.7: Comparison of ULQ and PWLQ quantization. In PWLQ the quantization range is divided into several pieces with different quantization precision.

A uniform quantization [27, 41] method is given in Equation 2.5, where r is the parameter to be quantized, [a,b] is the quantization range, n is the number of quantization levels. s is a scaling factor and [ů] denotes rounding to the nearest integer. Fang et al. think that ULQ always divide the quantization range evenly despite the parameter distribution. Since weights and activations of pre-trained neural networks are normally Guassian or Laplacian distributed [25,42], direct ULQ cannot preserve the original accuracy because of its large quantization error.

$$\begin{aligned} \text{clamp}(r; a, b) &:= \min(\max(x, a), b) \\ s(a, b, n) &:= \frac{b-a}{n-1} \\ q(r; a, b, n) &:= \left[\frac{\text{clamp}(r; a, b) - a}{s(a, b, n)}\right] s(a, b, n) + a, \end{aligned} \tag{2.5}$$

Instead, PWLQ is dedicated to minimizing the quantization error. PWLQ divides the the quantization range to two non-overlapping part, which consists of the central region and the high-magnitude region. A breakpoint k is set to divide the quantization range. Say if the quantization range is $[-q, q]$ then the central region is set to $[-k, k]$ and the sparse region is set to $[q, k) \cup (k, q]$. PWLQ divides the quantization range to four pieces and within each piece, for B-bits quantization, B-1 bits Uniform quantization is performed. PWLQ quantization results can be seen from Figure 3.15. The value of k controls the quantization precision of each region. In [40] the k is limited to $0 < k < \frac{q}{2}$ so that central region

can have higher approximation precision. Fang et al. also gave a proof that the optimal k^* exists by minimizing the quantization error. In practice $k*/q = ln(0.8614q+0.6079)$ was adopted for normalized Gaussian. PWLQ was experimented on several computer vision tasks including ImageNet [22], semantic segmentation and so forth. Experimental results confirm that PWLQ can obtain better accuracy without retraining.

Another problem of linear quantization is that it usually has a min-max limitation. As Equation 2.5 shows, the outlier weights are clipped to minimal or maximal values that determined in advance. Zhao et al. [43] used an outlier channel splitting (OCS) method to overcome this problem. Their OCS method can be explained by Equation 2.6. Note Equation 2.6 is a rewritten version of Equation 1.1. In their method the outlier weights are split to multiple duplicates. Therefore the width of certain layers is enlarged but the quantization error can be reduced. A graphical explanation is shown in Figure 2.8. The outlier neurons are duplicated thus the activations of these neurons can be reduced in half.

$$\begin{aligned} \mathbf{y}_j &= \sum_{i=1}^{m-1} \mathbf{x}_i * \mathbf{W}_{ij} + \left(\mathbf{x}_m * \frac{\mathbf{W}_{mj}}{2}\right) + \left(\mathbf{x}_m * \frac{\mathbf{W}_{mj}}{2}\right) \\ \mathbf{y}_j &= \sum_{i=1}^{m-1} \mathbf{x}_i * \mathbf{W}_{ij} + \left(\frac{\mathbf{x}_m}{2} * \mathbf{W}_{mj}\right) + \left(\frac{\mathbf{x}_m}{2} * \mathbf{W}_{mj}\right) \end{aligned} \tag{2.6}$$

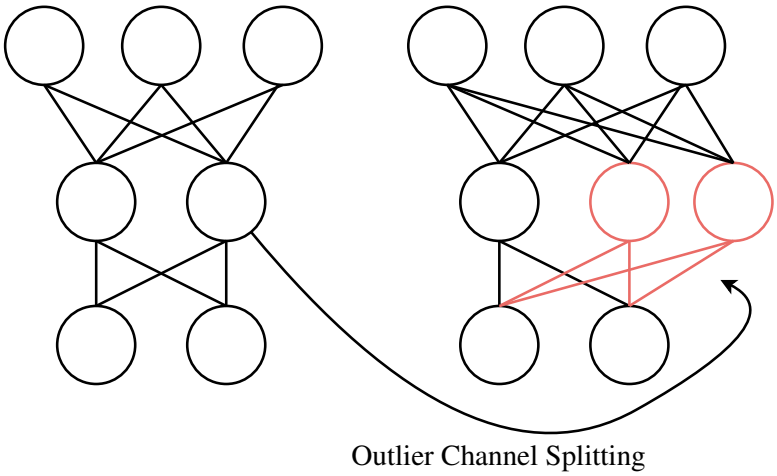


Figure 2.8: Outlier Channel splitting. The outlier activation is duplicated so that the layer width is enlarged but direct clipping is avoided.

Zhou et al. demonstrated that OCS can effectively reduce the quantization error and achieve better performance at the cost of wider layers. Since the clipping methodology is also used in NVIDIA TensorRT [44], this method can be easily adopted on commodity GPUs.

Binary Neural Network (BNN)

BNN [45] is an extreme case of neural network model compression. The method constrain all weights and activations to +1 and -1. BNNs can achieve good accuracy on small datasets such as MNIST and CIFAR-10. During the training of BNNs, a deterministic binarization function is adopted to transform the real-valued variables to +1 or -1 (see equation 2.7). It is worth mentioning that apart from the deterministic binarization function, Courbariaux et al. also developed a stochastic binarization function. The

stochastic binarization function transforms weights and activations to +1 or -1 stochastically. However, stochastic binarization is not easy to implement therefore usually not adopted.

$$x^b = Sign\,(x) = \begin{cases} +1 & if \quad x > 0, \\ -1 & otherwise \end{cases} \tag{2.7}$$

Although weights and activation are constrained to +1 and -1, real-valued gradients are kept for parameter updating. The network would not be able to converge if gradients are also binarized. Another problem is the backpropagation of gradients. The backpropagation algorithm utilizes chain rule to assign error to all connections. However the derivative of the Sign function is always zero which makes it incompatible with the backpropagation algorithm. To solve this, Htanh function is used during backpropagation instead of Sign function (Figure 2.9). Suppose we know a binarized weight which is $q = Sign(r)$ and the derivative for q is $\frac{\partial C}{\partial q}$. The derivative to r is then given by $g_r = Htanh(g_q)$.

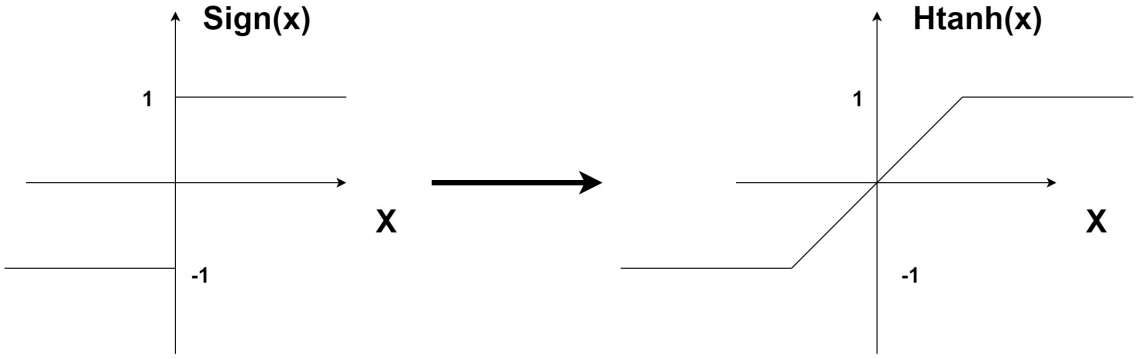


Figure 2.9: Htanh function for backpropagation instead of Sign function.

Parameters in BNN could be updated in a similar way to normal real-value neural networks, only an extra clip step is required to ensure the updated weights are still constrained to +1 and -1. Though the convergence speed of BNNs is slower than normal DNNs, they can output similar accuracy on small datasets. The drawbacks BNNs are listed as follows:

- To implement BNN, one needs to train a neural network from the beginning. Pre-trained networks are useless for BNN.
- BNN fails to converge on complex tasks such as Imagenet.
- The training speed of BNN is very slow.

Bi-Real Net

As we mentioned in the last section, BNN suffers accuracy loss on complex dataset like Imagenet. Bi-Real Net [46] proposed by Liu et al. is a recent research aiming to solve the deterioration of the final accuracy. Figure 2.10 illustrates a forward pass from inputs to binarized activations. Although weight parameters are binarized, activations after a convolution layer are real values. These activations in BNN are then transformed by Sign functions therefore again binarized. However, the real activations contain more information thus binarized activation could cause the decrease of final accuracy.

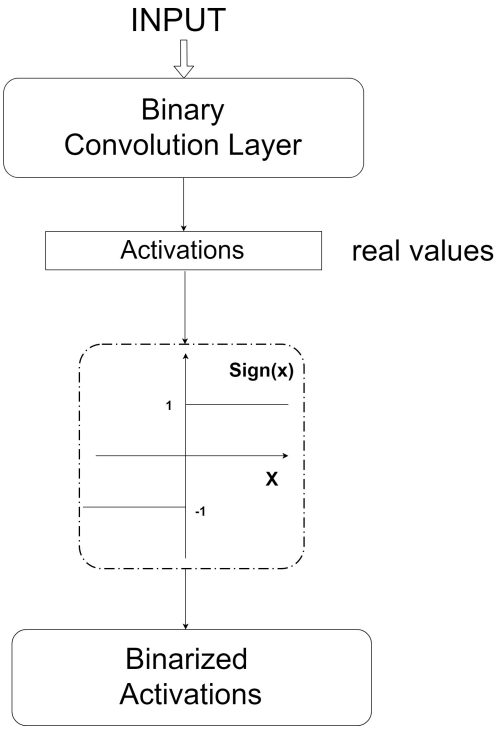


Figure 2.10: A typical forward pass in BNNs.

To preserve the information, Bi-Real Net adds shortcuts between different layers (Figure 2.11). Real activations are connected via an addition operator which brings no additional memory cost. The addition operation is element-wise, therefore this method is able to increase the representational capability of the whole network. Besides, Bi-Real Net also proposes some new methods for training. The paper claims these methods are more suitable for training binary neural networks. These methods are summarized as the following:

- Using a differentiable piecewise polynomial function instead of the Htanh function. The derivative of the function is triangle-shaped rather than rectangular shaped derivative of the Htanh function.
- When using backpropagation to update weights in BNNs, the gradient for each weight is only related to the sign of the current real weight. To compute more efficient gradients, the sign function used for weight updating is transformed to a magnitude-aware version.
- Using Htanh function in pre-training step instead of ReLU.

Compared to BNNs, Bi-Real Net can achieve up to 10% higher top-1 accuracy. Though outperforms BNNs, its accuracy still lags behind the full precision neural networks by more than 10%. And for all neural networks that use only binary or ternary weights, network retraining could be time and resource consuming.

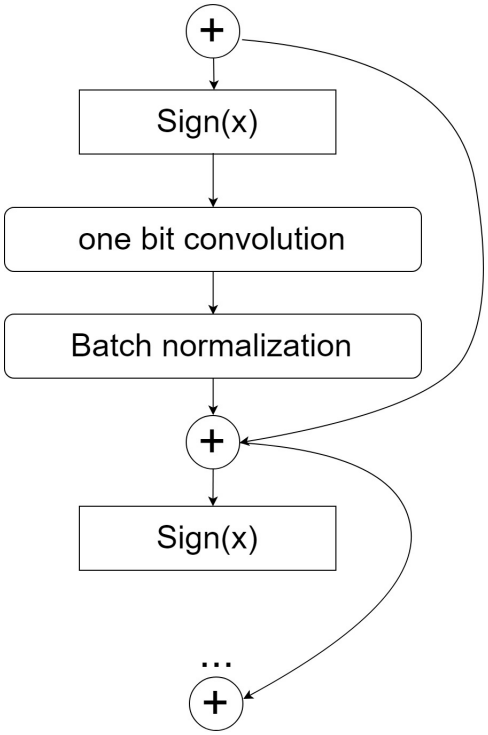


Figure 2.11: Adding shortcuts to increase the representational capability.

Deep Compression

Deep Compression consists of three stages: pruning, quantization and Huffman coding. In the pruning stage, small weights are removed since these weights are unimportant (less contribution during the forward pass). After pruning, the rest weights are clustered via K-means. And finally, the quantized weights are Huffman encoded.

In the pruning step, weights are processed through equation 2.8 and this step will output sparse matrices. To store these sparse matrices, CSC (compressed sparse column) or CSR (compressed sparse row) is used. The author made some modifications to the original CSC or CSR method. Instead of storing absolute positions, offset is used to reduce the bitwidth of indices.

$$Pruning\,(x) = \begin{cases} 0 & |x| \leq t \\ x & |x| > t \end{cases} \tag{2.8}$$

The quantization step clusters weights in the same layer into shared values. For AlexNet, Deep Compression is able to quantize weights in convolutional layers to 8 bits and fully connected layers to 5 bits. Only indices and a copy of shared value are required to be stored. For K-means method, initialization of the shared weights is important. Linear initialization can better preserve the weights distribution therefore is used to initialize the shared weights. Quantization partition weights $\{w_1, w_2, ..., w_n\}$ into clusters $\{c_1, c_2, ..., c_k\}$.

$$argmin_c \sum_{i=1}^{k} \sum_{w \in c_i} |w - c_i|^2 \tag{2.9}$$

BNN+

BNN+ [47] was released recently (on December 31th, 2018). Darabi et al. made several improvements to BNN. Firstly, they change the activation function to equation 2.10. In equation 2.10, the σ is sigmoid function and β is a hyper parameter controlling how fast the activation function asymptotes to -1 and +1. Equation 2.11 is used to compute the gradient.

$$\mathrm{SS}_\beta(x) = 2\sigma(\beta x)[1 + \beta x\{1 - \sigma(\beta x)\}] - 1 \tag{2.10}$$

$$\frac{d\mathrm{SS}_\beta(x)}{dx} = \frac{\beta\left\{2 - \beta x \tanh\left(\frac{\beta x}{2}\right)\right\}}{1 + \cosh(\beta x)} \tag{2.11}$$

Equation 2.10 offers the freedom to adjust the location where the gradients start saturating. The merit using this method to train is that the weights are still differentiable near -1 and +1. BNN poposed by Courbariaux et al. adopts Htanh function than the sign function for avoiding zero gradients. However, using Htanh functioin will cause gradients be disappear when the value is not in the [-1,1] interval (Figure 2.12).

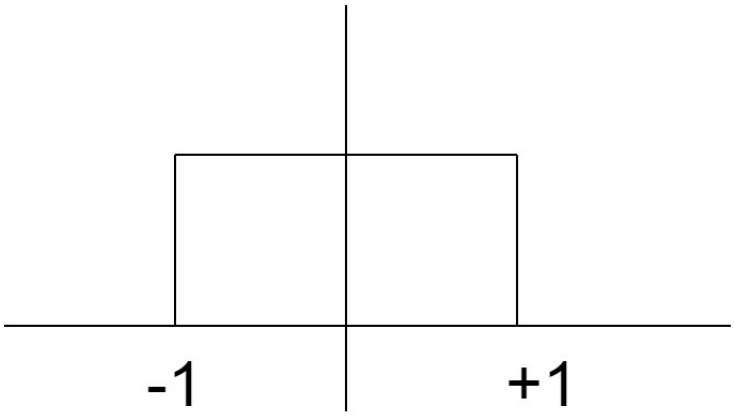


Figure 2.12: Derivatives of Htanh function. During backpropagation, gradients are disappear on two sides.

Activation function proposed in BNN+ is the solution to the problem above. Figure 2.13 presents the gradients of the proposed activation equation. It is claimed to be a closer approximation to sign function.

Apart from modification on activation function, BNN+ also introduces a regularization term. This regularization term in BNN+ is used to encourage the wegiths around -1 and +1. In the paper they used Manhatan and Euclidean regularization functions and the loss function is then given in equation 2.12.

$$J(W, b) = L(W, b) + \lambda \sum_l R\left(W_l, \alpha_l\right) \tag{2.12}$$

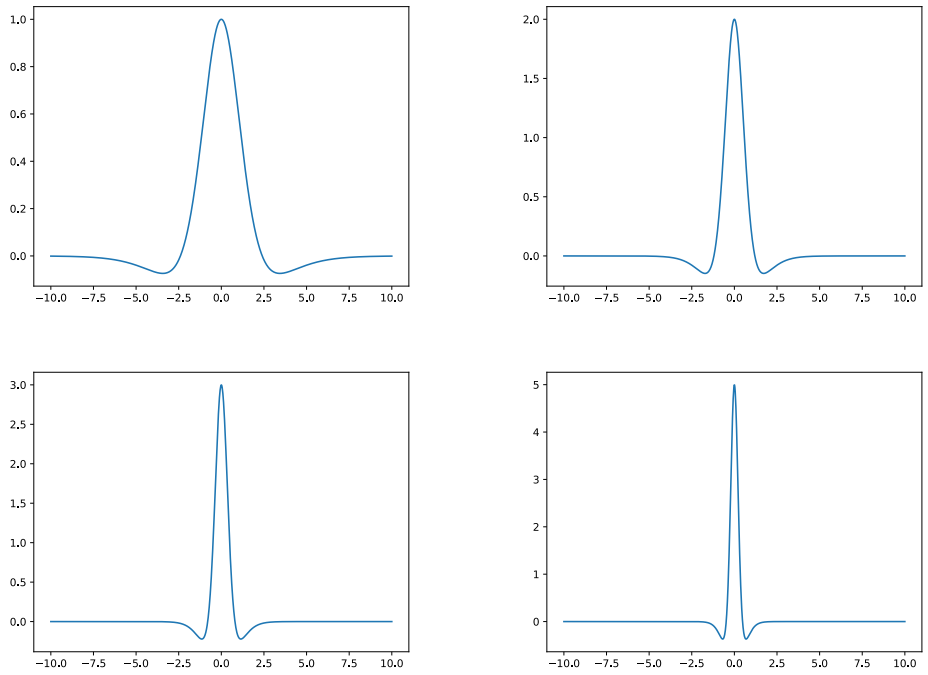


Figure 2.13: Derivative of the activation function with beta set for different values. From left to right and top to bottom, the β is set to 1, 2, 3 and 5.

In practice, these two main modifications do contribute to better accuracy. But for large datasets such as Imagenet, the improvements are limited. BNN+ performs better than BNN with about 5% accuracy increase on Imagenet. However, there is still a 10% accuracy gap between BNN+ and the full precision counterpart.

2.2.2 Hardware Optimization for Hyperbolic Functions with CORDIC

Hardware optimization for hyperbolic activation functions has been reported in previous studies. Although the chip area and power consumption are reduced, related studies apply hyperbolic functions only to activation computations, which limits the optimization gains. The CORDIC algorithm [48] is considered an efficient method for computing hyperbolic activations. Here, we briefly introduce the CORDIC algorithm, which is widely used in microcontrollers and FPGAs. Initially proposed in 1959, the CORDIC algorithm is an efficient way to calculate hyperbolic and trigonometric functions. A CORDIC module computes the trigonometric value of a certain angle through accumulative rotations of the preset base angles. It decomposes a certain angle into discrete base angles that are pre-stored in a look-up table. Note that all the base angles are chosen as powers of 2; thus, iterative rotations of an angle are computed through shift operations. The CORDIC algorithm then iteratively applies accumulative rotations of the base angles until the required precision is met.

Many accelerated CORDIC algorithms have been proposed over the years. For example, Jaime et al. proposed a scale-free CORDIC engine that can reduce the chip area and power consumption by 36% [49]. Mokhtar et al. proposed a sine and cosine CORDIC compute engine that requires only seven iterations for 16-bit precision [50]. An improved rotation strategy was also studied in [51], which achieved 32

precision with only approximately five iterations on average.

Chen et al. used a CORDIC engine to compute an application-specific algorithm [52], i.e., an independent component analysis (ICA), which is used for the direct separation of a number of mixed signals. In their research, only a part of the updating scalar tanh(x) was computed using the CORDIC engine. The method still relies on a comparatively large number of multipliers. A similar approach was adopted by Tiwari et al., who used CORDIC to compute activations in neural networks [53].

Another direction to take advantage of hyperbolic functions is by applying them to specific computation tasks. Heidarpur et al. utilize the CORDIC engine to compute modified Izhikevich neurons [54]. Similarly, Hao et al. realized an efficient implementation of a cerebellar Purkinje model [55].

To the best of our knowledge, there is still a lack of training algorithms that can remove the usage of multipliers on deep neural networks during the training stage.

2.2.3 Addition based Neural Networks

Introduction

AdderNet is recently revealed by Huawei Noah's Ark Lab [56]. In this research, multiplications are replaced by addition operations in the forward pass. Chen et al. took a different way of thinking. According to the paper, the function of convolution operations is feature extraction. Kernels in the convolution stage can extract various features during training. In the multiplication convolution stage, the output of each kernel functions as a similarity measurement.

As finding the correlation between two vectors, convolution operations can also be seen as the inner product operation of filter and input vectors which are similar to correlation computation. When visualizing the convolution kernels, usually they are divided by their angles. AdderNets utilize l1-norm to differentiate different kernels therefore visualization of features for AdderNets is mainly expressed by distances of the center points.

As shown in Figure 2.14, the trained network's filters are clearly differentiated by their Euclidean distances. This result suggests that the l1 norm can serve as a similarity measurement between inputs and convolution kernels.

Training of AdderNet.

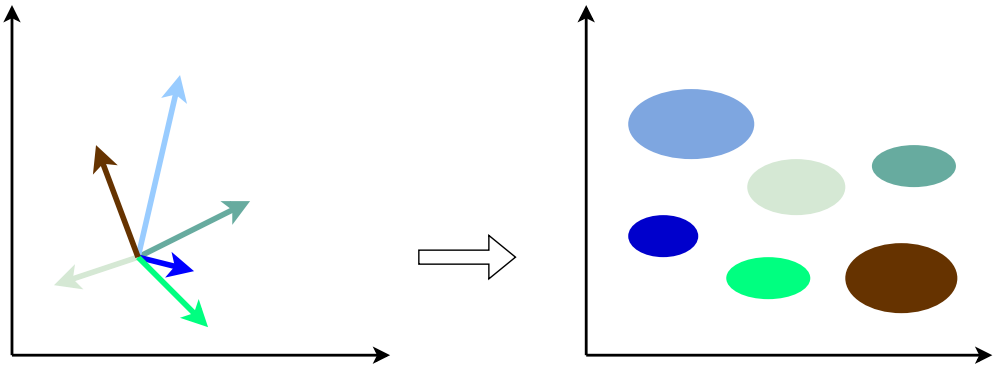
Although the name is AdderNet, it actually uses subtraction in the forward pass. As indicated by Equation 2.13, the l1 distance of the kernels and inputs of each layer is computed.

$$Y(m,n,t) = -\sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}} |X(m+i,n+j,k) - F(i,j,k,t)| \tag{2.13}$$

During the backpropagation stage, some tricks are made for promoting the learning. Rather than compute the gradient directly as the Equation 2.14 indicates, they use Equation 2.15 and 2.16 instead. This is because the result of Equation 2.14 can never obtain the steepest gradient.

$$\frac{\partial Y(m,n,t)}{\partial F(i,j,k,t)} = \operatorname{sgn}(X(m+i,n+j,k) - F(i,j,k,t)) \tag{2.14}$$

$$\frac{\partial Y(m,n,t)}{\partial F(i,j,k,t)} = X(m+i,n+j,k) - F(i,j,k,t) \tag{2.15}$$



(a) (b)

(a) Features in CNNs are differentiated by their angles.
(b) Features in AdderNets are differentiated by their central points.

Figure 2.14: Visualization of features in AdderNets and CNNs. Unlike conventional CNNs that its kernels are differentiated by their angles, AdderNets' kernel are divided by their center points. This phenomenon suggests that l1-norm between inputs and filters might be able to server as a inference method.

$$\frac{\partial Y(m,n,t)}{\partial X(m+i,n+j,k)} = \mathrm{HT}(F(i,j,k,t) - X(m+i,n+j,k)) \tag{2.16}$$

where the hard tangent function is:

$$\mathrm{HT}(x) = \begin{cases} x & \text{if} \quad -1 < x < 1 \\ 1 & x > 1 \\ -1 & x < -1 \end{cases} \tag{2.17}$$

Since the output of Equation 2.13 is always negative, batch normalization is introduced after each addition layer [57]. Also, the variances of layers in normal CNN and AdderNet are largely different. Equation 2.18 and 2.19 give estimations of variance in normal CNN and AdderNet. In practice, AdderNet layers tend to produce much larger variances. These larger variances will cause the failure of back-propagation since the gradients are much smaller.

$$\begin{aligned} \mathrm{Var}\left[Y_{CNN}\right] &= \sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}} \mathrm{Var}[X \times F] \\ &= d^2 c_{in}\, \mathrm{Var}[X]\, \mathrm{Var}[F]. \end{aligned} \tag{2.18}$$

$$\begin{aligned} \mathrm{Var}\left[Y_{\text{Adder } Net}\right] &= \sum_{i=0}^{d}\sum_{j=0}^{d}\sum_{k=0}^{c_{in}} \mathrm{Var}[|X - F|] \\ &= \left(1 - \frac{2}{\pi}\right) d^2 c_{\text{in}}\, (\mathrm{Var}[X] + \mathrm{Var}[F]), \end{aligned} \tag{2.19}$$

To solve the small gradients problem, an adaptive learning rate method was introduced in the paper. Equation 2.20 shows how to compute the adaptive learning rate of each layer. Note k denotes the convolutional kernel size and $\Delta L\left(F_l\right)$ is the gradient of the corresponding kernel. η is a hyper parameter to adjust the final learning rate. In the source code they offered, the η was set to 0.25 [3].

$$\alpha_l = \frac{\eta\sqrt{k}}{\|\Delta L\left(F_l\right)\|_2} \tag{2.20}$$

The core component of AdderNet source code is as follows. Note this version of the source code has not been optimized for CuDNN [58] thus the training speed is slow compared to normal CNN and requires much larger memory. Although multiplication in the forward pass is avoided,the backward pass is more complicated than conventional CNNs thus more computation expensive.

```
1
2  class adder(Function):
3      @staticmethod
4      def forward(ctx, W_col, X_col):
5          ctx.save_for_backward(W_col,X_col)
6          output = -(W_col.unsqueeze(2)-X_col.unsqueeze(0)).abs().sum(1)
7          return output
8
9      @staticmethod
10     def backward(ctx,grad_output):
11         W_col,X_col = ctx.saved_tensors
12         grad_W_col = ((X_col.unsqueeze(0)-W_col.unsqueeze(2))*grad_output.unsqueeze
       (1)).sum(2)
13         grad_W_col = grad_W_col/grad_W_col.norm(p=2).clamp(min=1e-12)*math.sqrt(
       W_col.size(1)*W_col.size(0))/5
14         grad_X_col = (-(X_col.unsqueeze(0)-W_col.unsqueeze(2)).clamp(-1,1)*
       grad_output.unsqueeze(1)).sum(0)
15
16         return grad_W_col, grad_X_col
```

Listing 2.1: forward and backward pass of the AdderNet.

Multiplication as Integer Addition Method

The introduction of an addition based neural network is a very new concept, so there are not many updates at present. Nevertheless, some methods have been proposed to improved addition based neural networks.

Tsuguo Mogami used an approximate multiplication method by using an "integer-add instruction" [1]. A similar method had been used before [59] but not for training a neural network. Mogami took advantage of the data representation in IEEE 754 standard that all floating point numbers are actually stored in a logarithmic form. As shown in Figure 2.15, a decimal is divided into three parts consist of sign, exponent and mantissa. The exponent part is the logarithm of the original number and the mantissa keeps the decimal part. Multiplication between decimals always includes operations of the exponents and mantissa. Mogami omits the operations mantissa and focus on only operations of exponent part therefore the multiplication could be accelerated in principle. The paper reported that this way of approximation causes 12.5% error at most and it functions as Mitchells approximate multiplication [60]. The method was tested with Chainer [61, 62] and the training curve showed no difference with the multiplication counterpart.

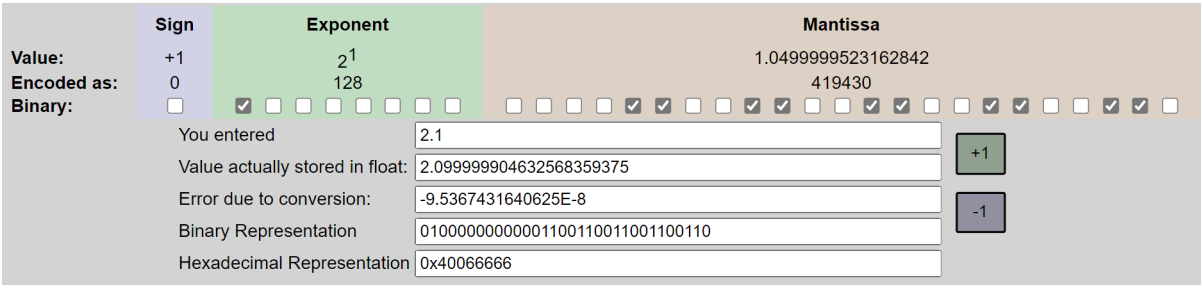


Figure 2.15: An example of a floating point number representation in IEEE 754 form. All floating points are represented by three part: sign, exponent and mantissa. In [1], multiplications between mantissas are omitted.

The pseudo-code is as follows. Note on modern GPU the speed of this method cannot surpass that of normal multiplications. Acceleration on ASICs requires future researches. As for accuracy, on ResNet-50, there was no deviation from the author's report.

```
float int mul(const float a, const float b){
int c = *(int*)&a + *(int*)&b - 0x3f800000;
return *(float*)&c;
}
```

Listing 2.2: Pseudo-code of [1] with mantissa multiplications omitted.

ShiftAddNet

Although AdderNet can partly remove multiplication operations in the forward pass, it still largely relies on multiplications during the backpropagation (see Listing 2.1). How to reduce the use of multiplication operations during backpropagation had become a problem. You et al. [63] released their shift based addition neural network training method in late 2020. In this work, they reported that their method can reduce the energy cost of over 80% on hardware DNN training and inference.

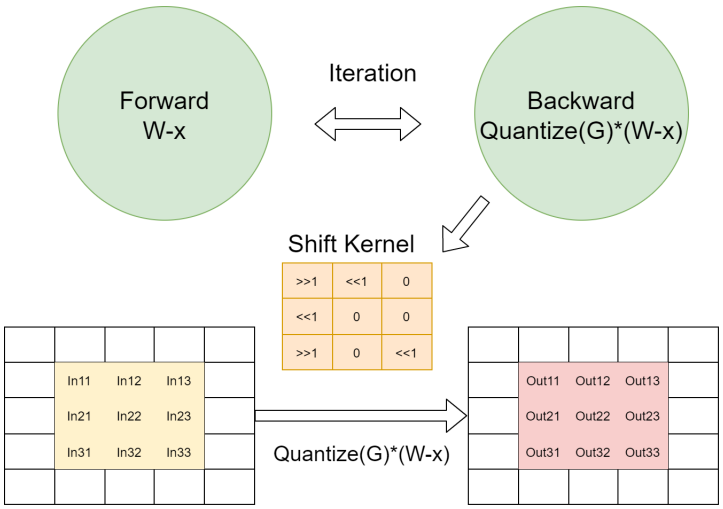


Figure 2.16: Backpropagation and inference of ShiftAddNet. During the backward pass, all gradients are quantized to powers of two, therefore the multiplication operations are transferred to bit shifts.

As shown in Figure 2.20, ShiftAddNet combines additional convolution and quantization techniques. During the forward inference, ShiftAddNer works exactly as the AdderNet. When the error signals are propagated backward, ShiftAddNet firstly quantizes all gradients to powers of 2. In this way all multiplications between kernels and layer outputs are transferred to bit shift operations which are hardware-friendly.

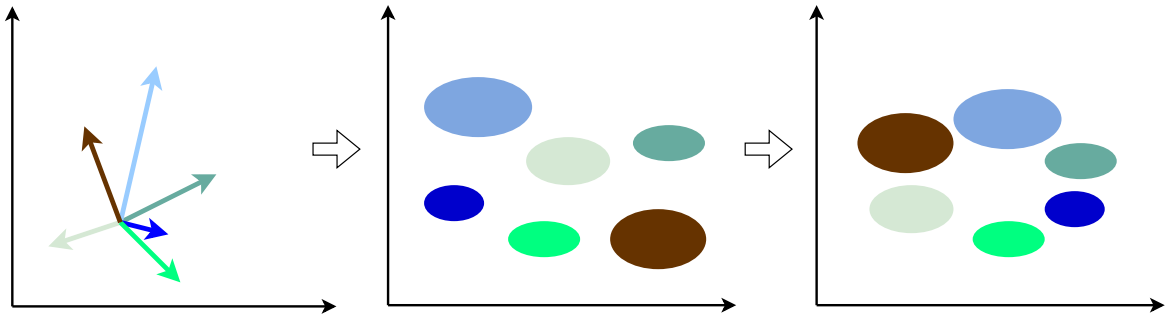
When compared with pure shift based neural networks such as DeepShift [36], ShiftAddNet can achieve much better accuracy with very little more energy consumption. Wang et al. designed a hardware kernel for ShiftAddNet [64] recently. In their work, they presented a hardware efficient shared-scaling-factor quantization method for AdderNets. As the result they achieved 81% resource saving on FPGAs with adder kernels than multiplication kernels.

```
class adder2d(nn.Module):
    def __init__(self,input_channel,output_channel,kernel_size, stride=1, padding=0,
     bias = False):
        super(adder2d, self).__init__()
        self.stride = stride
        self.padding = padding
        self.input_channel = input_channel
        self.output_channel = output_channel
        self.kernel_size = kernel_size
        self.adder = torch.nn.Parameter(nn.init.normal_(torch.randn(output_channel,
    input_channel,kernel_size,kernel_size)))

        # quantization
        shift = self.adder.abs().log2().round()
        shift = 2**shift
        shift[self.adder<0] = shift[self.adder<0] * -1
        self.adder = torch.nn.Parameter(shift)

        self.bias = bias
        if bias:
            self.b = torch.nn.Parameter(nn.init.uniform_(torch.zeros(output_channel)
    ))

    def forward(self, x):
        output = adder2d_function(x,self.adder, self.stride, self.padding)
        if self.bias:
            output += self.b.unsqueeze(0).unsqueeze(2).unsqueeze(3)

        return output
```

Listing 2.3: A sample code for shift-based AdderNet.

Knowledge Distillation

Knowledge distillation was used to bring up the accuracy of AdderNet [65]. It is verified that knowledge distillation can eliminate the accuracy drop compared with CNN for AdderNet. Here we first introduce some basics of knowledge distillation. Knowledge distillation can effectively learn a smaller student model from a giant teacher model [66]. In this way the large model can be compressed. As



(a) (b) (b)

(a) Features in CNNs are differentiated by their angles.
(b) Features in AdderNets are differentiated by their central points.
(c) Distll the angle differentiation based knowledge to distance based knowledge.

Figure 2.17: Progressive Distillation for Adder Neural Networks. A visualization of filters.

showed in 2.18, the knowledge that large model learned can be distilled to a smaller network. As for distillation strategies, there are many different types which play important roles.

In practice, first, we train the teacher network until it achieves the best accuracy. Then as indicated in Equation 2.21, we compute a KD loss in order to teach the student network. Where $\mathcal{H}_{cross}$ denotes the cross entropy and $\mathbf{y}_s$ and $\mathbf{y}_t$ are outputs of student and teacher network respectively. At last, a blend loss is computed to transfer the knowledge to the student network.

$$\mathcal{L}_{kd} = \sum_{i=1}^{n} \mathcal{H}_{cross}\left(\mathbf{y}_s, \mathbf{y}_t\right) \tag{2.21}$$

$$\mathcal{L}_{\text{blend}} = \sum_{i=1}^{n} \{\alpha \mathcal{H}_{\text{cross}}\left(\mathbf{y}_s, \mathbf{y}_t\right) + \mathcal{H}_{\text{cross}}\left(\mathbf{y}_s, \mathbf{y}_{gt}\right)\} \tag{2.22}$$

In [65], Xu et al. adopted a different method. In their work, the knowledge in teacher network is distilled progressively layer by layer from a CNN teacher network. They also compute the mean squared error and then combine it with the cross entropy to formulate the KD loss (Equation 2.23).

$$\begin{aligned} \mathcal{L} &= \beta \mathcal{L}_{\text{mid}} + \mathcal{L}_{\text{blend}} \\ &= \sum_{i=1}^{n} \sum_{m=1}^{M} \beta \mathcal{H}_{mse}\left(\mathbf{y}_a^m, \mathbf{y}_c^m\right) + \sum_{i=1}^{n} \{\alpha H_{\text{cross}}\left(\mathbf{y}_s, \mathbf{y}_t\right) + \mathcal{H}_{\text{cross}}\left(\mathbf{y}_s, \mathbf{y}_{gt}\right)\} \end{aligned} \tag{2.23}$$

With knowledge distillation, the AdderNet is able to learn from CNN and the learned features are more reasonably distributed (see Figure 2.17). The experimental results showed that the accuracy of PKKD (progressive kernel based knowledge distillation) can achieve the same or even better accuracy than the conventional CNNs. On Cifar and ImageNet datasets, the results are shown in Tables 2.1 and 2.2.

AdderNet for Super Resolution Tasks

Reconstructing a high-resolution image from a low-resolution image has been a challenging problem for a long time. In 2014, SRCNN (Super-Resolution Convolutional Neural Network [67]) was proposed

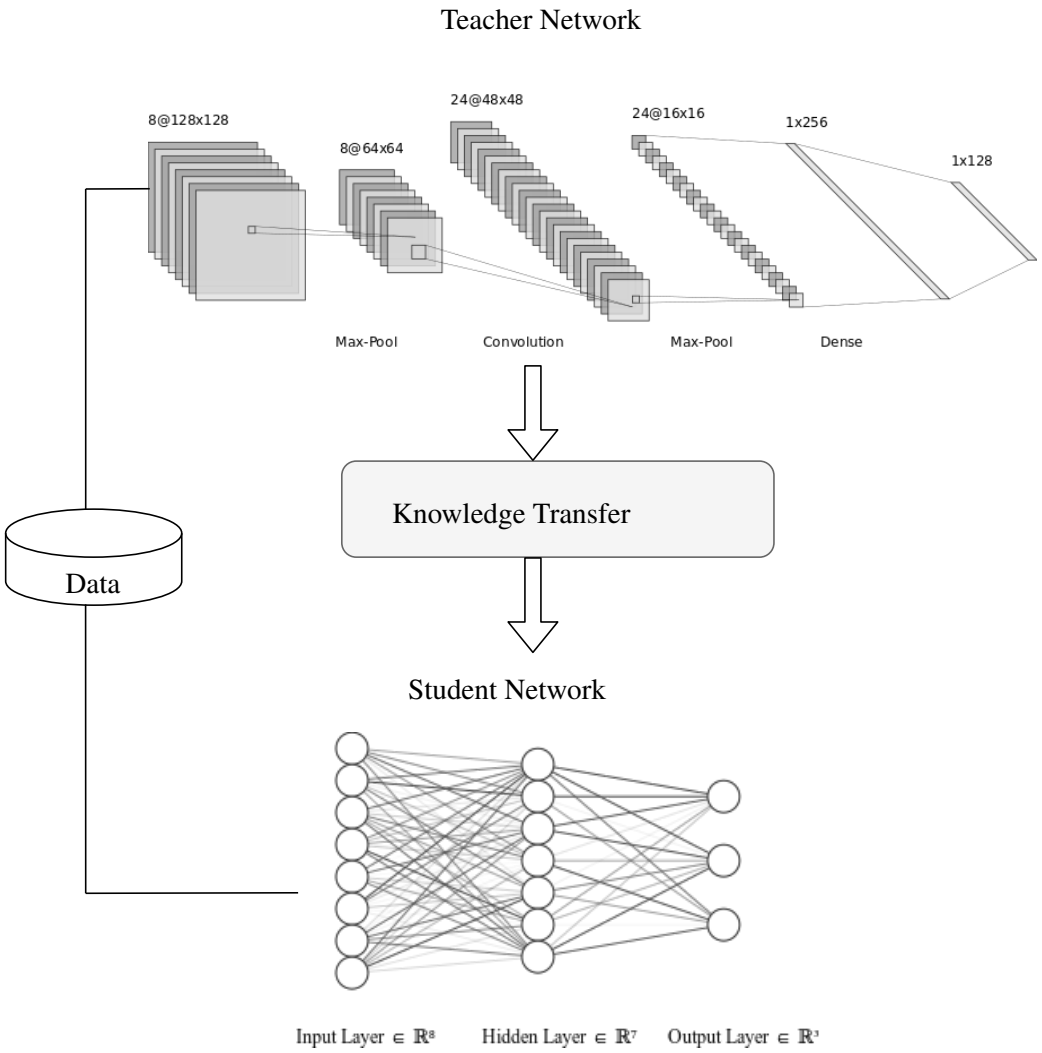


Figure 2.18: Teacher student framework of knowledge distillation. First, the teacher network is trained (usually a large model), then the knowledge is distilled to the student model (a small one for compression). Note the training samples are the same for both teacher and student models.

Table 2.1: Classification results on CIFAR-10 and CIFAR-100 datasets [3]

Model	Method	CIFAR-10	CIFAR-100
VGG-small	CNN	94.25 %	75.96 %
	ANN	93.72 %	74.58 %
	PKKD ANN	95.03 %	76.94 %
ResNet-20	CNN	92.93 %	68.75 %
	ANN	92.02 %	67.60 %
	PKKD ANN	92.96 %	69.93 %
	ShiftAddNet	85.10 %	-
ResNet-32	CNN	93.59 %	70.46 %
	ANN	93.01 %	69.17 %
	PKKD ANN	93.62 %	72.41 %

Table 2.2: Classification results on ImageNet datasets of AdderNets [3]

Model	Method	Top-1 Acc	Top-5 Acc
ResNet-18	CNN	69.8%	89.1%
	ANN	67.0%	87.6%
	PKKD ANN	68.8%	88.6%
ResNet-50	CNN	76.2%	92.9%
	ANN	74.9%	91.7%
	PKKD ANN	76.8%	93.3%

to solve this problem via convolutional neural networks. Since then, many methods based on the neural network have been proposed, such as VDSR [68] and EDSR [69].

Using adder filters, Song et al. achieved a comparable performance while reducing energy consumption by about 50% [70]. They applied the addition neural networks on several datasets (Set5 [71], Set14 [72], B100 [73] and Urban100 [74]). The experimental results show that there is almost no performance gap between AdderNets and the original CNNs (see Table 2.3).

Table 2.3: Experimental results of super resolution with AdderNets.

Scale	Model	Type	Set5 PSNR/SSIM	Set14 PSNR/SSIM	B100 PSNR/SSIM	Urban100 PSNR/SSIM
$\times 2$	VDSR	ANN	37.37/0.9575	32.91/0.9112	31.82/0.8947	30.48/0.9099
		CNN	37.53/0.9587	33.03/0.9124	31.90/0.8960	30.76/0.9140
	EDSR	ANN	37.92/0.9589	33.82/0.9183	32.23/0.9000	32.63/0.9309
		CNN	38.11/0.9601	33.92/0.9195	32.32/0.9013	32.93/0.9351
$\times 3$	VDSR	ANN	33.47/0.9151	29.62/0.8276	28.72/0.7953	26.95/0.8189
		CNN	33.66/0.9213	29.77/0.8314	28.82/0.7976	27.14/0.8279
	EDSR	ANN	34.35/0.9212	30.33/0.8420	29.13/0.8068	28.54/0.8555
		CNN	34.65/0.9282	30.52/0.8462	29.25/0.8093	28.80/0.8653
$\times 4$	VDSR	ANN	31.27/0.8762	27.93/0.7630	27.25/0.7229	25.09/0.7445
		CNN	31.35/0.8838	28.01/0.7674	27.29/0.7251	25.18/0.7524
	EDSR	ANN	32.13/0.8864	28.57/0.7800	27.58/0.7368	26.33/0.7874
		CNN	32.46/0.8968	28.80/0.7876	27.71/0.7420	26.64/0.8033

2.2.4 Decoupled Learning

Learning with Feedback Alignment

Training a deep neural network relies on the chain rule (Equation 2.24), where we do back propagation layer by layer with a global loss function. In the actual training process this is very computationally intensive, and each layer needs to wait for all subsequent layers to complete their gradient computations. If we can decouple the layers, we can save a lot of computation. Based on this idea, many decoupled

gradient methods have been proposed.

$$\Delta C \approx \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l} \Delta w_{jk}^l \tag{2.24}$$

Recent researches showed that strict weight symmetry between the forward and backward propagation pass is not necessary for learning in deep neural networks [75–77]. As shown in Figure 2.19, rather than backpropagate the precise symmetric error matrices, feedback alignment methods backpropagate pseudo error signals which are not precisely computed via the chain rule. Although it is not clear why random pseudo error feedback signals can provide learning in the neural network [76], experimental results showed that useful features could be learned with this method.

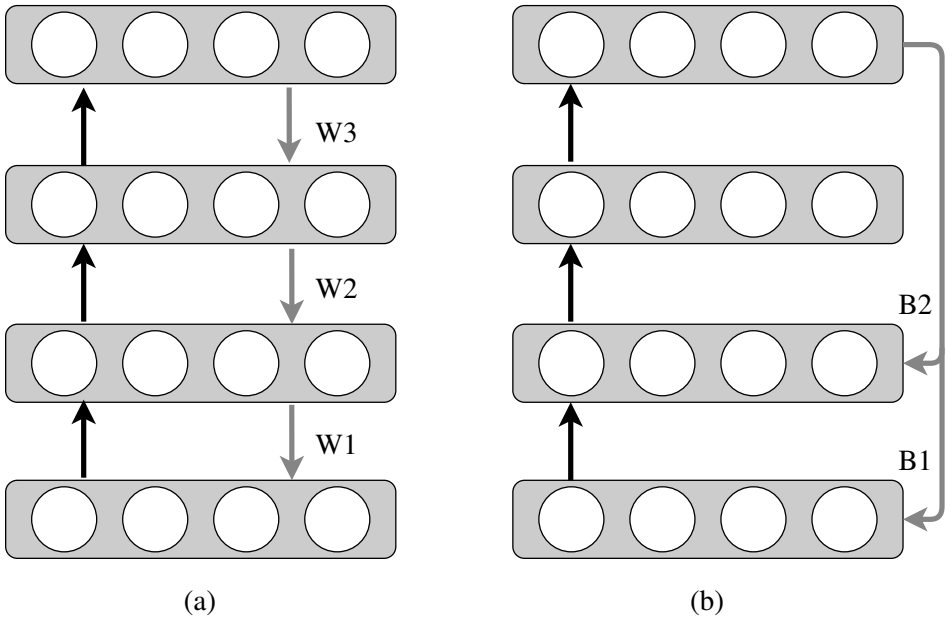


Figure 2.19: Training with feedback alignment. Arrows in black indicate forward signals (activations) and arrows in grey indicate backward error signals. Compared with the traditional backpropagation method (a), feedback alignment backpropagate pseudo error signals are not precisely computed.

Learning with Local Error Signals

Another way to avoid a global loss function is to use local error signals [78, 79]. The traditional approaches follow a forward-backward loop to train the neural network. If the depth of the neural network is large, there are many difficulties in parallelizing neural network computation.

Mostafa et al. took a novel approach by localizing the gradient computation for each layer without relying on a global loss function, as opposed to backpropagating the gradient layer by layer [79]. The advantage of this scheme is that the dependency between the layers can be reduced, thus the synchronization requirement of the training of each layer is reduced, and it has important implications in the parallelization of the network computation.

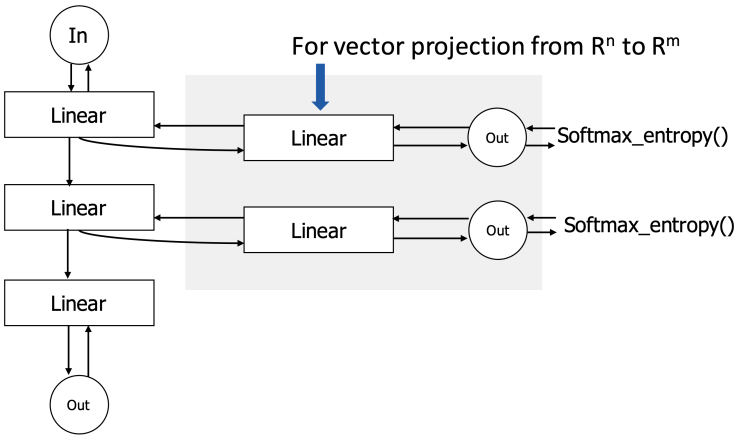


Figure 2.20: training with local error signals.

Chapter 3

Quantization

In this section, our contributions are mainly twofold, first we propose a new logarithmic quantization algorithm which transfer the original integer quantization exponents to decimal. And second, we argue that the widely used quantization error minimization objective is not suitable for designing quantization algorithms. We propose a weight aware quantization error minimization objective that better suits the quantization error computation. It can be assumed that memory reads are very frequent during the convolution operation (Listing 3.1).

3.1 On Hardware Efficiency

A major overhead during neural network training and inference is the memory read operation. Figure 3.1 gives the usual matrix multiplication operation, which is performed in the forward inference of a fully connected neural network. In the case of convolutional neural networks, we usually need to do the unrolling of the convolutional kernel and the input blocks first, and then do the matrix multiplication operation (Figure 3.2).

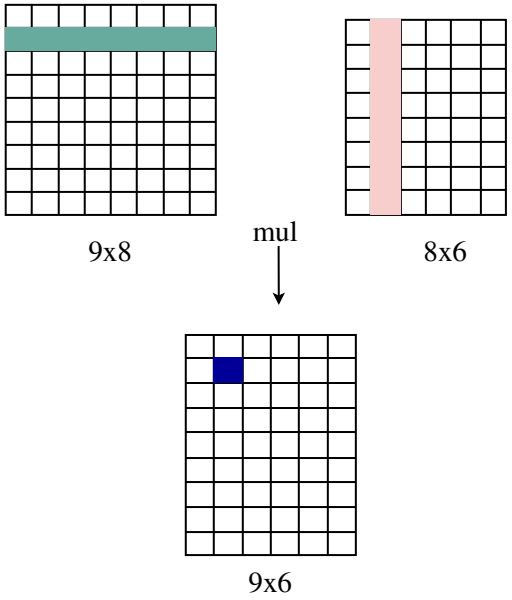


Figure 3.1: General matrix multiplication.

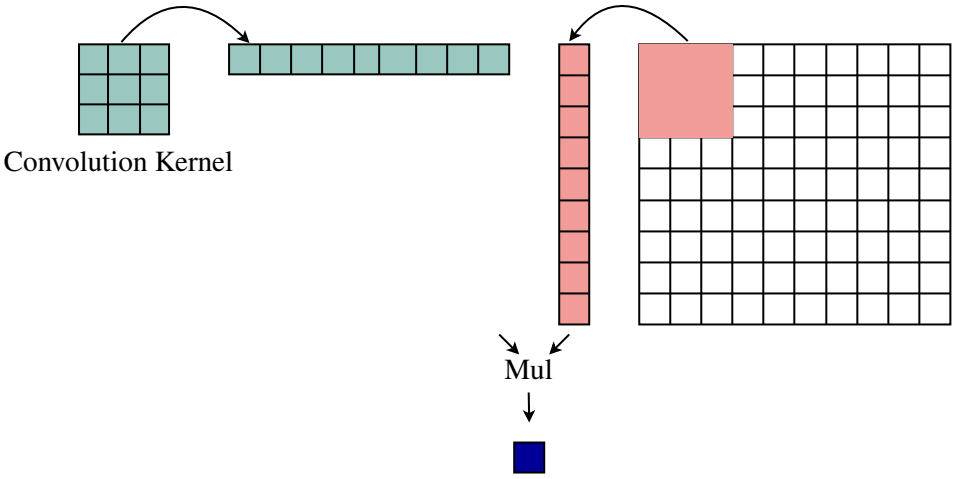


Figure 3.2: Unrolling of an input and convolution kernel.

Quantization can not only reduce the model size of neural networks, but make the neural network inference more efficient as well. Some companies make their own ASICs (application-specific integrated circuit) to accelerate the inference or training of deep neural networks. There are already some special hardware on the market at this point such as Huawei Kirin series' NPU (neural processing unit) and Google's TPU [80] (tensor processing unit, recently Google have customized a TPU acceleration unit on their latest generation mobile phone). Optimizing memory access is common in modern ASICs, CPUs and GPUs. For instance, with little change we can reduce the memory access by $4\times$ as Listing 3.2. Today's processors are largely SIMD (single instruction, multiple data) enabled, so optimizations that reduce memory read frequency operations are also very common even on the mobile side.

```
for m to M {
  for n to N {
    Out[m][n] = 0;
    for k to K {
      Out[m][n] += In[m][k] * Kernel[k][n];
    }
  }
}
```

Listing 3.1: For loops of GEMM.

```
for m to M {
  for n to N {
    Out[m][n + 0] = 0;
    Out[m][n + 1] = 0;
    Out[m][n + 2] = 0;
    Out[m][n + 3] = 0;
    for k to K {
      Out[m][n + 0] += In[m][k] * Kernel[k][n + 0];
      Out[m][n + 1] += In[m][k] * Kernel[k][n + 1];
      Out[m][n + 2] += In[m][k] * Kernel[k][n + 2];
      Out[m][n + 3] += In[m][k] * Kernel[k][n + 3];
    }
```

```
13    }
14 }
```

Listing 3.2: A simple trick to accelerate the GEMM with less memory access.

Some quantization methods constrain model parameters to very low bit widths and use shift operations to replace MAC (multiplyaccumulate) operations since shift operation is faster and less hardware resource intensive. The hardware costs on FPGA (field pragrammable gate array) for different bitwidths' MAC operations also differ [81]. Increasing bitwidth of model parameters can cost a lot more LUTs abd DSPs on an FPGA.

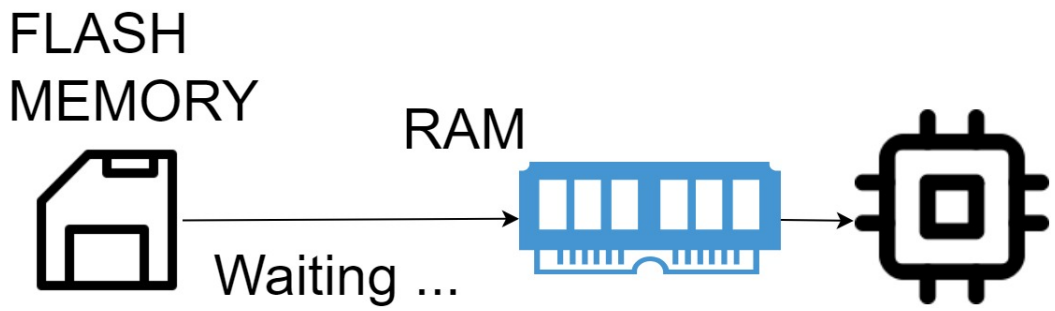


Figure 3.3: Low efficiency if flash memory is accessed frequently.

In real world applications on embedded devices, large storage requirements could cause high latency. Figure 3.10 presents the fact that waiting for parameters from flash memory may cause high latency if the low speed flash memory is accessed frequently. For resource limited devices, it is common to store model parameters in flash memory and then fetch them piece by piece. However this will make the CPU wait and we can expect much faster speed if we can store more parameters in the RAM during inference. Also, reducing the model size is also helpful in reducing the memory bottleneck.

3.1.1 Logarithmic Quantization

The main purpose of quantization is to compress the original deep neural networks. Quantization is an effective way to reduce memory consumption and has recently been adopted by Google's TensorFlow [82]. Various compression methods are detailed in the Chapter 2, and here we give a brief overview of logarithmic quantization (Figure 3.4) proposed by Miyashita et al. [35]. As we can judge from Figure 3.4, logarithmic quantization algorithm takes logarithms of the original weights and saves only exponents. Besides, to constrain the exponents within a smaller bitwidth, the decimal parts of the exponents are removed. As a trade-off, quantized weights incur relatively large quantization noises. To decrease the accuracy deterioration of quantized neural networks, despite troublesome, retraining of quantized networks is widely used and often plays an important role.

Given different bitwidths, the dynamic range of quantized weights varies. Supposing the given bitwidth is 4 then we have at most 16 exponents. Thus we get quantized weights as the equation 3.1. Note 's' in the equation refers to a hyper parameter used to adjust the dynamic range of the quantized weights. In real world applications empirically adjusting this hyper parameter could be cumbersome since the range of model parameters vary between layers. This power of 2 based encoding also enables Zhou et al.'s INQ (Incremental Network Quantization) [32] to function.

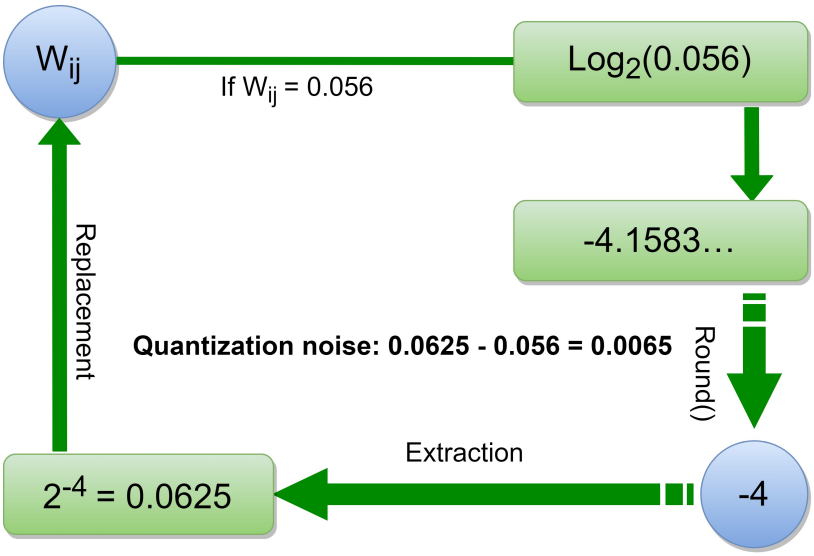


Figure 3.4: An overview of logarithmic quantization.

$$QuanWeights = \{\pm 2^{0+s}, \pm 2^{-1+s} ... \pm 2^{-16+s}\} \tag{3.1}$$

To compare the weights distribution before and after logarithmic quantization, we trained a fully connected neural network on MNIST [83]. The network contains 2 hidden layers with 1000 neurons each. We set s to 0 and the weights distribution is illustrated in Figure 3.5 before and after quantization on the first hidden layer. It is clear that quantized weights on the tail region are sparse. This phenomenon is attributed to the adoption of integer-only exponents. More specifically, only 3 quantized weights between 0.1 and 1 namely 2^{-1}, 2^{-2} and 2^{-3}.

Although the accuracy decrease roots from the sparsity on bigger weights can be alleviated to some extent through retraining, its time and resource consuming. Compared with some large companies with massive datasets and computational resources, small, medium-sized enterprises and universities do not have access to the same volume of datasets and computational resources. Moreover, nowadays open source models in some fields (e.g., image recognition) are publicly available, but the training datasets are not. In this case the retraining of quantized models is not possible. For the above reasons, what we need more is a quantization algorithm that does not require retraining, i.e., the reduction in accuracy after direct quantification is within an acceptable range.

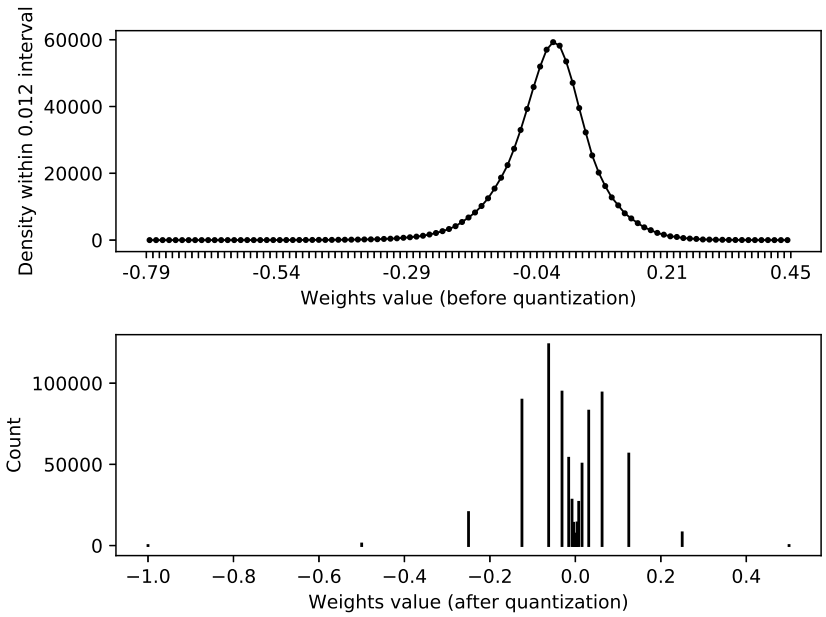


Figure 3.5: Distribution of weights before and after logarithmic quantization. The sparsity of quantization levels on the tail region causes huge accuracy loss.

For some quantization algorithms, such as linear quantization algorithms, the loss of accuracy can be greatly reduced by increasing the bitwidth. For example, for linear quantization, a 4-bit quantization to a 5-bit quantization results in a uniform increase in the precision on the tail region in Figure 3.5. However, for LogQuant, the 4-bit to 5-bit shift does not change anything in the tail region, and all the new quantization levels added in are wasted in the middle region near the 0 value. By improving this problem we can significantly improve the accuracy loss problem.

3.2 Towards Retraining-Free Quantization

3.2.1 Difficulties of Model Retraining.

In chapter 2 we analyzed several compression methods. To largely reduce the parameter size, all of these methods require retraining after model compression. Methods like pruning will even change the network structure of given pre-trained neural networks. Nevertheless, BNNs still cannot keep the accuracy after quantization when challenge large datasets. Figure 3.1 illustrates work flow for normal compression methods. All these methods cannot well preserve the performance of original pre-trained neural networks.

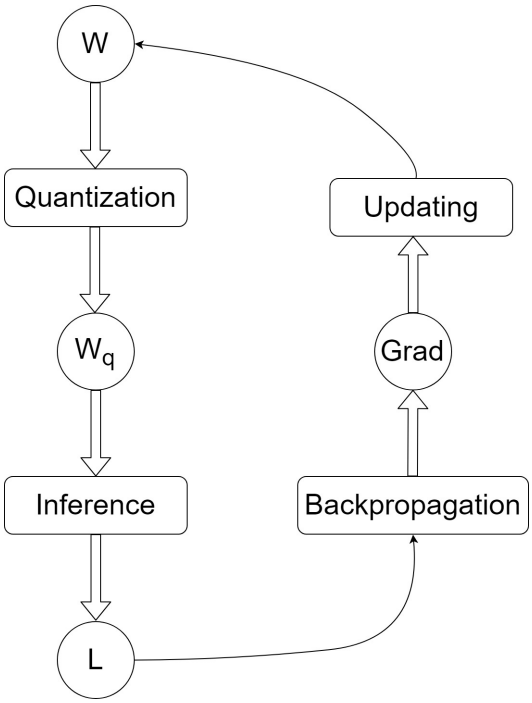


Figure 3.6: Workflow of quantization with retraining.

Many researches are committed to the retraining (right) part of Figure 3.6. It is acquiesced that quantizaiton will cause a huge accuracy loss on pre-trained models therefore retraining is inevitable. However, retraining is not always easy for it usually takes longer to train. Some modern deep learning frameworks do not support quantitative neural network training very well either. Let's take the quantization aware training of Tensorflow as an example. Tensorflow was developed by Google and has received wide adoption in industry and academia.

Listing 3.3 gives an overview for a retrianing sample for a convolution layer. Since the training hardware or platform might not support our datatype when we quantize a model parameter to a specific bitwidth, the retraining (also called quantization aware training in Tensorflow) step is performed by simulation, i.e. fake quantization.

```
class Conv2d_Q(Module):
    def forward(self, x):
        # quantize the layer weights to lower bitwidth
        self.conv_m.w.data = self.quantize_tensor(self.conv_m.w.data)

        # dequantize the layer weights to float32.
        self.conv_m.w.data = self.dequantize_tensor(self.conv_m.w.data)

        x = self.conv_m(x)
        return x
```

Listing 3.3: Quantizatioin aware training.

As indicated in Listing 3.4, the 'quantize_tensor' method is used to reduce the weight precision to our target precision. The 'num_bits' parameter in the function denotes the quantization bitwidth. From the above analysis we can learn that current deep learning frameworks do not natively support low precision

training. This also leads to the need to insert many extra nodes during the training process, which reduces the training speed.

As shown in Figure 3.7, the fake quantization and dequantization nodes are inserted into the training graph which bring extra computation burden. The fake quantization node is used to reduce the weights precision to the target precision, and the dequantization node is used to dequantize the quantized weight to float32 data type, so as to be compatible with the framework. Here for simplicity we depict a single layer quantization procedure, for models with large depth, say ResNet-100, these extra computation burdens will be huge.

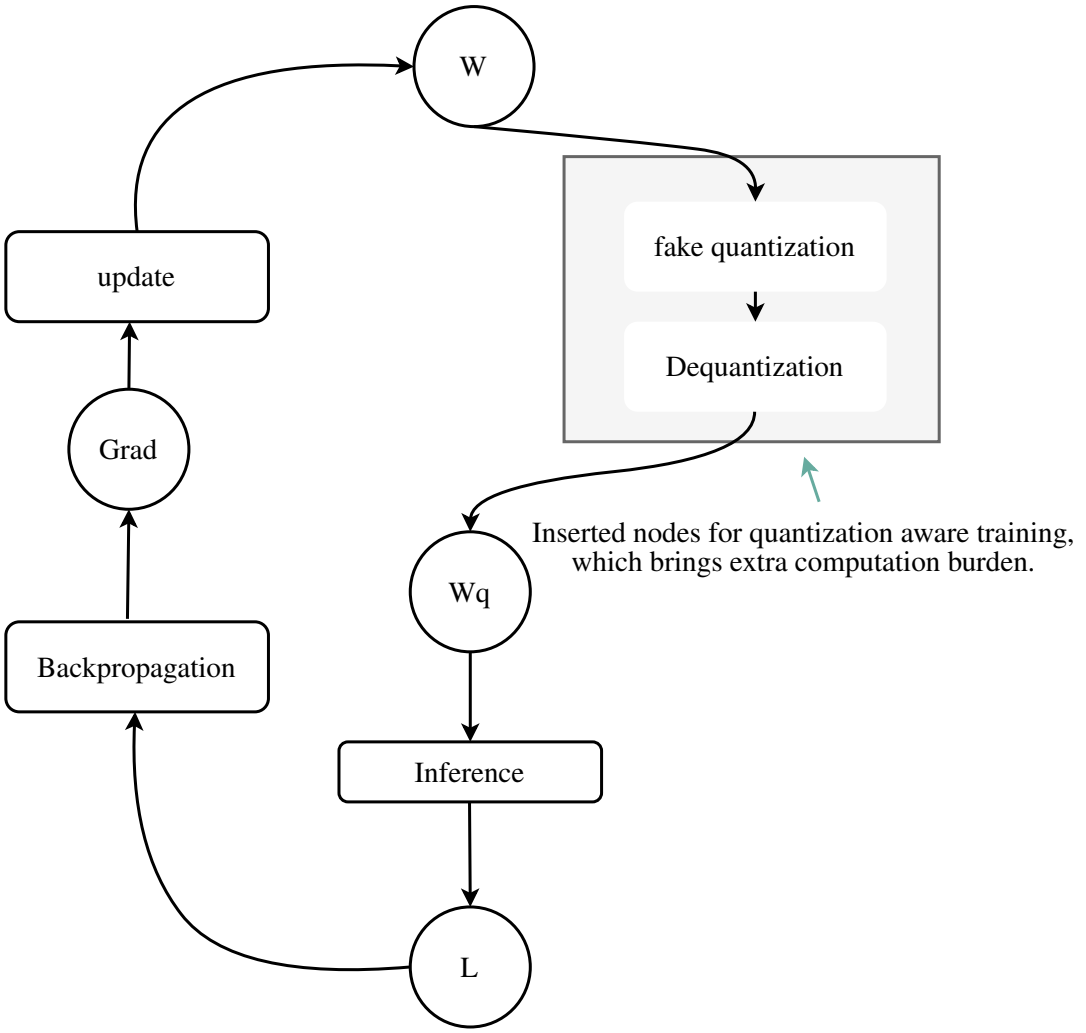


Figure 3.7: Quantization aware training. For compatibility with modern deep learning frameworks, extra nodes such as fake quantization and dequantization are inserted into the graph, which brings extra computation burden.

```
def quantize_tensor(x, scale, zero_point, num_bits=8, signed=False):
    if signed:
        qmin = -2. ** (num_bits - 1)
        qmax = 2. ** (num_bits - 1) - 1
    else:
        qmin = 0.
        qmax = 2.**num_bits - 1.
```

```
8
9       q_x = zero_point + x / scale
10      q_x.clamp_(qmin, qmax).round_()
11
12      return q_x.float()
```

Listing 3.4: A simulated quantization kernel.

Listing 3.4 contains a round function, which is also something to be noted in the retraining algorithm. As shown in Figure 3.8, the round function can only be 0 when calculating the gradient, and this causes the gradient to disappear, so that the neural network cannot continue training. An easy way to do this is to skip all the round functions (straight through estimator) during training and jump directly to the next layer when the gradient needs to be calculated. As shown in Listing 3.5, this skipping strategy could be realized by customizing the backward function.

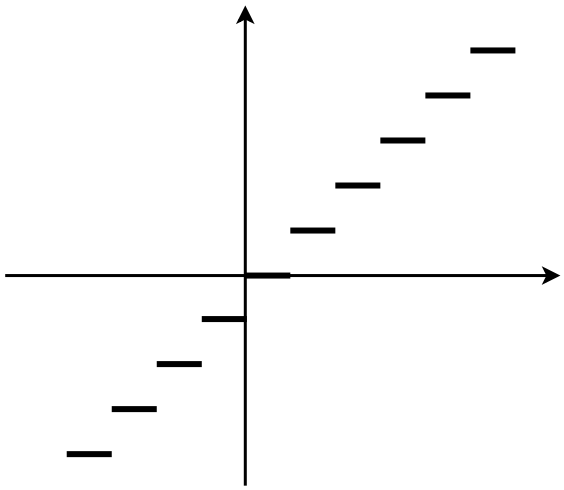


Figure 3.8: Round function. Gradient vanishing happens at round function nodes.

```
1  class Conv2d_Q(Module):
2      def forward(self, x):
3          self.conv_m.w.data = self.quantize_tensor(self.conv_m.w.data)
4          self.conv_m.w.data = self.dequantize_tensor(self.conv_m.w.data)
5          x = self.conv_m(x)
6          return x
7
8      # straight through estimator, pass gradient directly for skipping round
       functions.
9      def backward(self, grad_output):
10         return grad_output
```

Listing 3.5: Straight through estimator

3.2.2 Retraining Free Quantization

We found that retraining is not always necessary if the quantization algorithm is designed well. We will show that if the distribution of weights and **strong** connections (connections with large weight value)

in the neural network are preserved well in precision, the quantized model can perform as well as the full-precision one.

More importantly, a retraining free quantization algorithm can save the time and resource to train the quantized neural network. For practical implementations, there would be no need to design the retraining algorithm since our proposed algorithm can generalize well on both small and large datasets. The work flow of network quantization is then simplified to Figure 3.9.

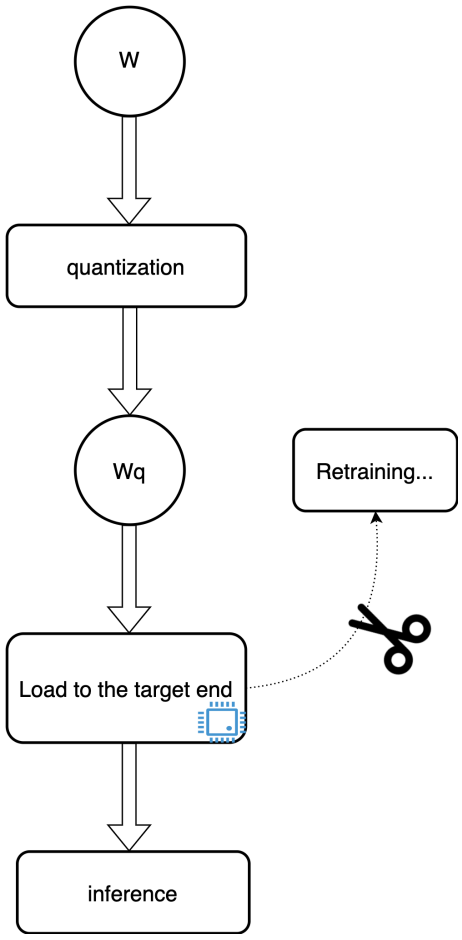


Figure 3.9: Workflow of proposed retraining-free quantization method. The quantized model can be loaded to the end device for inference directly. The time and resource intensive retraining step can be removed.

3.2.3 Drawbacks of LogQuant

Weights of trained neural networks with large complexity are usually highly concentrated around zero. This non-uniformity motivates Miyashita et al. to replace original parameters with logarithmic data representation since logarithmic quantized weights have a similar trend. We restate Miyashita et al.'s algorithm as the following and a graphical view of logarithmic quantization processes is given in Figure 3.10. Their algorithm (details in [35]) will be used as the baseline for comparison.

$$\text{LogQuant(x,bitwidth,FSR)}=\begin{cases}0 & \text{x=0,}\\ 2^{\widetilde{x}} & \text{otherwise}\end{cases} \tag{3.2}$$

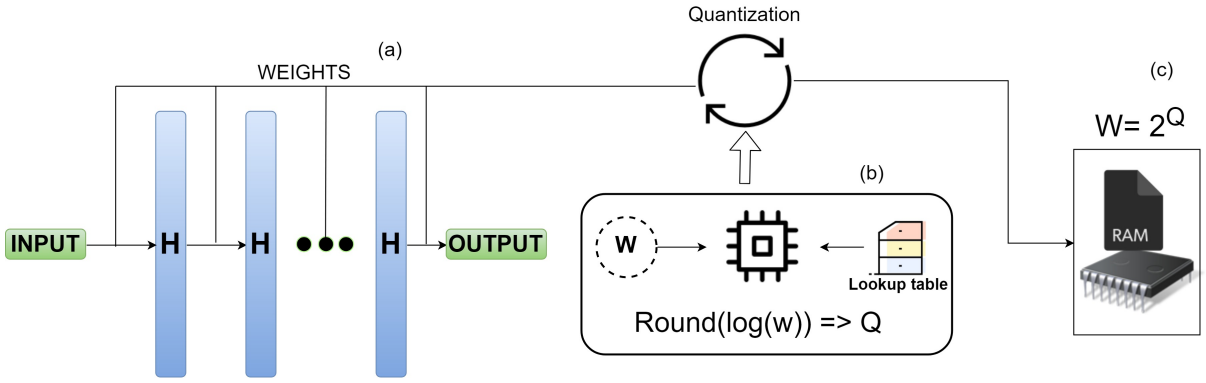


Figure 3.10: Processes of logarithmic quantization. (a) In the first step, weights are extracted for quantization. (b) Weights are quantized. The quantization algorithm takes logarithm of each weight and then rounds it to an integer. (c) Quantized weights are loaded to the target device. Since quantized weights consume less storage, inference on small ram devices could be accelerated.

where

FSR: Full scale range,

$$\widetilde{x} = \text{Clip(Round(log}_2|x|)\text{,FSR-2}^{\text{bitwidth}}\text{,FSR)} \tag{3.3}$$

$$\text{Clip(x,min,max)=} \begin{cases} 0 & \text{x} \leq \text{min}, \\ \text{max-1} & \text{x} \geq \text{max}, \\ \text{x} & \text{otherwise} \end{cases} \tag{3.4}$$

In addition, we define the rounding to the nearest scheme as the following:

$$\begin{cases} \lfloor x \rfloor & \text{if } \left(2^x - 2^{\lfloor x \rfloor}\right) \leq \frac{2^{\lceil x \rceil} - 2^{\lfloor x \rfloor}}{2} \\ \lceil x \rceil & \text{if } \left(2^x - 2^{\lfloor x \rfloor}\right) > \frac{2^{\lceil x \rceil} - 2^{\lfloor x \rfloor}}{2} \end{cases} \tag{3.5}$$

Quantized weights are given by Clip (equation 3.4) and the FSR parameter is manually tuned during the quantization process. The algorithm quantizes both activations and weights to replace multiplication operations with accumulations. Quantization of activation can eliminate the floating multiplications. This can be explained as the following:

if:

$$a = \log_2^x, b = \log_2^w$$

then:

$$wx = 2^{a+b} = \text{ Bitshift } (1, a+b)$$

Based on previous discussions, here we summarize the drawbacks of LogQuant as the following:

- 4bits or larger bit-width quantization brings little to no accuracy improvement than 3 bits quantization.

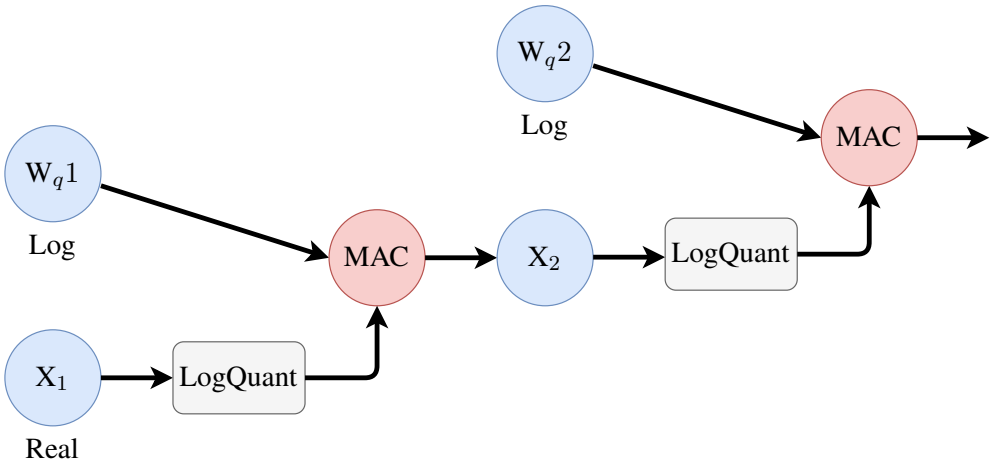


Figure 3.11: Activation quantization brings extra nodes to the computational graph.

- Manually fine tuning the FSR (full scale range) parameter is cumbersome.

  FSR is a hyperparameter that requires manually adjusting. However, it is troublesome if we have to set the FSR parameter for every layer. And it will restrict the implementation for different network architectures. For example, in [35], the value of FSR parameter varies between AlexNet and VGG16. Moreover, introducing FSR to the logarithmic quantization algorithm also brings extra computations such as minus and comparison operations.

- Using only integers in the log domain suffers accuracy decrease under some conditions since it penalizes strong connections (connections with larger weight values).

- Quantization of activations would incur local quantization processing on target devices.

  As shown in Figure 3.11, quantizing the activation introduces extra nodes to the computational graph and it is not necessary since the activation information is not saved to the model (memory compression merit does not exist on activation quantization). If all the weights are quantized to the logarithmic domain, then the multiplication operations between weights and real-valued activations are efficient without activation quantization.

3.2.4 Visualization of LogQuant

Unlike linear quantization, enlarging the bit width of logarithmic quantization can only cause the quantized weights to concentrate around zero. Figure 3.15 and Figure 3.12 show the output of logarithmic quantization, both original weights and quantized weights gather around zero. It can be concluded from Figure 3.13 and Figure 3.14 that regardless of how much we increase the bit width of LogQuant, quantized weights that are away from zero are not affected. For instance, LogQuant can only generate two values that are greater than 0.2. One important point is that when increasing the number of the hidden layer neurons, the Law of Large Numbers begins to function. When the number of hidden layer neurons is relatively small, increasing the number of neurons will cause the weights distribution to become smooth.

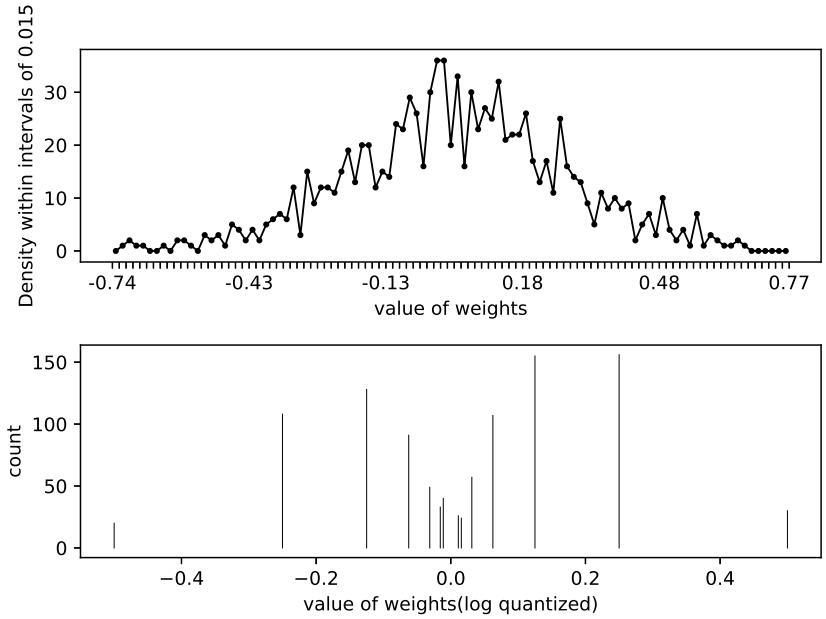


Figure 3.12: 3 bits logarithmic quantization (16 neurons in hidden layer)

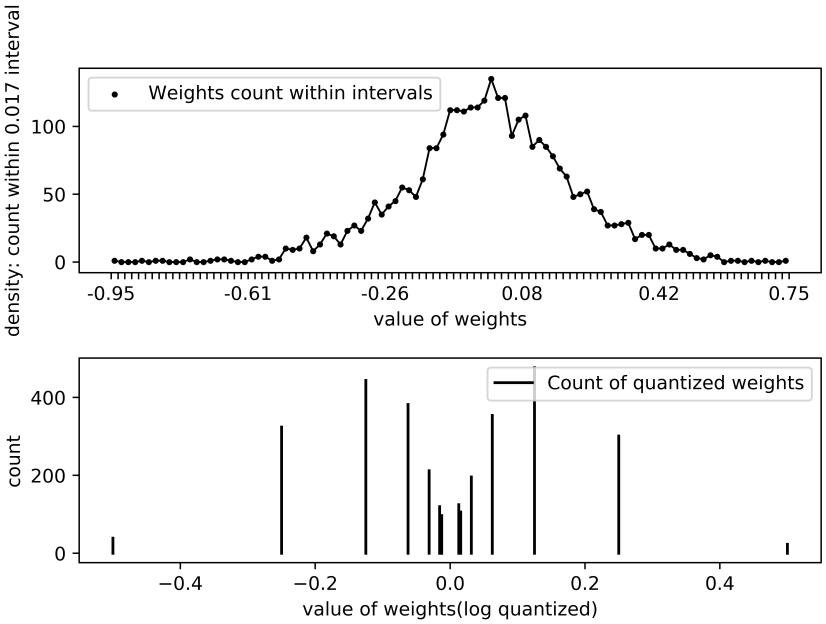


Figure 3.13: 3 bits logarithmic quantization (50 neurons in hidden layer).

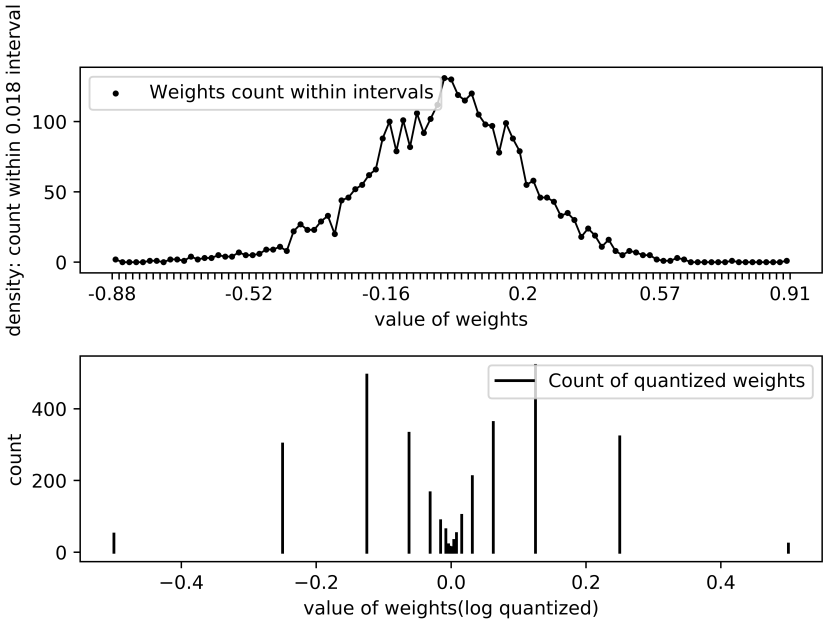


Figure 3.14: 4 bits logarithmic quantization (50 neurons in hidden layer).

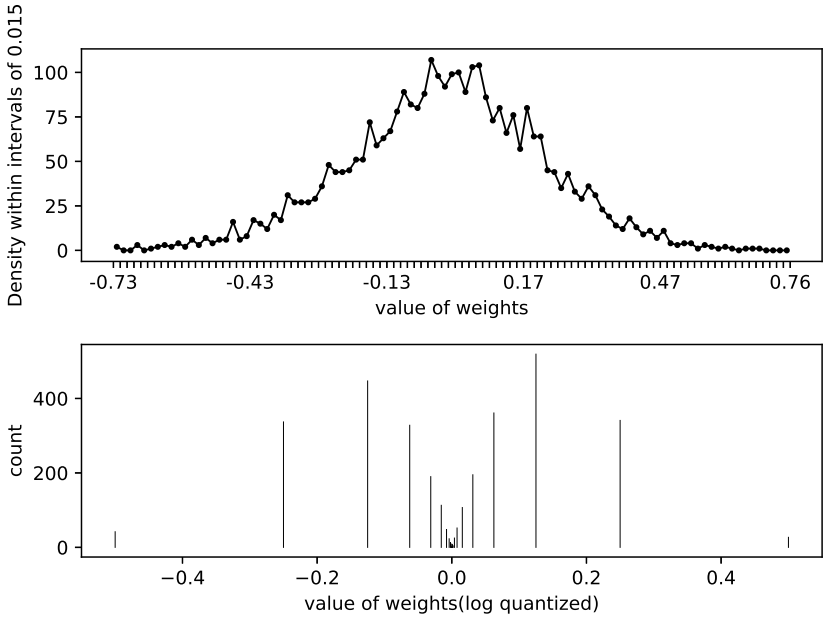


Figure 3.15: 7 bits logarithmic quantization (50 neurons in hidden layer).

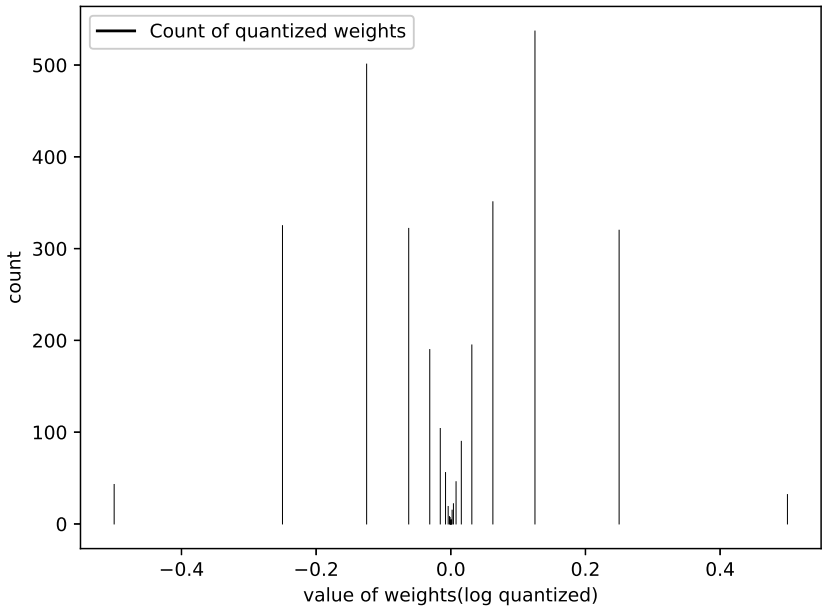


Figure 3.16: 32 bits logarithmic quantization (50 neurons in hidden layer)

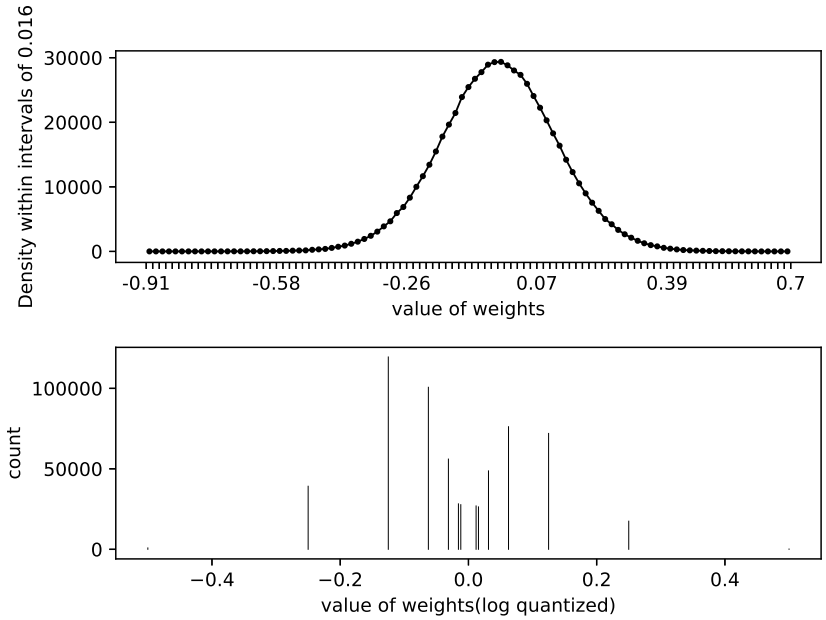


Figure 3.17: 3 bits logarithmic quantization (10000 neurons in hidden layer)

Figure 3.17 shows an example that the number of hidden layer neurons is relatively large (10000 in this case). The density distribution shows that weights density near two sides is almost squeezed to zero. It is also noticeable that the proportion of quantized weights on the tail region is small compared to Figure 3.15 and Figure 3.12. This phenomenon, from another point of view, well explains why Miyashita et al.s work on large neural networks can achieve no deterioration in accuracy. In this case, the network is oversized therefore the weights in the hidden layer are relatively small. The larger layer size we have, the more concentration of weights around zero we will get, which leads to concordance between original

and quantized weights.

3.3 Experiment on A Microprocessor

3.3.1 RISC-V

We experimented with digital recognition on a RISC-V device. RISC-V is an open source ISA (Instruction set architecture) which is available for both academia and industry. Compared with other commercial ISAs such as X86 and ARM, RISC-V is free, simple and highly customizable. We believe in the near future there will be many kinds of RISC-V based IoT devices functioning everywhere.

RISC-V ISA is defined as a base integer ISA with optional extensions. There are only 47 basic instructions which is easy to implement. The specification sheet of RISC-V is just 145 pages (version 2.2) which is much less than any other ISAs.

3.3.2 On Hifive-1 Board

Freedom E310 development board is chosen as our testing platform (on Sive Hive-1 board). The following explains why we choose Hifive-1 (RISC-V) to perform the implementation:

- Clock speed: 320+ MHz, Performance: 1.61 DMIPs/MHz, 2.73 Coremark/MHz.
- GNU tool chain support.
- Open and extensible ISA which allows customized commands.
- Compared to other models, Hifive-1 is cheap.

The RAM size of Hifive-1 is limited to only 16KB. Figure 3.18 presents the architecture of Hifive-1 board and Figure 3.19 is the actual picture of the development board. To perform tasks based on neural networks, a traditional method would store the weight and bias parameters in the external flash first, then load them into the RAM piece by piece for calculation. However, this method is exceedingly inefficient and unlikely to be implemented in practical applications. Therefore, to meet the needs of real-world applications on low cost ends, it is necessary to reduce the parameter size and access to low speed memories.

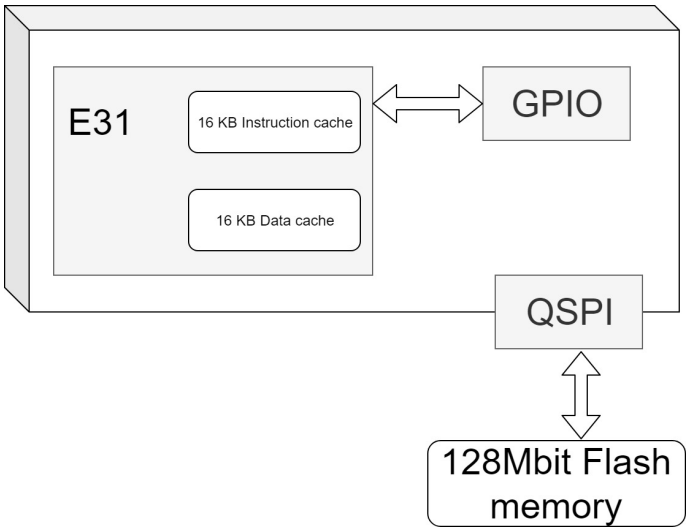


Figure 3.18: Architecture of Hifive-1 borad.

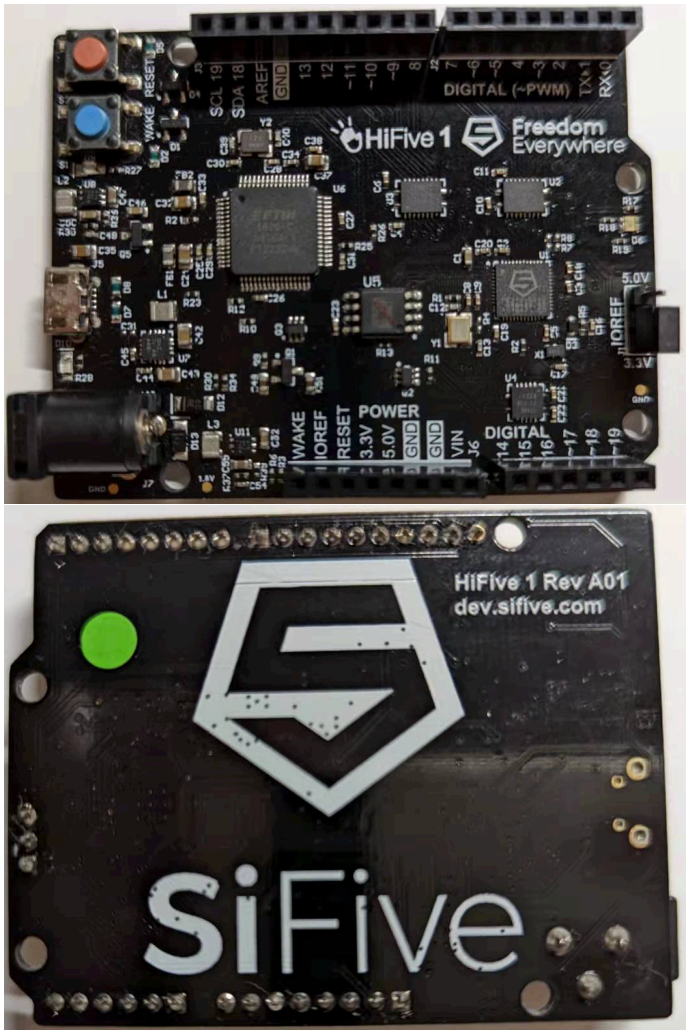


Figure 3.19: hifive board..

Although the training processes are time-consuming and require a large amount of computations, a

trained model is purely a group of matrices. We exported these matrices to arrays in C and performed several recognition tests on the Hive-1 board. On account of the fact that adding depth to the network will exponentially increase both the weights size and the amount of computation, we choose a single hidden layer neural network as the classifier. This setting is useful for embedded systems since their RAM size might be extremely limited. The algorithm is tested on neural networks containing 784 or 64 input neurons, one hidden layer of 16 neurons and an output layer of 10 neurons. Table 3.1 gives the evaluation result.

Table 3.1: Accuracy of a small size FC network

Input size	Accuracy
784 (without feature extraction)	94.125
64 (with feature extraction)	96.035

We used Chainer [62] to define and train the simple fully connected network. The network definition is given in Listing 3.6. Chainer adopts a define-by-run strategy so that the architecture of the entire neural network is dynamic and can be adjusted in real time while the neural network is running. Recently Tensorflow has enabled an eager execution mode, which is similar to Chainer's define-by-run strategy.

```
class MLP(chainer.Chain):

    def __init__(self, n_units, n_out):
        super(MLP, self).__init__()
        with self.init_scope():

            self.l1 = L.Linear(None, n_units)  # n_in -> n_units
            self.l2 = L.Linear(None, n_units)  # n_units -> n_units
            self.l3 = L.Linear(None, n_out)  # n_units -> n_out

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)
```

Listing 3.6: Network definition.

As the result, the feature extraction algorithm reduced the input neurons by 12.25×. Moreover, the combination of feature extraction and neural network outperforms the original fully connected neural network on the small size FC network. We also added some control experiments with small feasibility on Hive-1 for reference, the results are listed in Table 3.2.

The combination of feature extraction algorithm and quantization achieves a reasonable accurate recognition rate. Although Miyashita et al.s work show negligible or no loss in classification performance when logarithmic quantization been applied to fully connected layers [35]. Our work reveals that small sized DNNs cannot maintain the accuracy merit of logarithmic quantization. After quantized, the

networks recognition performance decreased substantially compared to the results in Table 3.1. There is a trade-off between accuracy and efficiency, especially on resource limited microchips.

Our work explores the practicability of implementing a small sized DNN on a microprocessor of extremely limited RAM and computational capability. Here we summarize the workflow of the entire processes targeting on embedded devices of limited resources. This should work on other datasets like fashion mnist, cifar10 and some other real world application datasets.

- Design a feature extraction algorithm which performs dimension reduction and preserves the relevant characteristics.
- Decide a network that meets the precision of the target application. Shallow depth with a hidden layer of considerable size is preferred by the Law of Large Numbers.
- Logarithmic Quantization of model parameters.
- Export the model parameters to the target device(s).

Table 3.2: Accuracy results.

Bit width and hidden layer size	Accuracy(%)
3 bits with 16 neurons	82.48
3 bits with 32 neurons	82.55
3 bits with 64 neurons	90.28
3 bits with 128 neurons	93.66
3 bits with 256 neurons	93.78
3 bits with 10240 neurons	95.88
3 bits with 102400 neurons	96.60
7 bits with 16 neurons	74.74
7 bits with 32 neurons	90.30
7 bits with 64 neurons	90.08
7 bits with 128 neurons	94.58
7 bits with 256 neurons	94.36
7 bits with 10240 neurons	96.58
7 bits with 102400 neurons	96.20
32 bits with 16 neurons	85.48
32 bits with 32 neurons	86.42
32 bits with 64 neurons	93.22
32 bits with 128 neurons	94.92
32 bits with 256 neurons	94.82
32 bits with 10240 neurons	96.36
32 bits with 102400 neurons	96.90

3.4 Decimal LogQuant

The weak points mentioned in section 3.2.3 might restrict the real world implementations of the proposed logarithmic quantization algorithm. Our purpose is to develop a new algorithm that keeps the advantages of the existing logarithmic quantization, while this algorithm shall get rid of the disadvantages from algorithm 1 (or LogQuant, existing logarithmic quantization algorithm). To achieve this goal, we extend the exponents from the integer domain to the real number field thus quantized weights now have a higher density. Extended exponents which beyond the precision of given bitwidth are kept in a lookup table. Moreover, We remove quantization of activations since quantized weights can be directly loaded into small devices of low computational power, while quantization of activations will incur local computing for quantization on these devices. Another benefit we gain from removing quantization of activations is that the time of tuning FSR parameter is largely saved. Therefore the generality of the algorithm is significantly increased.

In addition, we round all weights below a threshold to a minimum weight value. Stronger connections often perform prior roles in the forward computation. To better illustrate this point, we did some pruning tests on several networks. Making the use of Chainer [61] we take a snapshot of all parameters in a two layers FC (fully connected) network trained on MNIST, VGG [84] on Cifar10 and GoogLeNet on ImageNet dataset. Note networks we used are pre-trained ones from caffe's model zoo [85]. We pruned certain proportions of stronger connections in a certain layer and record the result. Then we repeated the same experiment but this time we pruned the weaker connections of the same proportion. Specifically, the FC network contains 2 hidden layers of 1000 neurons in each and we pruned 10 percent connections in the second hidden layer. On the GoogLeNet, 10 percent connections in the second convolution layer were pruned and to our surprise, a loss of 10 percent stronger connections would dramatically decrease the accuracy. Another interesting fact is that the accuracy loss on pruned VGG was inconspicuous until we pruned 30 percent of the connections. It is evident that VGG on Cifar10 is over-parameterized. The accuracy loss is presented in Table 3.3.

Table 3.3: Accuracy Loss After Pruning

Model	Upper pruning	Lower pruning
FC Network	38.56%	0%
GoogLeNet	78.45%	0%
VGG	11.19%	0%

The unimportance of smaller weights inspired us to round weights below a threshold to a specific minimum value. Here we provide details of our algorithm. We firstly generate an arithmetic progression via equation 3.5. Parameter R is a positive real number. In our cases, we set R to 7 on account of the fact that quantized values range from 0 to -7 can be decoded to weights ranging from $7.8e-3$ to 1. In practice, this range is appropriate for our tested networks and R could be tuned based on the complexity of a specific layer.

$$\text{QuanSeries} = \{\frac{-\text{R}x}{2^n} \mid (x \in 0 \sim 2^n) \cap (x \in Z), n\text{=bitwidth}\} \tag{3.6}$$

Secondly, produced series contains float numbers which cannot be stored within the given bitwidth. Hence a lookup table is utilized to store these exponents. In this way, higher bitwidth offers denser exponents which are particularly useful for strong connections. Note that when the bitwidth is 3 our algorithm performs similar to algorithm 1. For the simplicity of implementation, we do not prune all connections below the threshold but round them to the bottom of the arithmetic progression, namely 2^{-7}. We will show in chapter 4 that this rounding scheme does not affect the accuracy.

After quantization, only indexes of the series are stored in the memory. We use Numpy's argmin [86] method to perform rounding to the nearest scheme (Listing 3.1). Note that the $'-'$ operator would be executed element-wisely. We ignore the process of fetching weight signs for ease of explanation.

```
Indexes = (QuanSeries - numpy.log2(numpy.abs(W))).argmin()
// W: weight matrices
```

Listing 3.7: Generating Indexes

Indexes could be directly loaded to target devices. Before inference on targets devices, the quantized weights should be reconstructed via equation 3.6. '$\widetilde{W}$' denotes the quantized weights.

$$\widetilde{W} = 2^{\text{QuanSeries[Indexes]}} \tag{3.7}$$

An overview of our algorithm is given in Figure 3.20. Compared with the existing logarithmic quantization algorithm, algorithm 2 (or Decimal LogQuant) can round original weights to a quantized version with smaller quantization noise on strong connections. Figure 3.21, 3.22 and 3.23 show weight distributions after 4 or 5 bits quantization. Weights are extracted from the same layer of the pre-trained FC network (shown in Figure 3.5). Compared with Figure 3.5, it is notable that our algorithm outputs a more reasonable distribution especially for stronger connections.

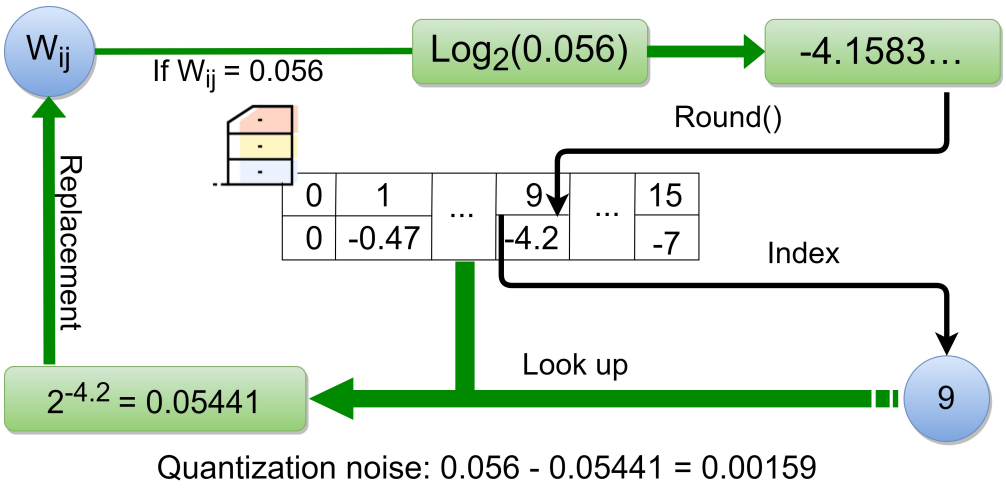


Figure 3.20: An overview of the proposed quantization algorithm.

It is clear that increasing the bitwidth can solve the sparsity of strong connections in our algorithm. Particularly, 4 bits and 5 bits quantization with the original LogQuant show almost no difference. On the contrary, our algorithm with the same condition can increase the density of quantized weights on the tail region. The ability to maintain the coherence of strong connections between original and quan-

tized weights is the key to realize retraining-free quantization. Owing to its simplicity, we realize the quantization kernel within 20 lines of code (listing 3.8).

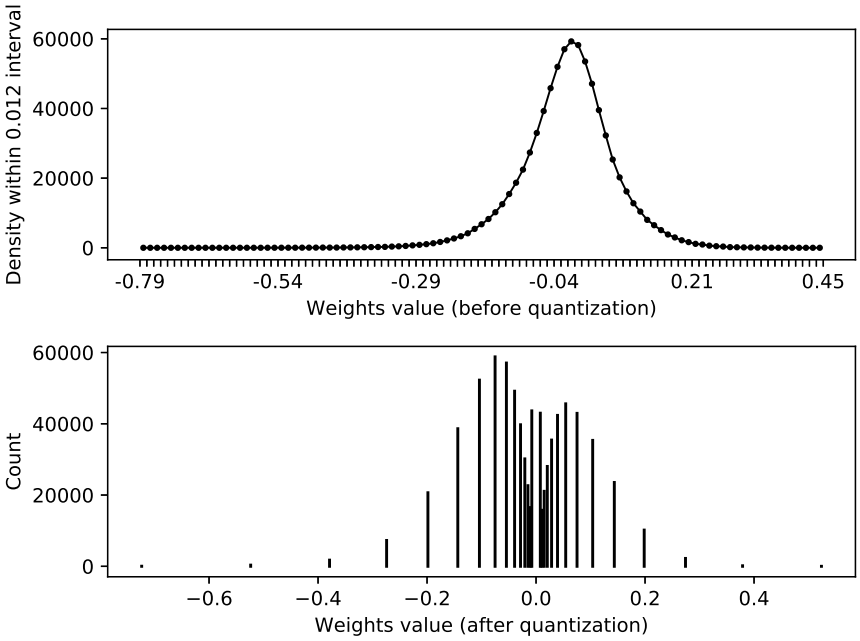


Figure 3.21: 4 bits quantization via proposed algorithm.

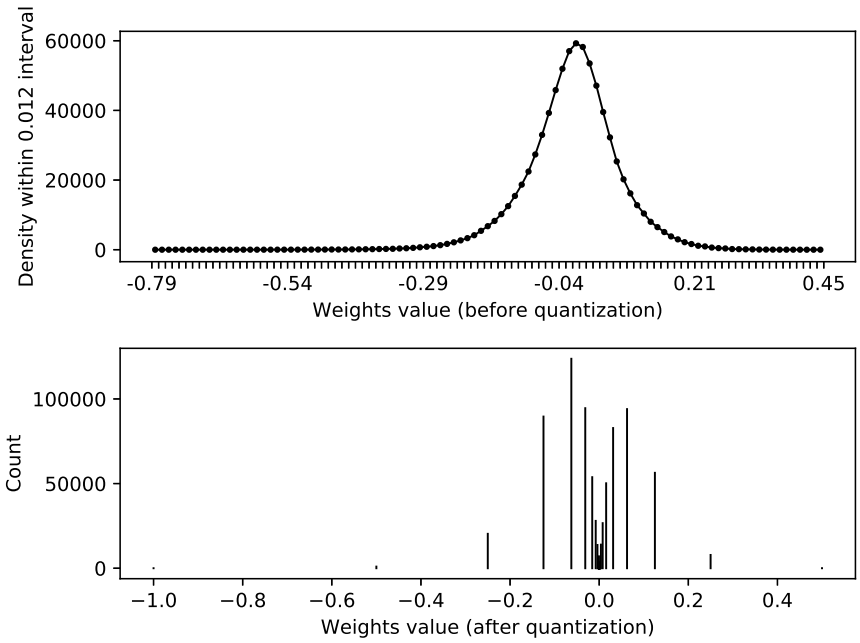


Figure 3.22: 5 bits quantization via algorithm 1. The distribution of quantized weights on tail region shows almost no difference with 4 bits quantization.

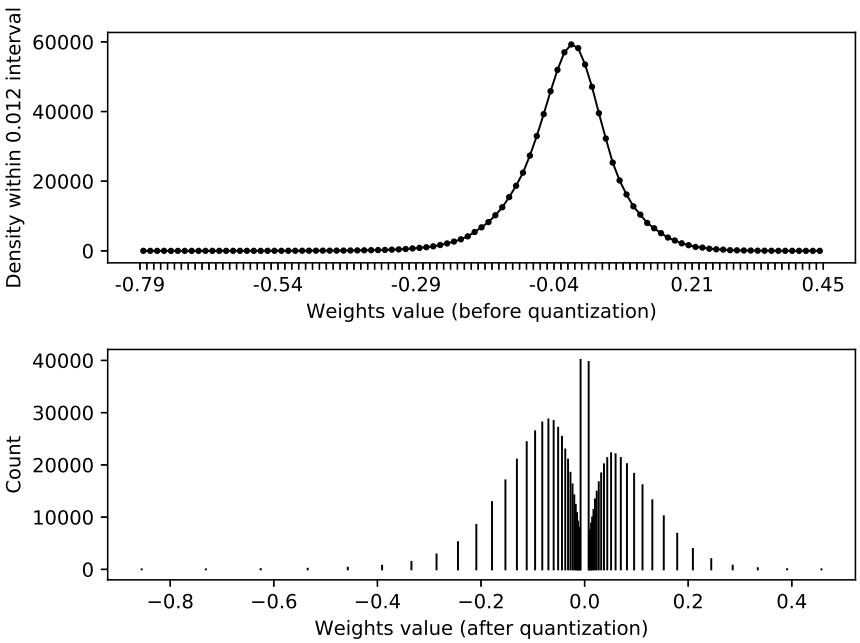


Figure 3.23: 5 bits quantization via proposed algorithm.

We set R to 7 in equation 3.6 which can quantize small weights to value 0.0078125. In our experience this value is already small enough for weak connections. Listing 3.8 gives the quantization kernel of our algorithm. Key operations in the kernel are computed element wisely using numpy. Therefore it is compatible with most deep learning platforms.

```
import numpy
class LogQuant:
    def __init__(self,layer,bitwidth=4):
        self.layer_data = layer
        self.width = bitwidth
        self.sign = numpy.sign(layer)
        self.lookup = numpy.linspace(0,-7,2**self.width)
    def __round(self,x):
        idx = (numpy.abs(self.lookup - x)).argmin()
        return idx
    @property
    def log_quantized(self):
        round = numpy.vectorize(self.__round)
        return numpy.array(round(numpy.log2(numpy.abs(self.layer_data))),dtype=numpy
    .int8)
    @property
    def de_quantized(self):
        x = numpy.power(2.0, self.lookup[self.log_quantized])
        x = numpy.array(x,dtype=numpy.float32)
        return x * self.sign
```

Listing 3.8: Quantization kernel

3.4.1 Quantization Error

Quantization error (equation 3.8) is used to measure the performance of the quantization algorithm in [35] and [24], whereas we would argue that it is inaccurate to use equation 3.8 as the evaluation criterion especially for over-parameterized networks. Equation 3.8 treats all weights equally while we already know that pruning different parts of a neural network will cause different effects. When all connections are treated equally, the prior role of stronger connections is ignored. Nonetheless, we measured the mean squared error of two algorithms on a pre-trained VGG network and the results are collected in Table 3.4.

$$E=\frac{1}{2}\sum_{i=1}^{N}(Q(w_i)-w_i)^2 \tag{3.8}$$

Table 3.4: Quantization Noise (4 Bits Quantization)

VGG	Weights Number (K)	LogQuant	Decimal LogQuant
conv1_1	1.7	0.300	0.079
conv1_2	36.9	0.501	0.228
conv2_1	73.7	0.802	0.384
conv2_2	147.5	1.274	0.710
conv3_1	294.9	1.664	1.455
conv3_2	589.8	1.893	**3.206**
conv3_3	589.8	1.870	**3.311**
conv4_1	1179.6	1.840	**8.816**
conv4_2	2359.3	1.064	**28.395**
conv4_3	2359.3	0.369	**44.201**
conv5_1	2359.3	0.139	**53.469**
conv5_2	2359.3	0.148	**50.590**
conv5_3	2359.3	0.201	**51.053**
fc1	26.2	0.154	**4.057**
fc2	5.1	0.599	0.214

Obviously, our proposed algorithm (Decimal LogQuant in the table) outperforms the original one (LogQuant) if we judge through their quantization error. However, as we will demonstrate in the following sections, our algorithm still surpasses the original logarithmic quantization algorithm. We can then conclude that over-parameterized networks are more capable of noise tolerance. Though quantization error of certain layers is heavier, it wouldn't impose a huge impact on final accuracy.

3.4.2 Benchmark on a Fully Connected Neural Network

At first, a simple FC network with 2 hidden layers is used to explore the limits of our algorithm. We gradually decrease the number of hidden layer neurons thus the size of each layer decreases respectively. Then the network is quantized using two algorithms. The results of two algorithms are gathered in Table 3.5.

Table 3.5: 2 Layers fully connected network on MNIST

Hidden Layer Size	LogQuant	DLQ	FP.
1000	0.9831	0.9846	0.9841
600	0.9796	0.9801	0.9792
200	0.9798	0.9814	0.9811
100	0.9726	0.9753	0.9751
80	0.9748	0.9744	0.9751
60	0.9611	0.9683	0.9723
40	0.9510	0.9690	0.9701
20	0.9159	0.9550	0.9626
10	0.8056	0.9300	0.9376

It is noticeable that even when hidden layers contain only 10 neurons, algorithm 2 can still keep close to the original accuracy. On the contrary, the original logarithmic quantization algorithm suffers big accuracy loss when the hidden layer size is small.

3.4.3 Benchmark on Convolutional Networks

VGG and GoogLeNet

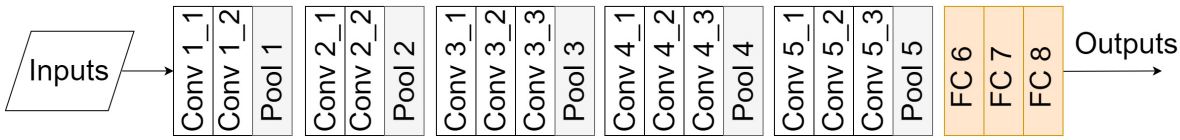


Figure 3.24: Network architecture of VGG16.

VGG [84] is a convolutional neural network model proposed by Simonyan et al from Visual Geometry Group at Oxford University. The network was used on ILSVRC2014 and showed that adding depth to the network can bring better results. VGG has several architectures which differ in depth, in this paper we use VGG16 (Figure 3.24) for evaluation. In addition, compared with AlexNet [87], VGG uses smaller 3 by 3 convolution filters rather than 7 by 7 or 11 by 11 filters in AlexNet. All 3 by 3 filters are convolved with input at every pixel. Smaller filters can better preserve the original information of the dataset and reduce the size of model parameters. Two 3 by 3 filters could be used to replace one 5 by 5 filter (illustrated in Figure 3.25). About the reduction of parameter size, 7 by 7 convolution needs $49C^2$ (C is the number of channels) parameters while three 3 by 3 convolution only requires $27C^2$ parameters.

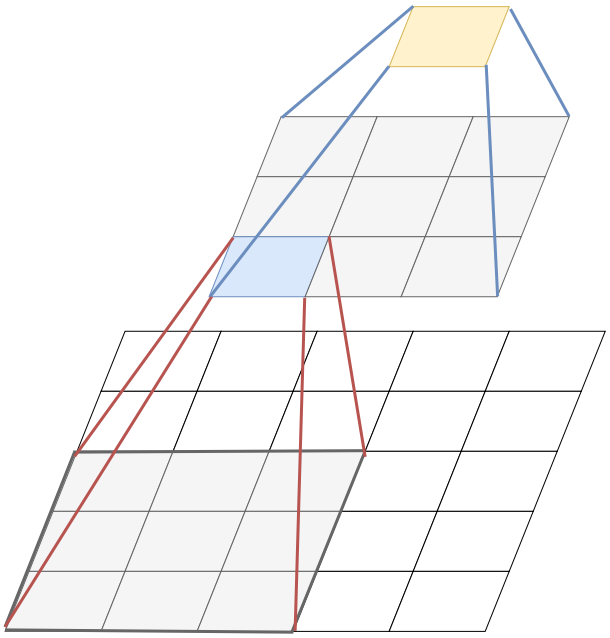


Figure 3.25: Two 3 by 3 filter replacing one 5 by 5 filter.

GoogLeNet [88] is the champion of ILSVRC2014. It utilizes Inception modules to solve some problems of the previous neural network models. Large CNN models are always accompanied by the very large size of parameters, which make it hard to train. Sometimes large network size could be a problem of overfitting. Another problem that CNNs bring is the sparse data structure, which makes it inefficient to process during training.

GoogLeNet shares some similar ideas of NiN [89]. In the Inception module, filters of different sizes can extract various abstractions. Inception use 1 by 1, 3 by 3, 5 by 5 convolutional filters along with a 3 by 3 max pooling filter. The 1 by 1 filters could be used to reduce dimension and they can also be used the same way as ReLU functions. All four filters are applied to the input array and their outputs are concatenated. A typical inception module looks like Figure 3.27.

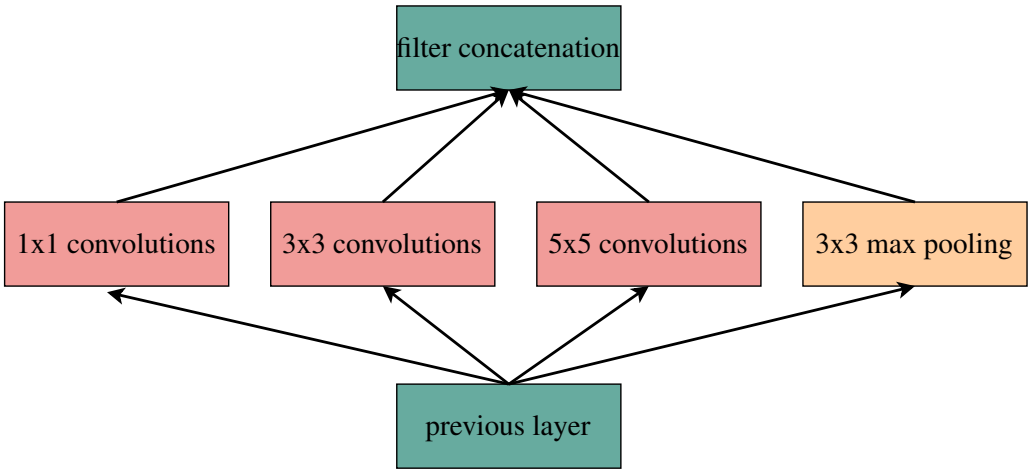


Figure 3.26: Inception module, naive version.

Much of the previous work was based on networks such as Alexnet and VGG to further deepen the neural network, while directly deepening the neural network tends to lead to problems such as gradient

vanishing, gradient exploiting and overfitting. The proposal of Inception improves the network performance from another perspective: it can use computational resources more efficiently and extract more features with less computation.

As shown in Figure 3.27, compared with Figure 3.26, many 1x1 convolutions are used for dimension reduction. Note in Figure 3.27 every convolution operation is followed by an activation function (ReLU) for increasing the non-linearity. 1x1 convolution is operationally equivalent to a fully connected neural network.

Here we give an example to illustrate how 1x1 convolution reduces the computational effort. Suppose the input is 192 32x32 feature maps, and the output is 256 32x32 feature maps generated by 3x3 convolution, which requires about 453 million multiplications. With 1x1 convolution we can first reduce the 192 32x32 feature maps to half (96) and then perform a 3x3 convolution and upscale the number of feature maps to 256, in which case we add nonlinearity and reduce the number of multiplications to 245 million.

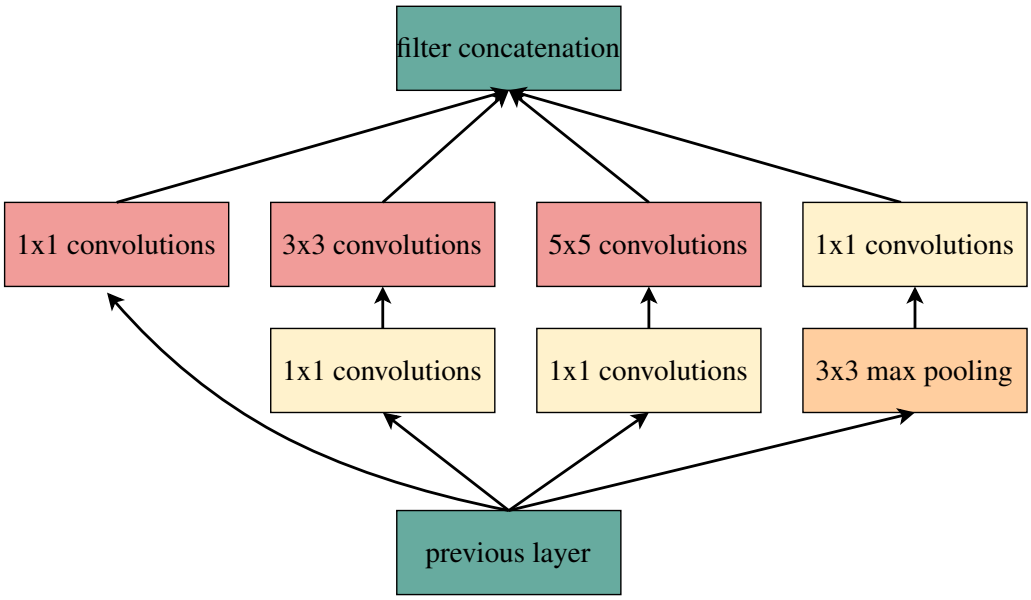


Figure 3.27: Inception module, with dimension reductions.

The GoogLeNet is composed of several inception modules (refer to Listing 3.9 and Figure 3.28). Through the architecture of GoogLeNet we can see that increasing the sparsity of the network can effectively enhance the learning ability of the network as the depth of the neural network deepens.

```
class Inception(link.Chain):
    def __init__(self, in_channels, out1, proj3, out3, proj5, out5, proj_pool,
                 conv_init=None, bias_init=None):
        super(Inception, self).__init__()
        with self.init_scope():
            self.conv1 = convolution_2d.Convolution2D(
                in_channels, out1, 1, initialW=conv_init,
                initial_bias=bias_init)
            self.proj3 = convolution_2d.Convolution2D(
                in_channels, proj3, 1, initialW=conv_init,
                initial_bias=bias_init)
```

```
            self.conv3 = convolution_2d.Convolution2D(
                proj3, out3, 3, pad=1, initialW=conv_init,
                initial_bias=bias_init)
            self.proj5 = convolution_2d.Convolution2D(
                in_channels, proj5, 1, initialW=conv_init,
                initial_bias=bias_init)
            self.conv5 = convolution_2d.Convolution2D(
                proj5, out5, 5, pad=2, initialW=conv_init,
                initial_bias=bias_init)
            self.projp = convolution_2d.Convolution2D(
                in_channels, proj_pool, 1, initialW=conv_init,
                initial_bias=bias_init)

    def forward(self, x):
        out1 = self.conv1(x)
        out3 = self.conv3(relu.relu(self.proj3(x)))
        out5 = self.conv5(relu.relu(self.proj5(x)))
        pool = self.projp(max_pooling_nd.max_pooling_2d(
            x, 3, stride=1, pad=1))
        y = relu.relu(concat.concat((out1, out3, out5, pool), axis=1))
        return y
```

Listing 3.9: Inception module.

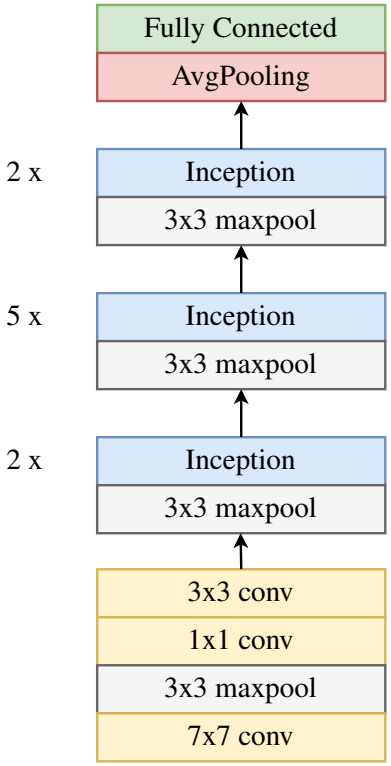


Figure 3.28: Architecture of the GoogLeNet.

Evaluation

We used Cifar10 [2] dataset to train the VGG16 model on Chainer [61]. Cifar10 dataset consists of 60K 32x32 color images in 10 classes, with 6K images of each class. Figure 3.29 shows some samples from the Cifar10 dataset. VGG reached 89.9% of accuracy which is acceptable since we did not use batch normalization, dropout, data augmentation or any other training methods to improve the final results. This accuracy will be used as the baseline for the comparison of quantization algorithms.

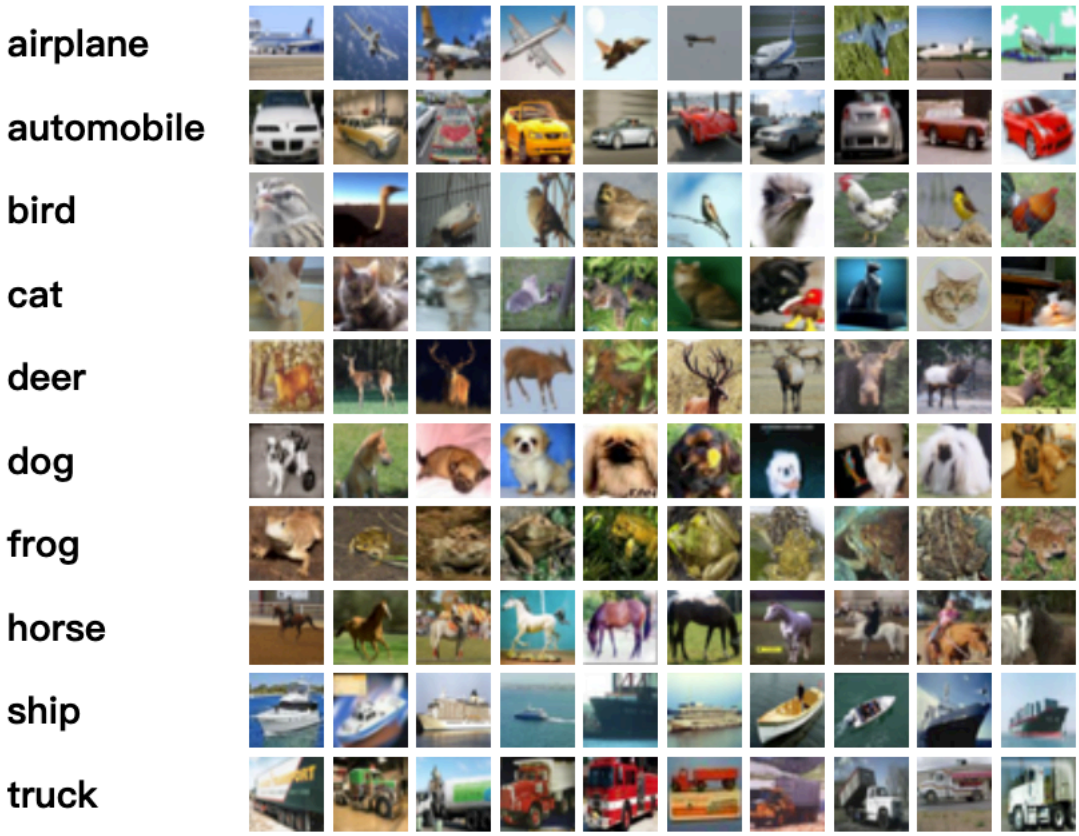


Figure 3.29: Cifar10 samples [2].

GoogLeNet with ImageNet dataset [22], however, requires hardware with strong performance and long time training. Instead, we use the off the shelf pre-trained network from caffe's model zoo. This also matches our main purpose: better utilize pre-trained networks without retraining. For large models like GoogLeNet, a retraining-free quantization algorithm could be a beneficial tool for both time and power saving. Besides, to bring back the accuracy as the original full-precision model, one needs to design the retraining algorithm. Therefore a quantization algorithm that can quantize without hurting the accuracy saves a lot of work in practical implementations.

Note the size of the first and the last layer in normal convolution networks is relatively small. Since layers of small complexity are weak in noise tolerance, Miyashita et al. and Zhou et al. [90] chose not to quantize the first layer. Another reason is that the first layer often contains only a few channels and constitutes a small proportion of total computation [90]. Thus it does not affect much without quantizing the first layer. As shown in Figure 3.30, compared with the second convolution layer, weights in layer 1 are not as concentrated as layer 2 and also feature a higher dynamic range.

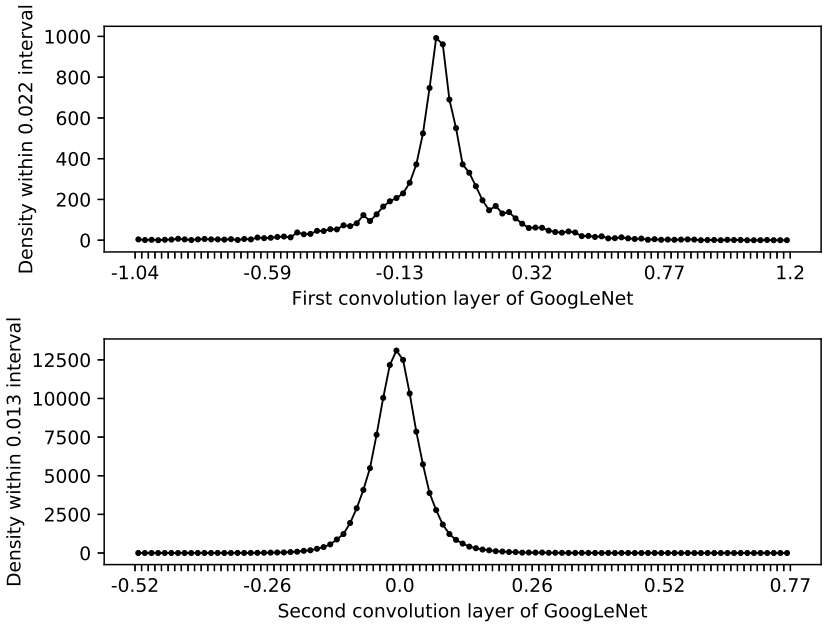


Figure 3.30: Weights distribution of the first two layers in GoogLeNet.

Rather than skipping the first layer, we quantize all layers in both VGG and GoogLeNet since it is conducive to the simplicity of forwarding computation. Moreover, quantization of small complexity layers would be helpful for seeking the performance difference between two algorithms. The distribution of quantized weights is shown in Figures 3.31 and 3.32. As we introduced in the former section, algorithm 1 penalizes all strong connections regardless of bitwidth.

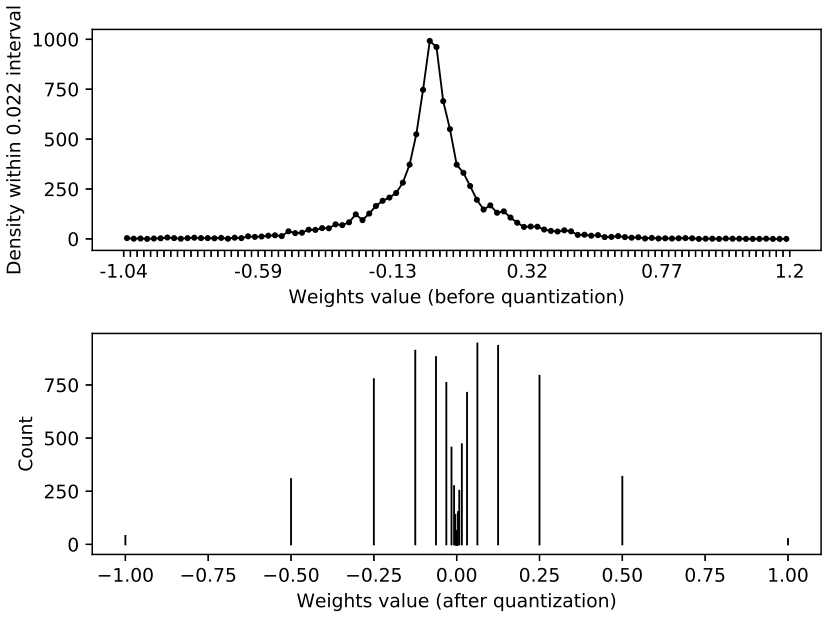


Figure 3.31: 4 bits quantization of the first layer in GoogLeNet via algorithm 1.

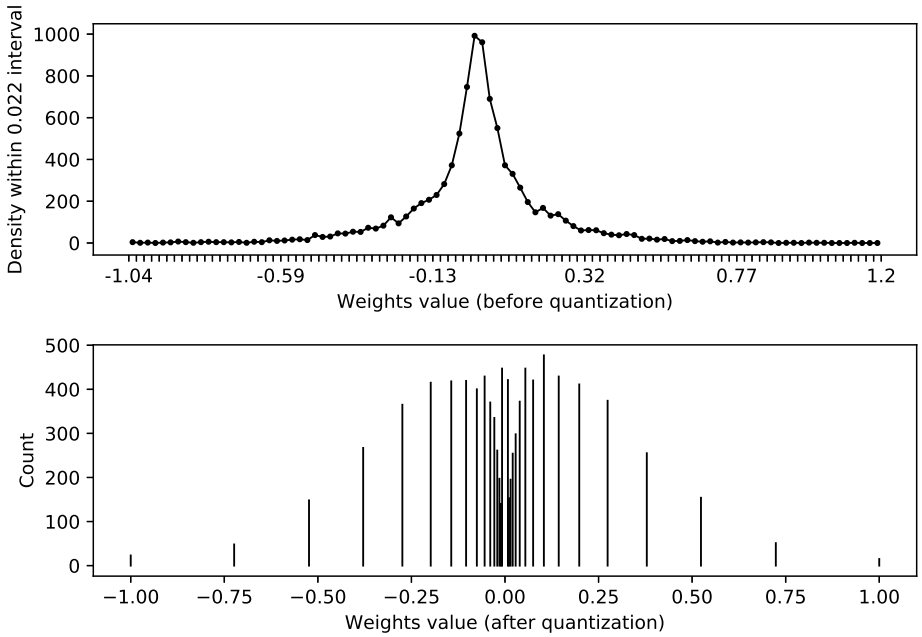


Figure 3.32: 4 bits quantization of the first layer in GoogLeNet via algorithm 2.

A tabular representation of results is shown in Table 3.6. We set FSR parameter to 1 in algorithm 1. The result shows that our algorithm is in the lead in every item. Particularly, 6 bits quantization via our algorithm causes almost no loss on GoogLeNet. Since GoogLeNet consists of only 28 megabytes of model parameters in full resolution, it is quite challenging to quantize it without retraining. To the best of our knowledge, our algorithm achieved by far the minimum accuracy loss of quantized neural networks originates from GoogLeNet. We also expect that a better pre-trained GoogLeNet can increase the accuracy of the quantized version.

Table 3.6: Benchmark on GoogLeNet and VGG (GoogLeNet on ImageNet and VGG on Cifar10)

Model	LogQuant	Decimal LogQuant	Std.
3 bits quantization			
GoogLeNet(Top-5)	0.7629	0.8021	0.8914
GoogLeNet(Top-1)	0.5143	0.560	0.6950
VGG	0.8894	0.8911	0.8993
4 bits quantization			
GoogLeNet(Top-5)	0.7593	0.8836	0.8914
GoogLeNet(Top-1)	0.5093	0.6607	0.6950
VGG	0.8942	0.8957	0.8993
6 bits quantization			
GoogLeNet(Top-5)	0.7593	0.8886	0.8914
GoogLeNet(Top-1)	0.5093	0.6914	0.6950
VGG	0.8910	0.8962	0.8993

3.5 Weight Aware Optimization Objective

As discussed in section 3.4.1, we empirically found that computing the quantization error via equation 3.8 is not appropriate since it treats all connections as equal importance. Compressing the original 32-bit parameter to 8 bits or even lower is bound to introduce quantization error, and optimizing the quantization error from quantization is a common technique in many quantization studies [40]. Many previous studies have considered minimizing the mean square error (MSE) as the goal, and have used various schemes to reduce the MSE, thus improving the accuracy of the compressed model. The results of a large number of studies indicate that this is a feasible solution.

However, in our experiments, we found that the direct optimization of MSE is not a good solution. Because the parameter distribution of well-trained neural networks tends to be bell-shaped Gaussian or Laplacian distributed [25, 37, 40], the direct optimization of MSE will result in quantization algorithms assigning higher precision to parameters with smaller values. Usually, a trained neural network has a large number of weight parameters close to zero and a very small number of parameters with higher values. A situation similar to that in Figure 3.33 is common in neural networks. Assuming that we design the quantitative algorithm with the goal of minimizing MSE, in the example of Figure 3.33 because the smaller weights are more in number, it is easy to cause the algorithm to bias the precision of the smaller weights, thus ignoring the precision of the larger weights.

$$\min E = \frac{1}{2}\sum_{i=1}^{N}\left(Q\left(w_i\right) - w_i\right)^2 \tag{3.9}$$

Many previous studies have noted this and performed some modifications for large-valued parameters in the design of the quantization algorithm from an empirical point of view, rather than optimizing the algorithm from the perspective of the optimization objective.

Value-aware Quantization.

In [91], Park et al. proposed a value-aware quantization method (V-Quant) that applies low precision quantization only on small values which occupy the majority of data and a small number of activations are kept in full-precision. This approach requires retraining to restore accuracy. After retraining, they achieved state-of-the-art accuracy on ResNet-101 and DenseNet-121 with less than a 1% top-1 accuracy drop. In V-Quant, the percentage ($1\% \sim 5\%$ reported in this paper) of values that will maintain full precision is determined empirically. Moreover, V-Quant requires sorting during training, which is time-consuming.

Hessian Based Quantization.

Hessian-based loss-aware quantization methods have also been well studied [33, 92–97]. In loss-aware quantization methods, the importance of weights is quantified using the Hessian of the weight matrices. The quantization contribution to performance loss can be approximately quantified by the Hessian-weighted distortion measure [92]. Hessian-aware methods assign the importance of each weight according to its second-order derivative, which is computationally intensive. To reduce computational consumption, these methods choose to compute the Hessian approximately. Hessian-based approaches rely on backpropagation and require retraining because of the need to compute Hessian.

We reconsidered the suitability of MSE as an optimization objective and performed some improvements to propose our new weight-aware optimization objective. We found that this method greatly im-

proves the accuracy of some existing algorithms and reduces the retraining time. We hope that this approach can be generalized and will hopefully bring better accuracy to existing algorithms.

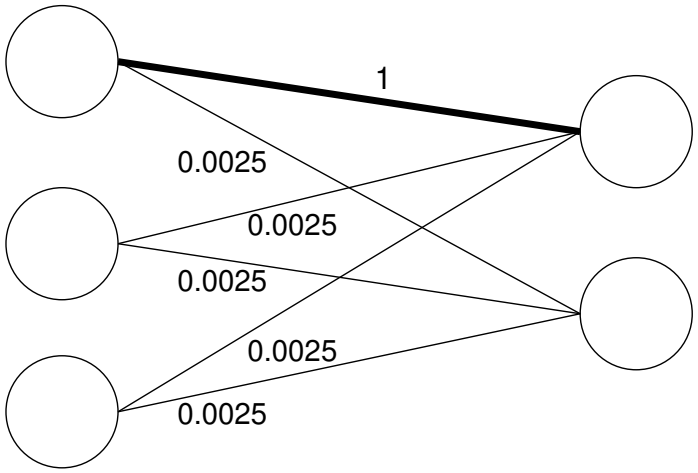


Figure 3.33: A simple case that one weight dominates the activation. Assuming the quantization levels are [0.3,0.5,0.8], minimizing the MSE will lead the quantization algorithm to assign higher precision for smaller weights.

In the following part of this section our main contributions are summarized as follows:

- We propose weight-aware optimization objective for the first time and found that this method can significantly improve some existing quantization algorithms.
- Our method requires only minimal modifications in quantization algorithms and does not require sorting of the inputs (weights), resulting in substantial savings in time and computational resources.
- We tested our method on state-of-the-art quantization algorithms. For a quantization algorithm that requires retraining, our method can significantly reduce the retraining time, and in a post-training quantization algorithm, our method achieves better classification accuracy.

3.5.1 Weight-aware minimization objective

An empirical study [25, 40, 42, 43] showed that the weight parameters in deep neural networks tend to exhibit a bell-shaped distribution such as Gaussian or Laplacian. As shown in Figure 3.34, a large number of weights are concentrated at zero values, and a very small number of weights are distributed at the tails. Empirically, the weights at the ends of the distribution are more important than the weights at the center [91].

Many quantization algorithms are designed to minimize the MSE as the optimization objective (Equation 3.9) [37, 40, 98–102]; however, we found that this is not an optimal objective. In practice, we found that this direct minimization of quantization error is not appropriate because the weights are not of equal importance. A direct minimization of Equation 3.9 can easily lead the quantization algorithm to assign higher precision to small weights. Whereas, in general, larger weights are more important. Therefore, they should be assigned higher precision in the quantization process.

To better understand the importance of weights, we conducted a pruning experiment from an empirical perspective. For some common neural networks, we pruned different weights according to the

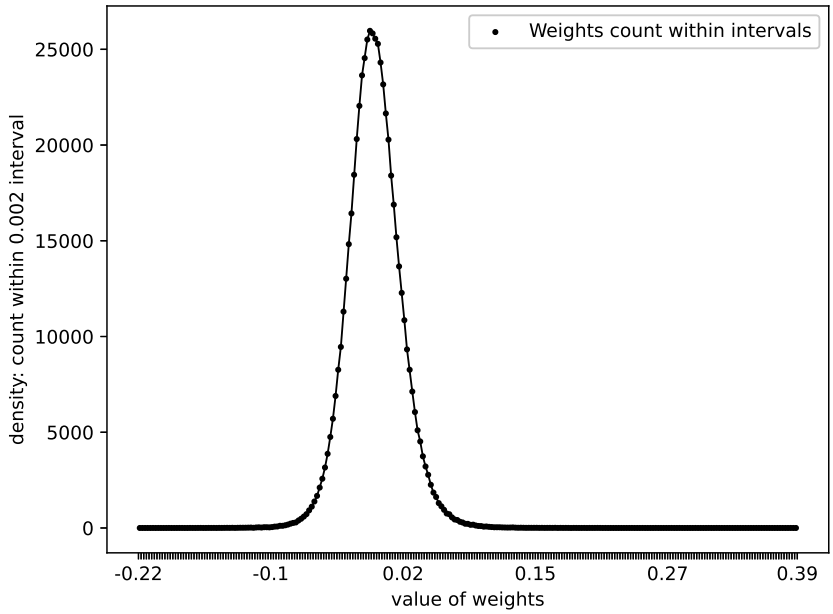


Figure 3.34: Weight distribution of a convolutional layer in VGG16. More weights in the center region and fewer weights in the tail region.

l1-norm. The weights of each layer were sorted and divided into ten groups according to the l1-norm, and pruned one group at a time. Thereafter, we performed the inference to determine the final accuracy. As per the experimental results shown in Figure 3.35, the importance of the weights of neural networks increases with the magnitude of the weight. Although, in general, this change is not linear, it shows that the larger the value is, the more important it is.

Although many previous studies have clarified the importance difference of weights, none of them have incorporated this importance difference into the quantization minimization objective. As a result, we proposed our weight-aware minimization objective (WAMO). Note that we intend to improve the existing algorithms with the weight-aware minimization objective instead of proposing a new quantization algorithm.

In the calculation of quantization error, we assign a weight to each weight parameter, and the value of this weight is exactly equal to the weight parameter itself (Equation 3.10). The calculation formula is simple and eliminates the need to sort the weights in the calculation process, thereby significantly simplifying the calculation.

$$E = \frac{1}{2} \sum_{i=1}^{N} w_i \times \left(Q\left(w_i\right) - w_i \right)^2 \tag{3.10}$$

To further elucidate our approach, we incorporate WAMO with two state-of-the-art quantization algorithms, and on both quantization algorithms, our method pushes the limits to obtain better performance. In the next two sections, we introduce these two algorithms and the two algorithms after merging with WAMO.

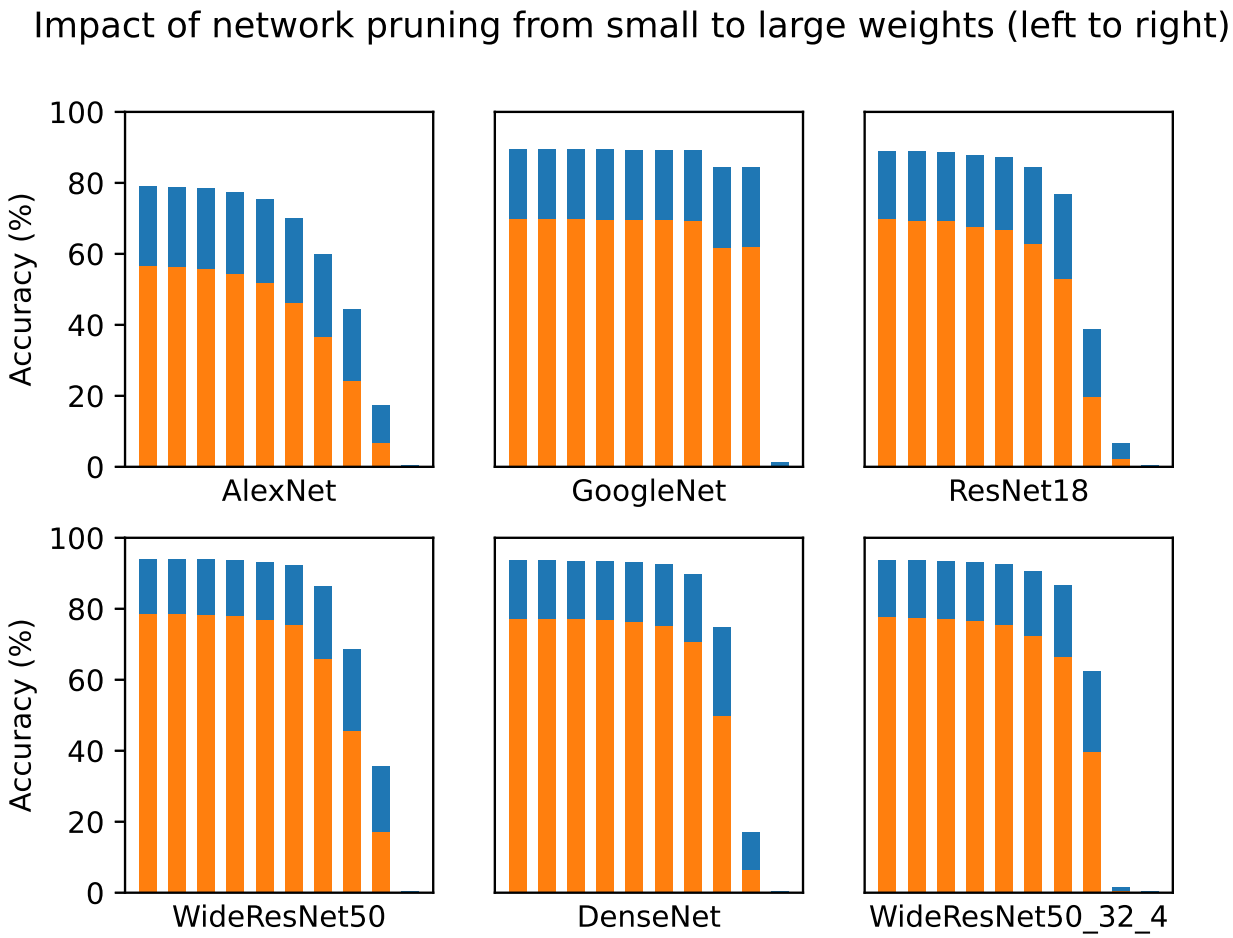


Figure 3.35: Impact of weight pruning on common neural network models (all performed on Imagenet-ILSVRC2012). For each model, the weights of each layer are divided into 10 groups according to the l1-norm and at each time one group of weights is pruned. The results displayed from left to right are the pruning test from small to large weights.

3.5.2 Weight-aware additive powers of two quantization.

The additive power of two (APoT) quantization [37] is a quantization method that constrains all quantization levels as the sum of the powers of two terms. In 2016, Miyashita et al. proposed a logarithmic quantization algorithm [35] that can convert weights in a neural network to powers-of-two representations so that expensive multiplications can be replaced by cheap shift operations. For instance, 0.06 is quantized to 2^{-4}, and -4 is then stored in the memory. Moreover, multiplication with 0.06 is then replaced by 4 bits right shift instead of using bulky digital multipliers. This approach substantially improves the inference performance [36, 103]. The quantization levels are given in Equation 3.11, where b indicates the bit width.

$$\mathcal{Q}^p(b) = \left\{0, \pm 2^{-2^{b-1}+1}, \pm 2^{-2^{b-1}+2}, \ldots, \pm 2^{-1}, \pm 1\right\} \tag{3.11}$$

The main drawback of power-of-two (PoT) quantization is that it cannot benefit from a larger bit width. In Equation 3.11, if we increase b from 4 to 5, the newly added quantization levels range from

2^{-8} to 2^{-15}, which can only increase the resolution for small weights because all other quantization levels are unchanged.

APoT quantization is proposed to solve the abovementioned problem. In APoT quantization, all quantization levels are the sum of the n PoT levels. As indicated in Equation 3.12, each quantization level is a summation of the n PoT quantization levels. Note that in Equation 3.12, the sign is omitted for simplicity. k is computed by b/n, which is the bitwidth for each additive term. By adding different quantization levels in PoT, APoT offers a balanced resolution for both small and large weights. In addition, this is achieved in an additive manner; therefore, no huge computation cost is introduced.

$$\begin{aligned} &\mathcal{Q}^a(kn) = \textstyle\sum_{i=0}^{n-1} p_i \\ &\text{where} \\ &\mathrm{p}_i \in \left\{0, \tfrac{1}{2^i}, \tfrac{1}{2^{i+n}}, \ldots, \tfrac{1}{2^{i+(2^k-2)n}}\right\} \end{aligned} \tag{3.12}$$

However, in practice, we found that APoT relies heavily on retraining because the quantization levels are set empirically, and large quantization noises are introduced during quantization. Because the APoT quantization levels are pre-set, the range of these levels cannot be dynamically adjusted during the quantization process. For example, in VGG, as illustrated in Figure 3.34, the weight range in this case is $[-0.22, 0.39]$, whereas the pre-set 4 bits APoT quantization contains useless levels such as 0.6875, 0.75, 1.0, and so forth. To solve this problem, Li et al. used weight normalization [104] to adjust the weight distribution with zero minimum and unit variance (Equation 3.13). However, in our opinion, this is not worth the gain because it increases the computing cost during network inference, while one of the biggest purposes of quantization is to speed up network inference.

$$\begin{aligned} &\tilde{\mathcal{W}} = \tfrac{\mathcal{W}-\mu}{\sigma+\epsilon}, \\ &\text{where} \\ &\mu = \tfrac{1}{I}\textstyle\sum_{i=1}^{I} \mathcal{W}_i, \sigma = \sqrt{\tfrac{1}{I}\sum_{i=1}^{I} (\mathcal{W}_i - \mu)^2}, \end{aligned} \tag{3.13}$$

With WAMO, APoT can significantly reduce the retraining time, and more excitingly, weight normalization can be removed. Instead of empirically presetting the quantization levels, weight-aware APoT (WA-ApoT) can actively choose the most suitable quantization levels during the quantization process.

We have included WA-APoT in our Algorithm 1. The idea of WA-APoT is to minimize Equation 3.10 by actively selecting the quantization levels. Given 2^{b+1} APoT quantization levels, we project the full-precision weights to the nearest quantization levels and then compute the accumulated weight aware quantization error (WAQE) for each quantization level. The top 2^b APoT quantization levels with the largest WAQE will be selected as the WA-APoT quantization levels because they contribute most to WAQE and therefore should be preserved with the best precision by maintaining the nearest quantization levels.

```
a, b, c = 0., 0., 0.
for i in range(3):
    if i < 2:
        a.append(2 ** (-2 * i - 1))
        b.append(2 ** (-2 * i - 2))
    else:
        c.append(2 ** (-2 * i - 1))
        a.append(2 ** (-2 * i - 2))
        b.append(2 ** (-2 * i - 3))
```

Algorithm 1 WA-APoT quantization levels

Input: The full precision pre-trained weight tensor W, the bit-width b of the quantized weight tensor.
Output: WA-APoT quantization levels $\mathcal{Q}^{wa}$.

1: Compute the APoT quantization levels $\mathcal{Q}^{a}$ by Equation 3.12 with bit-width equals to b+1.
2: Project all full-precision weights to the nearest quantization levels: $W \rightarrow W_q$
3: Compute accumulated weight aware quantization error E^a of each quantization level in $\mathcal{Q}^a$ by Equation 3.10.
4: Output the top 2^b quantization levels $\mathcal{Q}^{wa}$ with the largest weight aware quantization errors in E^a.

```
values = []
for i in a:
    for j in b:
        for k in c:
            values.append((i + j + k))
values = Tensor(values))
values = values.mul(1.0 / max(values))

return values
```

Listing 3.10: APoT quantizaiton level generator (5 bits) [37].

```
values=tensor([0.0000, 0.0100, 0.0200, 0.0300, 0.0400, 0.0500, 0.0600, 0.0700,
    0.0800, 0.1000, 0.1200, 0.1400, 0.1600, 0.1700, 0.2000, 0.2100, 0.2400, 0.2800,
    0.3200, 0.3400, 0.3600, 0.3800, 0.4800, 0.5200, 0.6400, 0.6500, 0.6800, 0.6900,
    0.7200, 0.7600, 0.9600, 1.0000])
```

Listing 3.11: APoT quantizaiton levels (5 bits) [37].

3.5.3 Weight-aware piecewise linear quantization.

In some scenarios, we need to use pretrained models trained on datasets that are not publicly accessible, and for those models, a post quantization algorithm is desired. Proposed by Sumsang Semiconductors in 2020, piecewise linear quantization (PWLQ) is a state-of-the-art post-training quantization algorithm that does not require retraining [40].

PWLQ is an improved uniform quantization algorithm that divides the weight distribution into several regions and then performs uniform quantization separately. Here, we first introduce a uniform quantization. Uniform quantization linearly maps all the full-precision weights to low-precision quantization levels. A b-bit uniform quantization can be defined by Equation 3.14. For each full-precision weight, uniform quantization replaces it with the nearest quantized weight in $\mathcal{Q}^u(\alpha, b)$. Note that α indicates the range of the quantization levels.

$$\begin{aligned} &\text{uni}(w, \alpha, b) = \underset{\mathcal{Q}^u(\alpha,b)}{\text{argmin}} \sum_{i=0}^{k} \sum_{q \in \mathcal{Q}^u(\alpha,b)} |w_i - q|^2 \\ &\text{where } \mathcal{Q}^u(\alpha, b) = \\ &\alpha \times \left\{0, \frac{\pm 1}{2^{b-1}-1}, \frac{\pm 2}{2^{b-1}-1}, \frac{\pm 3}{2^{b-1}-1}, \ldots, \pm 1\right\} \end{aligned} \tag{3.14}$$

To improve the accuracy after direct quantization, PWLQ uses breakpoints to divide the weights into different groups. For instance, if one breakpoint p is used and the weights range from $[-m, m]$ ($m > 0$,

assuming symmetric weight distribution), the weights are grouped into the following regions: the center region R_1: $[-p, p]$ and tail region R_2: $[-m, -p) \cup (p, m]$. The center and tail regions are then uniformly quantized separately as in Equation 3.15. In this case, if $p < \frac{m}{2}$, the center region has a smaller range and higher precision than the tail region, and conversely, if $p > \frac{m}{2}$, the tail region has a larger precision .

$$\text{pwlq}(r, b, \alpha_1, \alpha 2) = \begin{cases} \text{uni}(r, \alpha_1, b), r \in R_1 \\ \text{uni}(r, \alpha_2, b), r \in R_2 \end{cases} \tag{3.15}$$

Although multiple breakpoints can improve the performance of the model, they also introduce additional storage consumption. For a PWLQ with one breakpoint, two sets of α are obtained (Equation 3.14). For hardware implementation, these two sets of α are realized by using two accumulators, and PWLQ uses one additional bit per weight to indicate the region. Note that the extra bit is only used to determine the accumulator, and it does not increase the multiply accumulate computation (MAC). In practice, PWLQ with one breakpoint is recommended because multiple breakpoints increase hardware overhead [40].

For PWLQ, finding the optimal breakpoint p^* is the most important. In [40], p^* was set by minimizing the MSE quantization error, as shown in Equation 3.16.

$$p^* = \text{argmin}_{p \in \left(0, \frac{m}{2}\right)} \mathbb{MSE}\left(W, \mathcal{Q}^u\right) \tag{3.16}$$

For comparison, we propose a weight-aware PWLQ (WA-PWLQ). The WA-PWLQ changes only the minimization objective in Equation 3.16. The MSE quantization error was replaced with our weight-aware version. The new p^* is found by Equation 3.17, where WAE (weight aware error) is defined by Equation 3.10.

$$p^* = \text{argmin}_{p \in \left(0, \frac{m}{2}\right)} \mathbb{WAE}\left(W, \mathcal{Q}^u\right) \tag{3.17}$$

Sample codes for the two methods are given in Listing 3.12 and Listing 3.13.

```
def pwlq_quant_error(w, bits, scale_bits, abs_max, break_point):

    if overlap:
        qw_tail = ulq(w, bits=bits, scale_bits=scale_bits, minv=-abs_max, maxv=
    abs_max) # ulq: uniform linear quantization.
        qw_middle = ulq(w, bits=bits, scale_bits=scale_bits, minv=-break_point, maxv
    =break_point)

        qw = torch.where(-break_point < w, qw_middle, qw_tail)
        qw = torch.where(break_point > w, qw, qw_tail)
    else:
        qw_tail_neg = ulq(w, bits=bits, scale_bits=scale_bits, minv=-abs_max, maxv=-
    break_point)
        qw_tail_pos = ulq(w, bits=bits, scale_bits=scale_bits, minv=break_point,
    maxv=abs_max)
        qw_middle = ulq(w, bits=bits-1, scale_bits=scale_bits, minv=-break_point,
    maxv=break_point)

        qw = torch.where(-break_point < w, qw_middle, qw_tail_neg)
```

```
        qw = torch.where(break_point > w, qw, qw_tail_pos)

    # PWLQ error minimization:
    err = torch.sqrt(torch.sum(torch.mul(qw - w, qw - w)))
    return err, qw
```

Listing 3.12: PWLQ quantization logic.

```
def pwlq_quant_error(w, bits, scale_bits, abs_max, break_point):

    ...

    # WA-PWLQ's weight aware error minimization objective:
    err = torch.sqrt(torch.sum(torch.mul(torch.abs(w),torch.mul(qw-w, qw-w))))
    return err, qw
```

Listing 3.13: WA-PWLQ's weight aware quantization error minimization objective.

3.5.4 Experiments

In this section, we evaluate our weight-aware minimization objective on the aforementioned two quantization algorithms: APoT and PWLQ. Note that our goal is not to propose another state-of-the-art quantization algorithm, but to improve existing quantization algorithms.

ResNet and EfficientNet

(1) ResNet

ResNet [105], surpasses GoogLeNet in achieving further improvements on the ImageNet dataset. With the advent of ResNet, neural networks that were usually considered difficult to train (e.g. 1000 layers) in the past can be trained effectively. ResNet proposed residual learning (as shown in Figure 3.36). By shortcut connecting the layers as Figure 3.36, in extreme cases, the output of the residual module can exactly replicate the output of the previous layer, thus not causing network performance degradation by increasing the number of network layers.

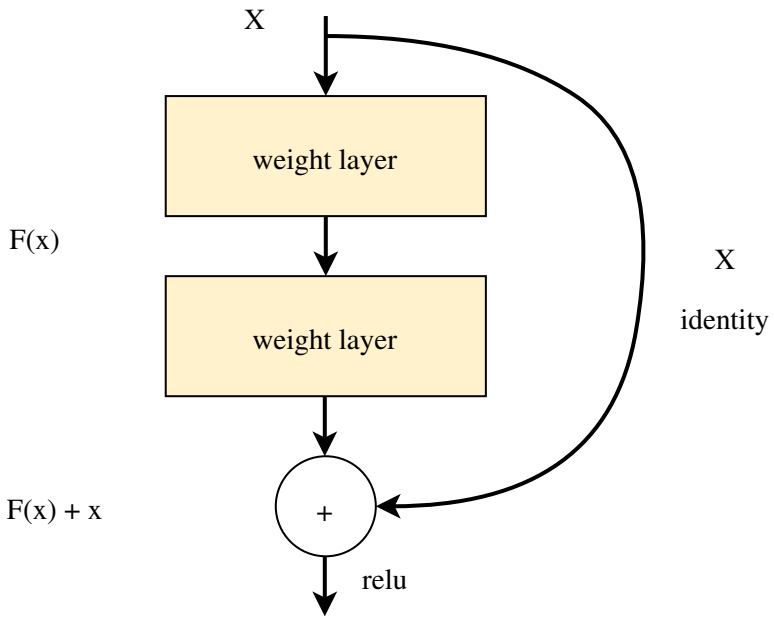


Figure 3.36: Residual unit of ResNet.

A typical ResNet module is defined as Listing 3.14. Equation 3.18 indicates the computation of a residual module. When backpropagate the gradient from the loss function, according to equation 3.20, the 1 in parentheses indicates that the gradient can be short-circuited to the previous layer as well.

```
class ResModule(link.Chain):

    def __init__(self, in_channels, mid_channels):
        super(ResModule, self).__init__()
        with self.init_scope():
            self.conv1 = Convolution2D(in_channels, mid_channels, 1, 1, 0,)
            self.bn1 = BatchNormalization(mid_channels)
            self.conv2 = Convolution2D(mid_channels, mid_channels, 3, 1, 1)
            self.bn2 = BatchNormalization(mid_channels)
            self.conv3 = Convolution2D(mid_channels, in_channels, 1, 1, 0)
            self.bn3 = BatchNormalization(in_channels)

    def forward(self, x):
        h = relu(self.bn1(self.conv1(x)))
        h = relu(self.bn2(self.conv2(h)))
        h = self.bn3(self.conv3(h))

        # shortcut connection
        return relu(h + x)
```

Listing 3.14: Sample code of a ResNet module.

$$\begin{aligned} y_l &= h\left(x_l\right) + F\left(x_l, W_l\right) \\ x_{l+1} &= f\left(y_l\right) \end{aligned} \tag{3.18}$$

$$x_L = x_l + \sum_{i=l}^{L-1} F\left(x_i, W_i\right) \tag{3.19}$$

$$\frac{\partial \text{ loss}}{\partial x_l} = \frac{\partial \text{ loss}}{\partial x_L} \cdot \frac{\partial x_L}{\partial x_l} = \frac{\partial \text{ loss}}{\partial x_L} \cdot \left(1 + \frac{\partial}{\partial x_l} \sum_{i=l}^{L-1} F\left(x_i, W_i\right)\right) \tag{3.20}$$

ResNet-50	**ResNet-101**
$7 \times 7, 64$, stride 2	
3×3 max pool, stride 2	
$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix} \times 3$
$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1 \times 1, 128 \\ 3 \times 3, 128 \\ 1 \times 1, 512 \end{bmatrix} \times 4$
$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1 \times 1, 256 \\ 3 \times 3, 256 \\ 1 \times 1, 1024 \end{bmatrix} \times 23$
$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1 \times 1, 512 \\ 3 \times 3, 512 \\ 1 \times 1, 2048 \end{bmatrix} \times 3$
average pooling, fc layer, softmax	

Table 3.7: Network configuration of ResNet-50 and ResNet-101.

(2) EfficientNet

Unlike ResNet or GoogLeNet in the previous section, which relies on experience-based design. EfficientNets are products of neural network search techniques. Google's team first design a baseline network called EfficientNet-B0, then this baseline network is scaled up by adjusting its width, depth and resolution. To test our quantization method, we chose the baseline EfficientNet-B0 with the fewest parameters so that the challenge would be greater when going through quantization compression. Table 3.8 summarizes the configuration of EfficientNet-B0. As a comparison, EfficientNet-B0 has only 5.3 million parameters while ResNet-50 has 26 million ($4.9\times$). With such a gap, both can achieve similar top5 accuracy on ImageNet.

Module	Resolution $H \times W$	Channels	Layers
Conv3x3	224×224	32	1
MBConv1, k3x3	112×112	16	1
MBConv6, k3x3	112×112	24	2
MBConv6, k5x5	56×56	40	2
MBConv6, k3x3	28×28	80	3
MBConv6, k5x5	14×14	112	3
MBConv6, k5x5	14×14	192	4
MBConv6, k3x3	7×7	320	1
Conv1x1 & Pooling & FC	7×7	1280	1

Table 3.8: Configuration of EfficientNet-B0

WA-APoT

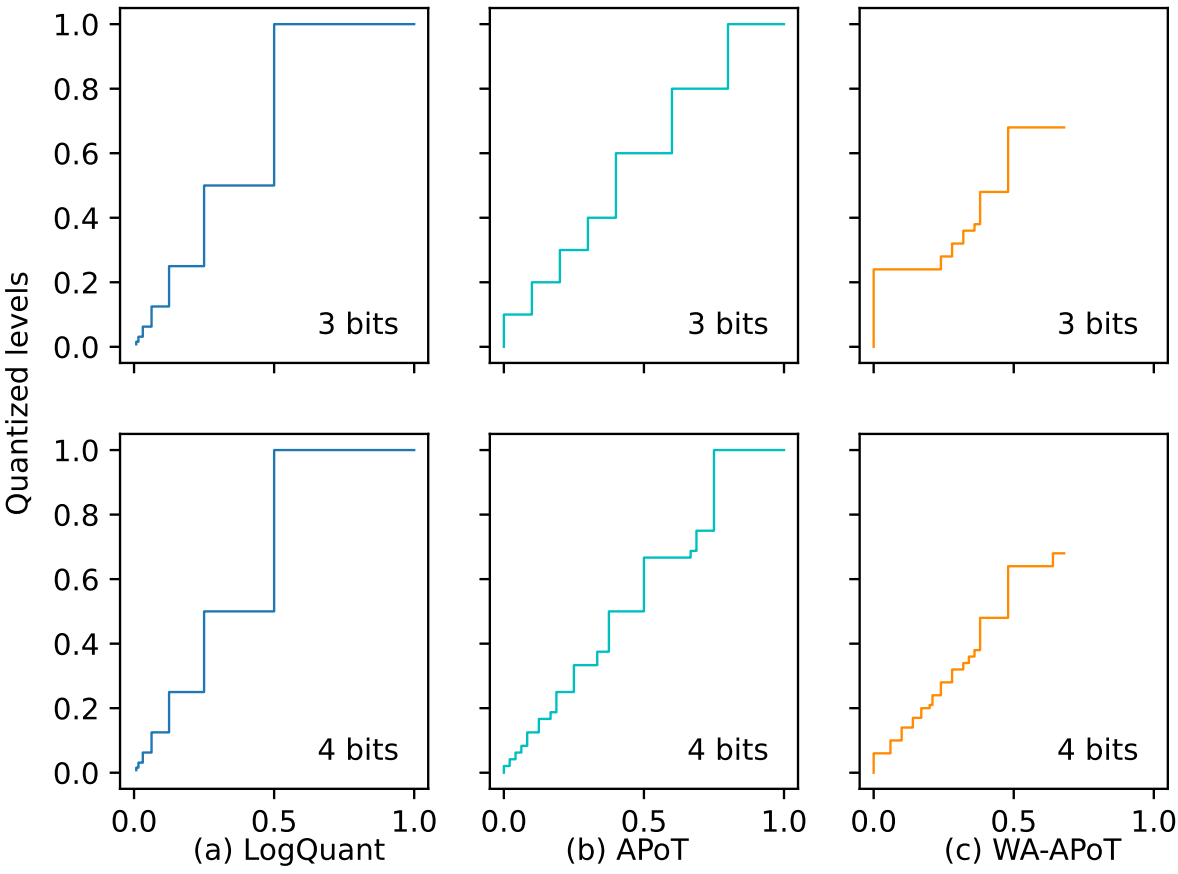


Figure 3.37: Comparison of quantization levels of LogQuant, APoT and WA-APoT (ResNet-20 on Cifar10). WA-APoT has a more reasonable resolution and eliminates the empirical quantization levels settlments.

We used Cifar-10 to test the effectiveness of WA-APoT. To ensure the accuracy of the test results, we used the same pre-trained ResNet-20 network as APoT [37]. As we introduced in Section 4, we removed the weight normalization because it would bring an extra computational burden. Without weight normalization, the APoT suffers a huge accuracy loss when the quantization bit width is less than 4.

First, we compared the quantization levels for ResNet-20 on Cifar-10. As shown in Figure 3.37, when 3 bits quantization is adopted, WA-APoT assigns most of the precision to weights near 0.3. For logarithmic quantization, the difference between the 3-bits and 4 bits quantization is not visible. With the increase in bitwidth, the increased bitwidth is assigned to quantization levels near 0 by logarithmic quantization; therefore, the quantization levels for larger weights are the same for 3-and 4 bits quantization (Equation 3.11).

APoT solves the low-resolution problem of large weights after quantization, and it can be clearly seen in Figure 3.37 that the resolution of large weights increased significantly as compared to the logarithmic quantization. However, for different neural networks or different layers of the same network, the weights are fixed when the number of bitwidths is given, which results in poor adaptability of APoT and requires the use of weight normalization to enforce the weights of each layer to obey the standard distribution.

In contrast, WA-APoT assigns quantization levels adaptively through the autonomous perception of the importance of weights via Algorithm 1. In this resent20 example, we can see that larger weights are assigned more quantization levels. Interestingly, in the case of 3 bits, the weights that are close to 0 are all dropped (quantized to 0), which is different from the PoT and APoT quantizations; however, it is in line with our previous conjecture, that is, large weights are more important than small weights, and weights that are close to 0 will have low or no impact even if they are pruned.

Figure 3.38 shows the results of this test. Compared with APoT, WA-APoT has two obvious advantages. (1) Whether it is 3 bits or 4 bits quantization, WA-APoT can quickly restore the original performance of the pre-trained model. In the case of Cifar-10, WA-APoT can restore the original performance of the pre-trained model after only one epoch of retraining, which greatly reduces the fine-tuning time and computational resource consumption as compared to APOT. (2) In the absence of weight normalization, WA-APoT can completely restore the original performance of the model, whereas APOT suffers accuracy loss in the case of three bits. The restriction that the APOT requires weight normalization is lifted on WA-APoT.

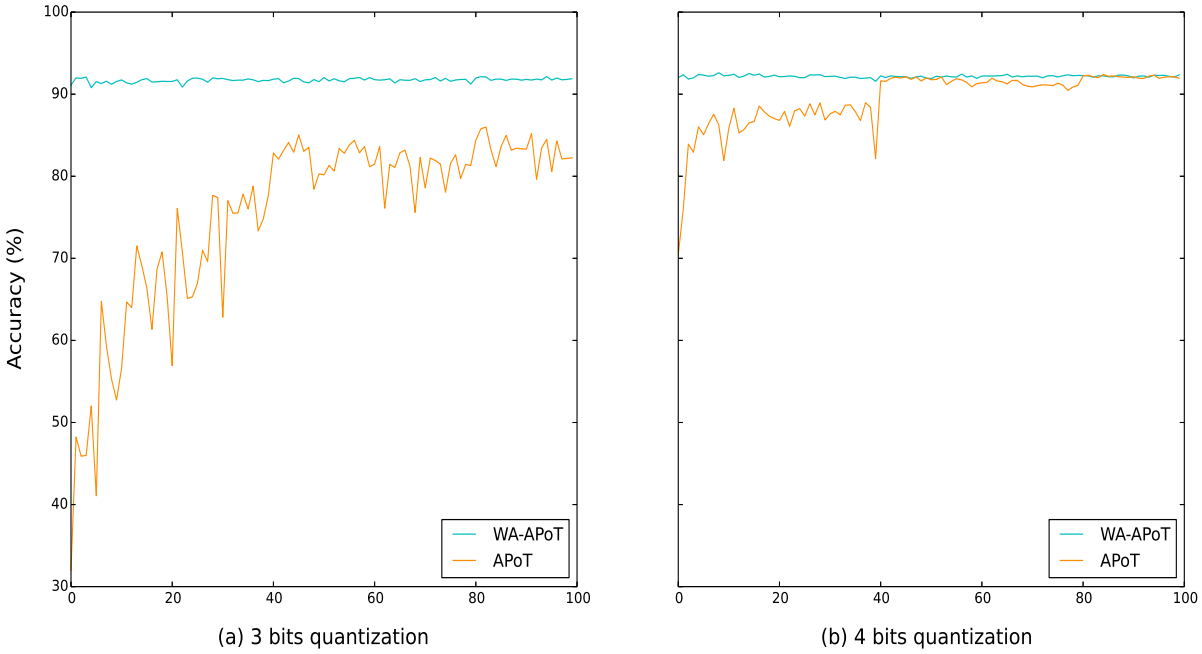


Figure 3.38: Retraining curves of APoT and WA-APoT (ResNet-20 on Cifar10). On both 3 bits and 4 bits quantization, WA-APoT can restore the original performance within one epoch. In contrast, APoT requires a much longer training time. Especially in the case of 3 bits quantization, APoT cannot restore the original performance in the absence of weight normalization.

WA-PWLQ

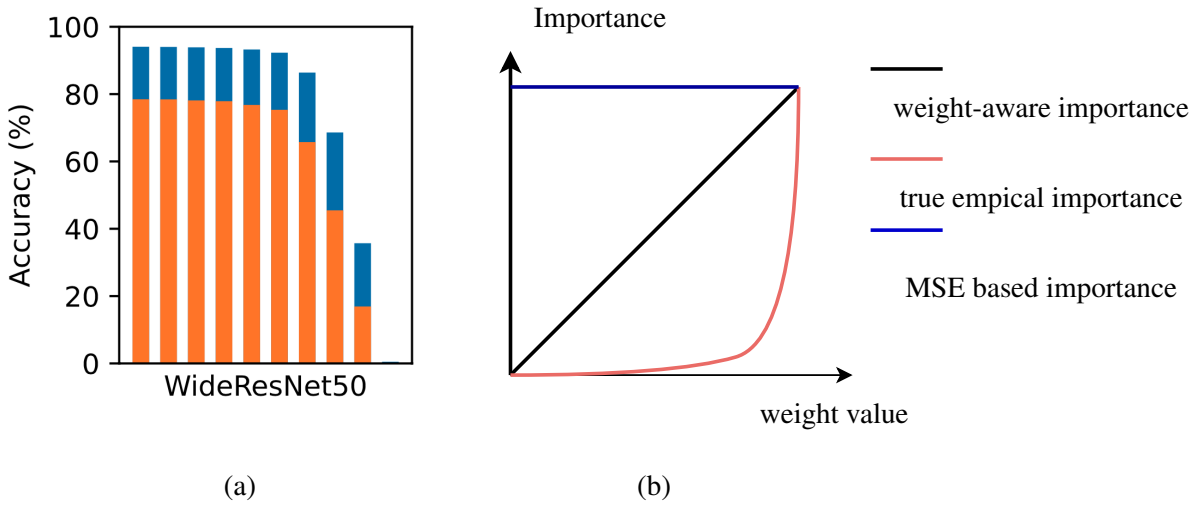
The Imagenet-ILSVRC2012 [22] dataset was used to validate the effectiveness of our weight-aware quantization optimization objective on post-training image classification tasks. As explained in Section 4, we only changed MSE to WAE during the quantization error minimization step in PWLQ to form WA-PWLQ. Moreover, we used one breakpoint because multiple breakpoints increase the hardware overhead.

We first tested two quantization methods on the widely used ResNet-50 and ResNet-101 [105]. Note that the neural network models are pretrained models provided by PyTorch's torchivision module [106]. The test results are summarized in Table 3.9. For both ResNet-50 and ResNet-101, our WA-PWLQ is superior to the PWLQ, especially for 4 bitwidth quantization. Both PWLQ and WA-PWLQ assign larger precision to tail regions; however, WA-PWLQ maintains the tail region in a narrower range than PWLO. For instance, on ResNet-50 $4 \sim 5\%$, the tail region is divided by WA-PWLQ compared with approximately 10% by PWLQ. It is clear that the piecewise linear quantization method can achieve good performance, mainly because it assigns larger precision for larger weights. WA-PWLQ tends to narrow the large weight percentage and the result that $4 \sim 5\%$ of larger weights are assigned larger precision is very close to [91]'s empirical percentage ($1 \sim 5\%$).

We also used EfficientNet-b0 [107] to test the performance of the two methods under extreme conditions. EfficientNet-b0 has only 5.4M parameters, which is already $4.9\times$ smaller than that of ResNet-50. With 4 bits quantization, WA-PWLQ leads PWLQ by a large margin.

Table 3.9: Comparison of PWLQ and WA-PWLQ on Top-1 and Top-5 Imagenet-ILSVRC2012 classification accuracy (%).

Network	Weight Bit-width	4-bit	6-bit	8-bit
ResNet-50	PWLQ (Top1)	74.650	75.920	76.014
$(76.102(top1))$	WA-PWLQ (Top1)	74.868	75.804	76.038
$(92.934(top5))$	PWLQ (Top5)	91.966	92.898	92.954
	WA-PWLQ (Top5)	92.150	92.794	92.932
ResNet-101	PWLQ (Top1)	75.968	79.168	79.298
$(79.210(top1))$	WA-PWLQ (Top1)	76.758	79.082	79.266
$(94.556(top5))$	PWLQ (Top5)	92.968	94.516	94.510
	WA-PWLQ (Top5)	93.332	94.528	94.524
EfficientNet-b0	PWLQ (Top1)	44.972	72.148	73.562
$(73.636(top1))$	WA-PWLQ (Top1)	56.666	72.616	73.662
$(91.316(top5))$	PWLQ (Top5)	68.114	90.496	91.296
	WA-PWLQ (Top5)	79.126	90.688	91.284



(a) (b)

Figure 3.39: Comparison of weight importance assignments of different strategies. The red curve in subfigure (b) is an empirical importance fitting curve for subfigure (a).

3.5.5 Future Works of Quantization

In the future, we plan to research a new weight aware optimization objective that can actively adjust the importance of each weight according to the distribution of the weights.

As shown in Figure 3.39, compared with vanilla MSE based methods, assigning each weight with importance that linear to its magnitude is more realistic. However, this strategy also does not fit the true importance curve very well. As indicated by the red line, the importance of the weights does not change linearly, but jumps up when the weights reach a certain level. In our tests, the weight importance curves of different networks are not the same, but without exception, they all show a nonlinear importance change. We have tried to nonlinearize the importance during our experiments ($w \times$ MSE to $w^2 \times$ MSE

for instance), but this change did not bring much difference in performance. The next step we want to do is to do an adaptive weight importance assignment on different neural networks to better fit the weight importance curve of that network, so as to protect the information of the original neural network to a greater extent when quantizing.

RISC-V Core for Quantized Neural Networks

We also set a goal to implement RISC-V cores for quantized neural networks. Since RISC-V is highly modifiable and open source, we can build a many cores system for low bitwidth computations. In addition, a many cores system can offer scalable computing resources for neural networks. We would like to develop learning methods for discontinuous weights and design a RISC-V core for these special purpose algorithms.

Chapter 4

Trigonometric inference

4.1 trigonometric inference

As neural networks increase in size, the training of the neural network becomes more dependent on specialized hardware consisting of thousands of computing units. General matrix multiply (GEMM) is at the core of deep learning because a tremendous amount of matrix multiplication operations are required for neural network training. To accelerate the training and inference of deep neural networks, hardware vendors are constantly increasing the chip area and the number of processing units. Nevertheless, accelerating matrix multiplication operations remains cost-ineffective, in contrast to accelerating simple operations, such as addition, subtraction, and a bit shift.

In terms of hardware implementation, multipliers are bulky and power-intensive compared to other logic resources. Consequently, this prohibits deployment to scenarios when the multipliers are insufficient. For instance, digital signal processing (DSP) used for floating-point multiplication is often found to be a scarce resource when attempting to implement neural networks on field-programmable gate arrays (FPGAs). To reduce the usage of hardware multipliers, many researchers use a coordinate rotation digital computer (CORDIC) [48]) module to compute layer activations rather than the direct usage of DSPs on FPGAs when hyperbolic activation functions are applied. This method involves only additions, subtractions, a bit shift, and look-up tables, and has been confirmed to be more hardware efficient [52, 108, 109]. Although widely used in microcontrollers and FPGAs, CORDIC can only partially eliminate dependence on multipliers, and a large number of multipliers are still required for inference and error backpropagation, which hinders on-FPGA neural network training.

Efforts had been put into hardware optimizations. Many optimization possibilities are motivated by the various inherent characteristics of the CNN models. For instance, CNN's spatially correlated characteristics, i.e., adjacent output feature map activations will share close values per feature map, motivated Shomron et al. to propose a value prediction based method that reduces MAC operations [110] in DNNs. In [13], a lightweight CNN is implemented to predict zero-valued activations. The prediction step is calculated prior to the convolution step, thus the convolution operations can be largely saved. In addition, the sparsity of DNN values causes the underutilization of the underlying hardware. For example, DNN tensors usually follow a bell-shaped distribution that is concentrated on zero, therefore a high percentage of values will only be represented by a portion of least-significant bits (LSBs). These zero-valued bits may cause inefficiencies when executed on hardware. To address it, a non-blocking simultaneous

multithreading (NB-SMT) method [111] was designed to better utilize the execution resources. Many quantization algorithms have also been proposed to reduce the usage of multipliers in DNNs. For example, to quantize all network weights into powers of 2, one can avoid multiplications [103]. However, the calculation of weight parameters directly involved only accounts for a part of the neural network training. If we further quantize the layer inputs, we can reduce the use of multipliers to a greater extent, which will cause a huge accuracy loss and is only suitable for simple tasks [112]. For circumstances in which hardware multipliers are insufficient, we propose a trigonometric approach to eliminate the dependence on multipliers.

Consider a simple case in which only fully connected layers are used, and the non-linearity is omitted. As shown in Figure 1.4, to back propagate error signals, each layer conducts multiplications between the weight matrix and error matrix from the last layer (see subfigure (b)). In addition, to update the weights, the dot product of the activation and error matrix is computed (see subfigure (c)). It is clear that multiplications between errors, activations, and weights dominate the computation cost. However, in modern neural network models, both the weight parameters and error signals are highly concentrated around zero with an extremely small variance. This characteristic enables us to approximate the original error and weight using its sine value. Then, using the product to sum formula, we can convert the multiplications to easier operations. Here, we briefly introduce the ideas behind this study.

$$\lim_{x \to 0} \frac{\sin x}{x} = 1 \tag{4.1}$$

$$\sin\alpha \sin\beta = -\frac{1}{2}[\cos(\alpha + \beta) - \cos(\alpha - \beta)] \tag{4.2}$$

From Equation (4.1), we know that when x is infinitesimal, we regard $\sin(x)$ and x as equivalent. Moreover, we have $\sin(x) \approx x$ when x is a small value, and, thus, we can approximate x as $\sin(x)$. For example, the error gap is within $1.67e^{-7}$ when $|x|$ is smaller than 0.01. The upper subfigure of Figure 4.1 shows a typical distribution of weight parameters extracted from a random layer of a 28-layer WideResNet trained on the CIFAR-100 dataset. The bottom subfigure of Figure 4.1 shows the error curve when we replace x with $\sin(x)$. Apparently, using the sine value as an approximation does not add much noise to the network when the parameters are highly concentrated at near 0.

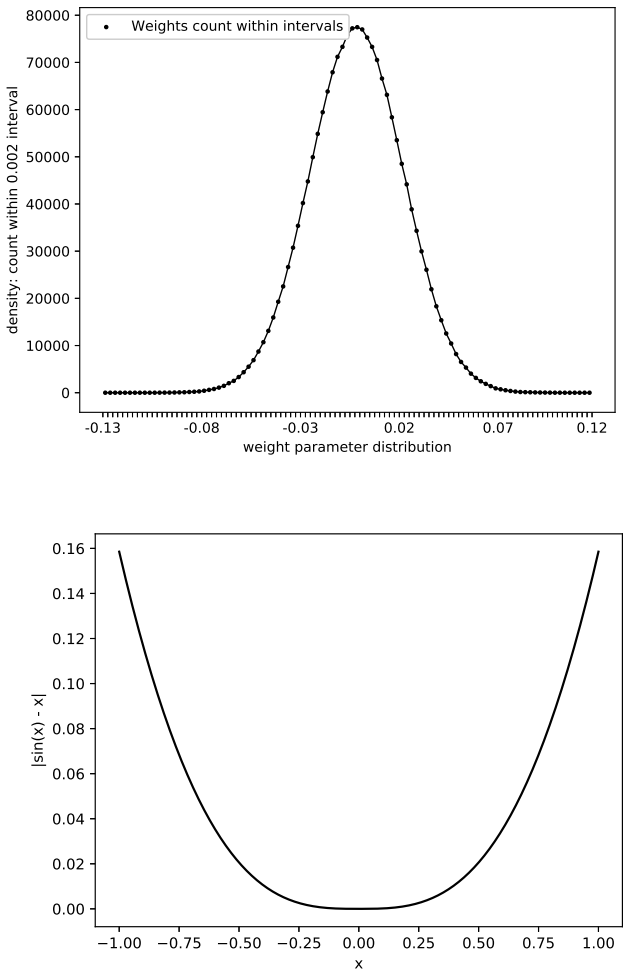


Figure 4.1: Weight distribution and approximation error curve. The upper subfigure shows the distribution of weight parameters extracted from a random layer of a 28-layer WideResNet trained on the CIFAR-100 dataset. The bottom subfigure is drawn from $|\sin(x) - x|$, which denotes the approximation error.

By applying Equation (4.2), i.e., the product to sum formula, we can convert multiplications of sine values into simpler addition, subtraction, and bit shift operations, which are far more economical than classical multiplication operations. Note that the cosine value can be computed by the aforementioned CORDIC engine, which is also hardware friendly and only requires simple operations. We also introduce a sine-based activation function. Rather than a mere sine activation, we adopt a rectified variant that combines the ReLU [113] and sine activation. In this way, we realize the sine value replacement of activation, error, and weight, between which the multiplications are removable in inference and training. We call this method trigonometric inference.

Trigonometric inference offers an alternative training method when the hardware multipliers are insufficient. In addition, the method is superior when a hyperbolic function, such as $tanh(x)$, is adopted as the activation function. The approach is evaluated on image classification tasks, and the experimental results confirm a performance comparable to that of the classical training method.

In this section we will cover the following points:

- We propose a novel training and inference method that utilizes trigonometric approximations.

To the best of our knowledge, this is the first work that shows trigonometric inference can provide learning in deep neural networks;

- By replacing the model parameters and activations with their sine value, we analyze that multiplications could be transferred to shift-and-add operations in training and inference. To achieve this, a rectified sine activation function is proposed;
- We evaluate trigonometric inference on several models and show that it can achieve performance that is close to conventional CNNs on MNIST, Cifar10, and Cifar100 datasets.

4.1.1 Cordic Introduction

Since we will be using trigonometric operations instead of the original multiplication operations, here we introduce some of the underlying operational mechanisms of trigonometric functions. When calculating trigonometric functions, an accurate method is to expand the function in Taylor series and then calculate the result of the first N terms of the series to approximate the true value (equation 4.4 and 4.4).

$$\sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 + o\left(x^5\right) \tag{4.3}$$

$$\sin(1) \approx 1 - \frac{1}{2!}1^3 + \frac{1}{5!}1^5 = 1 - \frac{1}{6} + \frac{1}{120} = \frac{101}{120} \tag{4.4}$$

However, this method involves too much multiplication and is therefore very hardware intensive. In contrast, the CORDIC algorithm is a lookup table based method.

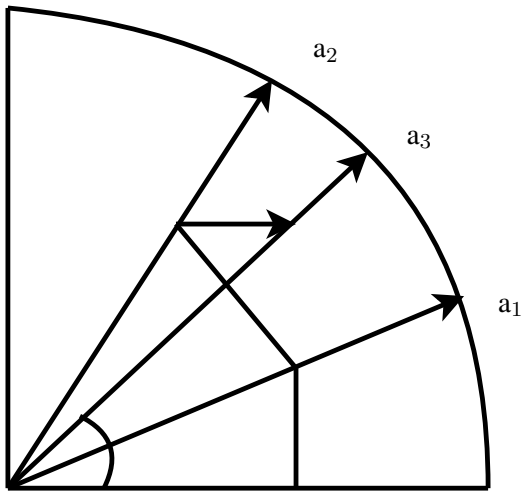


Figure 4.2: Cordic rotation illustration.

As shown in Figure 4.2, assuming that the Figure is a quarter unit circle, what we want to require is the angle formed by a_3 and the bottom edge. Note for a unit circle if we know the coordinate of a certain angle then the sine and cosine of that angle are solved. The problem is that it is very difficult to find the angle of a_3 so we can decompose the angle into combinations of other known angles, for example here the angle a_3 can be decomposed into combinations of three angles as shown in the figure. At first, Volder was inspired by the formula as equation 4.5 with $K_n = \sqrt{1 + 2^{-2n}}, \tan(\varphi) = 2^{-n}$.

$$\begin{aligned} K_n R\sin(\theta \pm \varphi) &= R\sin(\theta) \pm 2^{-n} R\cos(\theta) \\ K_n R\cos(\theta \pm \varphi) &= R\cos(\theta) \mp 2^{-n} R\sin(\theta) \end{aligned} \tag{4.5}$$

In practice many elementary angles are precomputed and saved in a lookup table. By iteratively accumulate precomputed micro-angles the target angle can be approximated. As shown in equation 4.6, the computation strategy is consists of two parts: rotation and scaling [114]). Note in equation 4.6 a is the precomputed angle and d denoted rotation direction.

$$\begin{cases} x^{(i+1)} = x_s^{(i)} + d_i y_s^{(i)} 2^{-i} \\ y^{(i+1)} = y_s^{(i)} + d_i x_s^{(i)} 2^{-i} \\ z^{(i+1)} = z^{(i)} - d_i a^{(i)} \end{cases} \\ \begin{cases} x_s^{(i+1)} = S_i x^{(i+1)} \\ y_s^{(i+1)} = S_i y^{(i+1)} \end{cases} \tag{4.6}$$

4.2 Methodology

In this section, we elaborate on our trigonometric inference learning algorithm. For ease of description, we explain the algorithm based on fully connected neural networks. In convolutional models, the background idea is the same, and the only change is to replace multiplication marks with convolution marks.

4.2.1 Forward and Backward Inference

Forward Inference

Let w denotes the model weights and a activations of an arbitrary layer. Note that we use subscript j to indicate a middle layer, and i denotes the previous layer of j. Equation (4.7) provides the standard forward inference and Equation (4.8) indicates the activation calculation of our proposed trigonometric inference.

$$a_j = \sum_j w_j \sigma(a_i) + b_j \tag{4.7}$$

$$a_j = \sum_j \sin(w_j)\, \sigma_{sine}(a_i) + b_j \tag{4.8}$$

where $\sigma(x_j)$ is an activation function and $\sigma_{sine}(x_j)$ is our newly proposed sine-based activation function for trigonometric inferences. A sine function has been studied as an activation function in previous studies. In 1999, Sopena et al. [115] found that a multiplayer perceptron with sine as the activation learns faster than one with sigmoid activation on certain tasks. As a significant difference, the sine is periodic, whereas common activation functions are monotonic. Parascandolo et al. [116] found that sine activation functions perform reasonably well on current datasets, and the network ignores the periodic nature of sine functions. After training, only the central region at near zero of the sine activation function was used. Ramachandran et al. [117] used automatic search techniques to discover new activation functions. They then found that $min(x, \sin(x))$ activation can achieve a performance comparable to that of ReLU

for image classification. Zhang et al. [118] proposed a sine-activated algorithm for a multi-input function approximation that can efficiently obtain a relatively optimal structure.

Based on previous findings, herein we propose our sine-based activation. Rather than directly applying the activation function as $sin(x)$, we combine the ReLU activation and sine activation because this combination has two merits: First, ReLU excludes negative outputs and, therefore, the computational burden is considerably reduced. In addition, regulating the activations to only positive values can lessen the poor influence caused by the periodicity property of the sine activation. In Section 4.3, we show that this rectified sine activation performs much better than a pure sine activation for trigonometric training. Equation (4.9) gives our activation function, and from Figure 4.3, we know that $sin(x)$ and $tanh(x)$ are extremely similar for small values.

$$\sigma_{sine} = \begin{cases} 0, & \text{if } x < 0 \\ \sin(x), & \text{else} \end{cases} \tag{4.9}$$

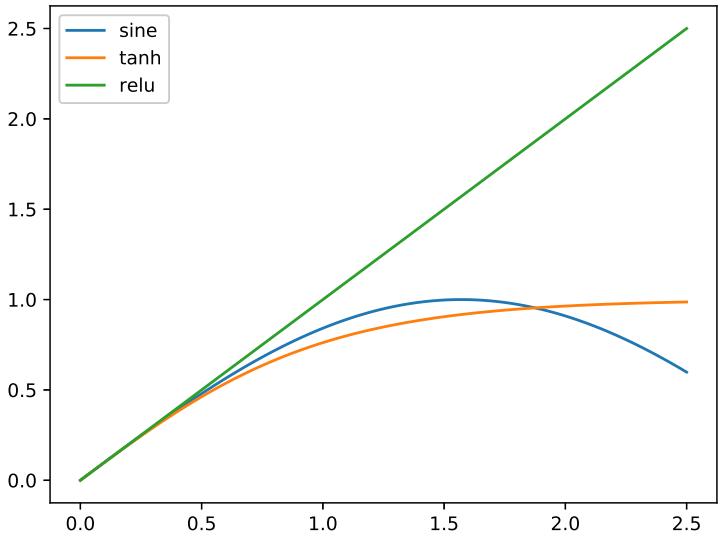


Figure 4.3: Activation functions. The sine activation behaves similarly to the hyperbolic tangent activation for $|x| < \pi/2$.

Backward Inference

When training a neural network, the error E calculated from the loss function is backpropagated to each layer (error backpropagation) for weight update. The weights in each layer are then updated for narrowing the error E. This parameter optimization problem can be solved using gradient descent, which requires calculating $\frac{\partial E}{\partial w}$ for all w in the model. Here we denoted E_j as the backpropagated error signal from the latter layer to the layer j. According to the chain rule, we can use Equation (4.10) to calculate the error signal E_i at layer i, which is backpropagated from layer j. For conventional CNNs, the error E_i is calculated by $E_j \frac{\partial a_i}{\partial w_i}$, which equals the multiplication of E_j and x_i ($\frac{\partial a_i}{\partial w_i} = x_i$).

$$E_i = \frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial a_i}\frac{\partial a_i}{\partial w_i} = E_j \frac{\partial a_i}{\partial w_i} \tag{4.10}$$

For our trigonometric inference, E_i is calculated differently since we use $sine(x)$ and $sine(W)$ to approximate x and W, respectively, in the forward pass (as Equation (4.8)). By taking the partial derivative of a_i over w_i, we get $\frac{\partial a_j}{\partial w_i} = sin(a_i)cos(w_i)$. Therefore, the error E_i in trigonometric inference should be calculated as Equation (4.11) when the activation is larger than zero (Equation (4.9)).

$$E_i = E_j \frac{\partial a_i}{\partial w_i} = E_j sin(a_i)cos(w_i) \tag{4.11}$$

Equation (4.11) gives the precise error signal $\frac{\partial E}{\partial w_i}$ for weight update in the middle layer i during gradient descent. However, the direct calculation of Equation (4.11) will incur multiplication overheads, so here we developed two methods for computing E_i. As shown in Table 4.1, the mean and variance of the weights and errors are very small in a downsized VGG of eight convolutional layers. This property, which makes the approximation possible, is common in modern deep neural networks. Experimental results show the trade-off between these two approaches; here, we outline the computation details:

(1) Equation (4.11) is divided into two parts and approximately computed. Equation (4.12) indicates a recursive approximation computation (E_j is approximated by $sin(E_j)$). In this way, multipliers are not required, and precise error signals are calculated despite the computation complexity being increased.

$$\begin{aligned} T_j &= \sin\left(E_j\right) \sin\left(a_i\right) \\ E_i &= \sin\left(T_j\right) \cos\left(w_i\right) \end{aligned} \tag{4.12}$$

(2) First, we replace E_j with $sin(E_j)$ in Equation (4.11). The above approximation strategy enables us to compute gradients using Equation (4.13).

$$\frac{\partial E}{\partial w_i} = \sin(E_i) \sin(a_i) \cos(w_i) \tag{4.13}$$

Second, we use a pseudo-error signal for backpropagation. Because the cosine term in Equation (4.13) adds extra computational complexity, we replace it with shift operations. Since the values of the weights are generally highly concentrated around the value of zero, in an ideal situation we can simply omit the cosine term ($cos(x) \approx 1$ if $x \to 0$). However, in practice, we found that omitting the cosine term directly sometimes causes an excessive error, which is contributed by those weights of larger magnitude, so we replace the cosine term with 1/2. By simply scaling the original error signals to half, the algorithm provided learning in our experiments. The half-scaling strategy is formulated as in Equation (4.14): A similar concept was found in feedback alignment methods [75, 119] for error backpropagation. Lillicrap et al. [75] discovered that the weights do not have to be symmetric in the forward and backward inferences during the training. In his research, Arild Nøkland [76] provided a theoretical analysis that supports the learning capabilities of backpropagation with pseudo-error signals.

$$\frac{\partial E}{\partial w_i} = \sin(E_i) \sin(a_i)/2 \tag{4.14}$$

Weight Update

For conventional CNNs, the learning rate (η in Equation (4.15)) can be set to an arbitrary value within a certain range. This may introduce an additional multiplicative burden when performing gradient descent calculations.

$$W_{new} = W_{old} - \eta * \frac{\partial(E)}{\partial\left(W_{old}\right)} \tag{4.15}$$

To avoid divisions, shift operations may be used instead when the learning rate is set to powers of 2. In this way, the divisions are converted into bit shifts.

$$W_{new} = W_{old} - 2^n * \frac{\partial(E)}{\partial\left(W_{old}\right)} \tag{4.16}$$

4.2.2 Precision Analysis

Our training algorithm performs several approximations to significantly decrease the usage of multipliers. It omits the cosine term when the value is very small and recursively approximates small values by their sine values (Equation (4.14)). Here, we analyze the precision loss of the approximation strategy.

By applying the binomial theorem, we rewrite product xy as Equation (4.17) for comparison. By the Taylor theorem, $sin(x)sin(y)$ is expanded to its polynomial form (Equation (4.18)).

$$\left[\left(1 - \frac{(x-y)^2}{2}\right) - \left(1 - \frac{(x+y)^2}{2}\right)\right]/2 \tag{4.17}$$

$$\begin{aligned}&\left[\left(1 - \frac{(x-y)^2}{2} + R_n(x-y)\right) - \right.\\ &\left.\left(1 - \frac{(x+y)^2}{2} + R_n(x+y)\right)\right]/2\end{aligned} \tag{4.18}$$

where Rn(x) denotes the Lagrange form of the remainder:

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x)^{n+1} \tag{4.19}$$

The difference of Equations (4.17) and (4.18) gives the error gap of our xy-to-$\sin(x)\sin(y)$ approximation. Because the $f^{(n+1)}(\xi)$ term is still trigonometric, the Lagrange remainders in Equation (4.18) are then bounded by:

$$R_n(x \pm y) < \frac{(x \pm y)^3}{6} \tag{4.20}$$

We substitute Equation (4.20) into $(R_n(x-y) - R_n(x+y))/2$ and through simple unequal relations, we have the error of our approximation upper bounded by:

$$\frac{(|x| + |y|)^3}{6} \tag{4.21}$$

Equation (4.21) indicates a fast error decay if the ℓ_1-norm between x and y is less than 1, which is a loose constraint for modern deep neural network models. As shown in Table 4.1, all convolutional layers of a downsized VGG easily meet the above conditions.

Table 4.1: Mean and variance of convolutional layer inputs (x), weights (W), and back propagated errors (E) from the first training iteration in a downsized VGG network. Note all these mean and variance values are calculated from all x, W, E in each layer. The CIFAR-10 dataset was used, and the network contained eight convolutional layers. All mean and variance values for the weights and errors are extremely small, which enables the sine approximation of the original values.

	Layer 1	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**	**Layer 6**	**Layer 7**	**Layer 8**
μ(x)	0.3985	0.3994	0.4015	0.8437	0.4036	0.6533	0.4101	0.3984
σ(x)	0.3411	0.3411	0.3376	0.5188	0.3360	0.4836	0.3312	0.3410
μ(W)	-3.589×10^{-5}	-6.064×10^{-5}	2.882×10^{-5}	1.087×10^{-5}	-1.145×10^{-4}	3.681×10^{-4}	-1.203×10^{-4}	-3.569×10^{-5}
σ(W)	8.704×10^{-4}	8.680×10^{-4}	8.691×10^{-4}	1.737×10^{-3}	1.735×10^{-3}	3.461×10^{-3}	3.481×10^{-3}	8.703×10^{-4}
μ(E)	6.497×10^{-14}	1.624×10^{-13}	-1.021×10^{-13}	-1.160×10^{-13}	$-4.269e\times10^{-13}$	$-3.434e\times10^{-13}$	-2.413×10^{-13}	1.114×10^{-13}
σ(E)	7.959×10^{-9}	1.490×10^{-8}	3.014×10^{-8}	4.779×10^{-8}	6.241×10^{-8}	1.195×10^{-07}	1.756×10^{-07}	8.560×10^{-09}

4.2.3 A Toy Example

To better illustrate how the learning algorithm works, we let a simple model fit a straight line. Consider a model of only two neurons, each forming a layer, and we have a model that outputs $W_2(W_1(x))$ in which the non-linearity is omitted. Using the trigonometric training method, we trained this model with different choices of parameter initials, input ranges, and target line slopes. If the target line is denoted as $y = 0.3x$, the goal is to train the model that will be stable at $\sin(w_1)\sin(w_2) = 0.3$. Figure 4.4 summarizes the fitting results and learning dynamics of the proposed algorithm under various conditions. The parameters for the different conditions are summarized in Table 4.2. In this case, the learning rate is chosen as 2^{-2} for avoiding multipliers in the gradient descent stage (note 2^{-2} is not required, other approximate values can work as well), and the learning dynamics subplots are drawn from the first 100 iterations. In addition, the training sample size is 300, and the final fitting curves are the results of 100 epochs of training.

Table 4.2: Initial parameters of the toy example. Note that m stands for the target slope, and $\mathcal{N}_x$ denotes the mean and variance of the samples. All samples were normally distributed.

Initials	**a**	**b**	**c**	**d**
w_1	0.5	0.43	-0.78	0
w_2	0.1	500	-0.07	1.83
$\mathcal{N}_x$	(0.03, 0.3)	(0.3, 0.3)	(0.03, 0.3)	(0.03, 2)
m	0.07	-0.61	1.73	-0.54

It is clear that the gradient descent functions when Equation (4.21) is satisfied. When the target slope is larger than 1 (see subfigure (c,g)) or sample data are drawn with large variance (subfigure (d,h)), the proposed algorithm struggles to reach the target; however, samples around 0 are still well approximated (subfigure (d)). Fortunately, in modern deep neural networks, these situations are not common and can be avoided. In addition, even if w_2 was set to 500 at the beginning, the algorithm still provided learning. Because trigonometric functions are periodic, a weight that is too large will not affect the convergence of the network.

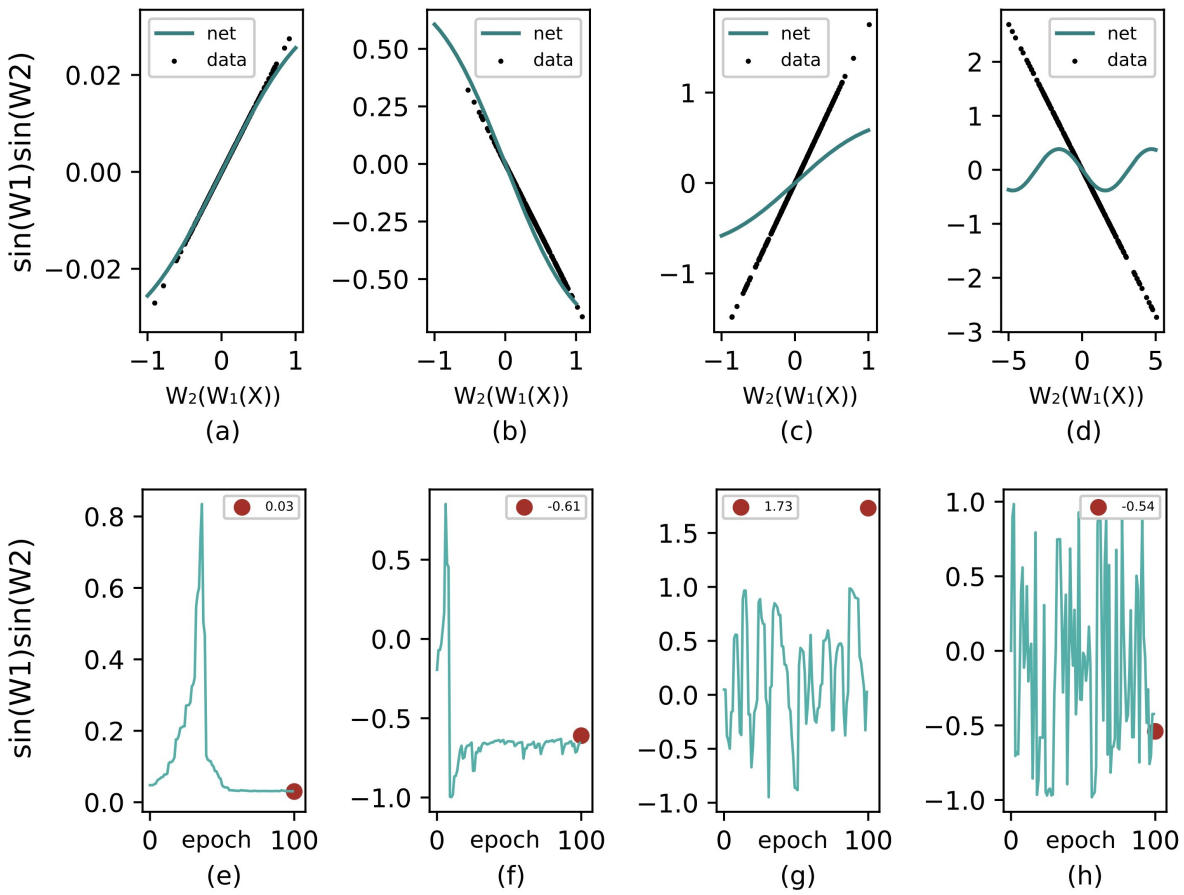


Figure 4.4: Line fitting and learning dynamics of trigonometric inference under various conditions. The initial parameters for (**a**–**d**) are listed in Table 4.2. (**a**,**e**) represent an ideal condition in which samples are concentrated around 0. (**b**,**f**) represents a situation in which samples are drawn from comparatively large means. (**c**,**g**) depict the learning dynamics when the slope is larger than 1, which is impossible for sine activation to reach. (**d**,**h**) designate a large sample variance.

4.2.4 Suitable Scenarios

Trigonometric inference allows forward and backward computations without using multipliers. It involves only simple shift-and-add operations to compute trigonometric functions when the CORDIC [48] algorithm is employed. Trigonometric inference is preferred when hardware multipliers are insufficient. Although ReLU is widely used owing to its simplicity and computational ease, many neural network models rely on hyperbolic activation functions. In FPGAs, where hardware multipliers are extremely limited resources, the CORDIC module is a more efficient alternative for computing hyperbolic activation functions [54, 120]. When CORDIC modules are utilized for computing activations, a trigonometric inference is preferred because it does not require extra multipliers. By contrast, a classical training approach still requires a large number of multiplications for inference and training, even if CORDIC modules are used for activation calculations.

4.3 Evaluation

In this section, we use the CIFAR and MNIST datasets to test the effectiveness of the proposed learning algorithm. All computations were conducted on the CUDA devices. Note that in CUDA devices, our algorithm will have no speed gains because CUDA devices (see cuDNN [58]) and modern machine learning frameworks are specially optimized for matrix multiplication operations. We used Chainer [61] to construct the learning system and neural network models. All experiments were conducted on NVIDIA Tesla P100 and GTX1660 Super GPUs.

4.3.1 Experiments on MNIST

MNIST is a dataset used for handwritten digit classification. On MNIST, even a small neural network can achieve a good accuracy. By using a small neural network, we can easily adjust the model shapes to investigate the performance of a trigonometric inference under various situations. The MNIST images were 32×32 in size, and only fully connected layers were used to build the neural net model. In this case, we discuss the training conditions suitable for the algorithm and test the performance for different sine activations. We used a three-layer neural network and tested the convergence of the network by changing the number of intermediate neurons. The results are presented in Table 4.3. We use Momentum SGD to optimize the models with momentum set to 0.9 and a learning rate of 0.01. The batch size was set to 128, and no data augmentation was used. For the standard model, ReLU was used as the activation function, and as the final accuracy of the pure sine activation and the proposed rectified sine activation.

As the results summarized in Table 4.3 indicate, it is noticeable that trigonometric inference fails to provide learning when the last layer size is inadequate. As the reason for this phenomenon, when the last layer size is too small, the error assigned to each connection will be too large, thus reducing the precision of the sine approximation. For comparison, Net_3 confirms that trigonometric inference functions when the last layer size is sufficient. It also shows a performance decrease when small layers are adopted in the model. We also tested a model with each layer having 10,000 neurons to check the difference in performance between the proposed algorithm and the classical inference. For small neural net models with large layer sizes, the two algorithms show almost no difference, and interestingly, trigonometric inference outperforms the baseline inference method.

Rows Acc_{trig_sine} and $ACC_{trig\_rectsine}$ in Table 4.3 represent the validation results from a network with a full sine activation and a rectified sine activation, respectively. By comparing their final accuracy, it is clear that the proposed rectified sine activation outperforms the full sine activation by a wide margin.

Table 4.3: MNIST test accuracy.

Model	$\boldsymbol{Net_1}$	$\boldsymbol{Net_2}$	$\boldsymbol{Net_3}$	$\boldsymbol{Net_4}$
layer 1	1000	1000	100	10,000
layer 2	1000	1000	100	10,000
layer 3	1000	100	1000	10,000
Acc_{std_relu}	0.9827	0.9832	0.9802	0.9833
Acc_{trig_sine}	0.9239	0.1087	0.9221	0.9241
$Acc_{trig\_rectsine}$	0.9832	0.1153	0.9289	0.9836

4.3.2 Experiments on CIFAR

We then tested our method on the CIFAR-10 and CIFAR-100 datasets, each containing 60,000 $32 \times 32 \times 3$ RGB images. We trained a downsized VGG model [84] and a 22-layer WideResNet [121] on the CIFAR-10 and CIFAR-100 datasets. Note that Equation (4.12) is used for backpropagation for TrigInf 1 (method 1), and for TigInf 2 (method 2), Equation (4.14) is adopted. The configurations of the downsized VGG and WideResNet are shown in Table 4.4. For VGG, the model width was much smaller than that of the original VGG-11. By decreasing the layer width, we can observe the learning results of trigonometric inference when the network size is relatively small. We trained the model using momentum SGD with the momentum set to 2^{-4} and decayed to half for every 25 epochs. The same training method is also used on WideResNet, in which the momentum decays every 40 epochs. The MNIST test indicates that a layer that is too narrow has a negative influence on the overall performance. Therefore, on WideResNet, we avoid using a trigonometric inference on the first layer because the number of filters is too small, which will cause performance deterioration. We conducted trigonometric inference only on convolutional layers because they contribute the most computational burden. The batch norm was also utilized to accelerate the convergence, and all tasks were trained for 300 epochs with data augmentation. The batch size was set to 128. For comparison, we trained VGG and WideResNet without approximations. ReLU was used as the activation function for comparison groups. We chose VGG because it is representative of the traditional CNNs. As for why we chose wide resnet over resnet, it is because we found that wide resnet performs better on Cifar-100 after several rounds of testing. Compared with Cifar-10, training on Cifar-100 is much more difficult because there are only 600 images per category. We used resnet at first, but neither CNN nor TrigInf could obtain an accuracy close to 80% without transfer learning. In comparison, wide resnet can achieve a higher accuracy of approximately 78%.

Table 4.4: Network configuration of the downsized VGG and the 22-layer WideResNet.

VGG	**WideResNet**
$[3 \times 3, 64] \times 2$	$[3 \times 3, 16]$
$[3 \times 3, 128] \times 2$	$\begin{bmatrix} 3 \times 3, 16 \times 10 \\ 3 \times 3, 16 \times 10 \end{bmatrix} \times 3$
$[3 \times 3, 256] \times 4$	$\begin{bmatrix} 3 \times 3, 32 \times 10 \\ 3 \times 3, 32 \times 10 \end{bmatrix} \times 3$
1024-linear $\times 2$	$\begin{bmatrix} 3 \times 3, 64 \times 10 \\ 3 \times 3, 64 \times 10 \end{bmatrix} \times 3$
	$[7 \times 7]$ avg-pool
	640-linear

Table 4.5 summarizes the classification results on two datasets. The training accuracy and loss curves of WideResNet are illustrated in Figure 4.5. As Figure 4.5 implies, trigonometric inference provides a close performance to standard convolution counterparts on both CIFAR-10 and CIFAR-100. TriInf 1, which calculates more precise error signals, achieves higher accuracy and the final results decrease by approximately 1%. Although the accuracy deteriorates by approximately 8% on CIFAR-100, coarse error signals (TrigInf 2) are still recommended on simple tasks because a similar performance is provided on CIFAR-10 and the computations are easier.

Table 4.5: Final validation accuracy of CIFAR-10 and CIFAR-100.

Model and Dataset	CIFAR 10	CIFAR 100
VGG Stdandard	0.9309	0.7158
VGG TrigInf 1	0.9192	0.7009
VGG TrigInf 2	0.9213	0.6957
WRN Stdandard	0.9564	0.7861
WRN TrigInf 1	0.9537	0.7759
WRN TrigInf 2	0.9334	0.7053

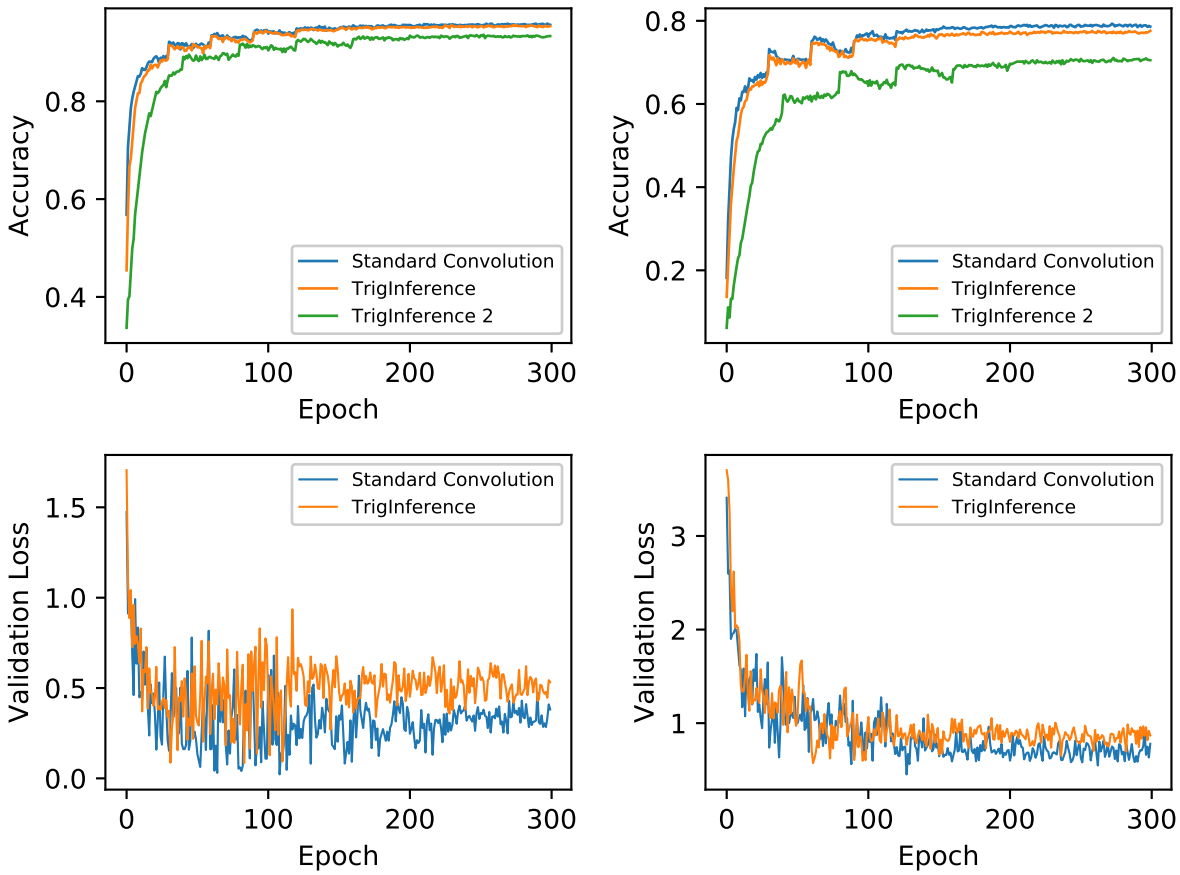


Figure 4.5: Training curves of WideResNet on CIFAR-10 and CIFAR-100. The left subplots denote the validation accuracy and loss records on CIFAR-10 and the right subplots denote those on CIFAR-100.

4.3.3 Narrowing the Accuracy Loss

As Equation (4.21) indicates, the approximation error of trigonometric inference is upper bounded by: $\frac{(|x|+|W|)^3}{6}$ Where x denotes layer inputs and W weights. This approximation error is negligible if the ℓ_1 norm between x and W is less than 1. Although most weights and activations in DNNs are very small, there are still a few large ones that affect the final performance.

Batch normalization is used to overcome the internal covariate shift problem [57,122]. As the training progresses, the weights in the network are constantly updated. On the one hand, when the parameters

in the underlying network changes, these changes are amplified as the number of layers deepens due to the linear and nonlinear mapping between layers; on the other hand, the parameter changes cause the input distribution to change constantly in each layer, and the network in the upper layers needs to adapt to these distribution changes in real time, which makes the converge difficult. Batch normalization is proved can reduce the internal covariate shift effect, and accelerate training (equation 4.22).

For conventional CNNs the mean of BatchNorm layer is usually set to 1, which is problematic for trigonometric inference since it will bring larger approximation errors.

$$\begin{aligned} \mu_{\mathcal{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma_{\mathcal{B}}^2 &= \frac{1}{m}\sum_{i=1}^{m}\left(x_i-\mu_{\mathcal{B}}\right)^2 \\ \hat{x}_i &= \frac{x_i-\mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2+\epsilon}} \\ y_i &= \gamma\hat{x}_i+\beta=\mathrm{BN}_{\gamma,\beta}\left(x_i\right) \end{aligned} \tag{4.22}$$

A simple trick to compensate for the accuracy loss is the enforcement of small activations in the batch normalization layers, which introduces no extra computation burdens. The sample code is given in Listing 4.1. To demonstrate this, we did experiments to observe the effect of the mean parameter of batch normalization layers.

```
def Batchnorm_simple_for_train(x, beta, eps, momentum, cache, gamma=0.1):

    mean=x.mean()
    var=x.var()

    # compute moving mean and var
    mvg_mean = momentum * cache.mvg_mean + (1 - momentum) * mean
    mvg_var = momentum * cache.mvg_var + (1 - momentum) * var

    # normalization
    x_normalized=(x - mvg_mean)/sqrt(mvg_var + eps)

    # This step scales the output. Originally it is set 0, here we force it to a
    small value to ensure the success of trigonometric inference.
    x_new = gamma * x_normalized + beta

    cache['mvg_mean'] = mvg_mean
    cache['mvg_var'] = mvg_var

    return x_new, cache
```

Listing 4.1: Our batch normalization layer, where we enforce small scaling (small gamma) so that the approximation error of trigonometric inference is negligible.

We trained ResNet50 models on Cifar-100 dataset, and Momentum SGD was used for gradient descent. For the conventional CNN, we set the mean of BN layers to the default value 1. For TrigInf models, the mean values of BN layers are set to 0.1, 0.5, and 0.7, respectively. We trained each model for 200 epochs (with learning rate decayed to half at (60, 100, 140, 180) epochs) and the results are summarized in Table 4.7. We can clearly see that the same accuracy and validation loss as the baseline CNN can be obtained by reducing the mean parameter of BN layers.

Table 4.6: Evaluation results on Cifar-100 with BN layers' mean set to different values.

Model (ResNet50) and Result	Accuracy	Training Loss	Validation Loss
CNN ($\mu(BN) = 1.0$)	0.7171	0.0582	1.0614
TrigInf ($\mu(BN) = 0.7$)	0.7058	0.0701	1.2853
TrigInf ($\mu(BN) = 0.5$)	0.7124	0.0644	1.6003
TrigInf ($\mu(BN) = 0.1$)	0.7205	0.0577	0.9063

4.3.4 Compatibility with Other Optimizers

Optimizers

(1) SGD

The simplest optimizer is SGD (stochastic gradient descent), which is used for our explanation in previous sections. The SGD is defined as equation 4.23. A vanilla SGD is implemented as Listing 4.2.

$$\theta = \theta - \eta \cdot \nabla J(\theta) \tag{4.23}$$

```
for i in range (epoch):
    gd = grad( loss , data , weights )
    weights = weights - lr * gd
```

Listing 4.2: Vanilla SGD optimizer.

(2) Momentum SGD

Momentum SGD remembers gradients from the last iteration, and updates weights according to not only the current gradient, but gradients of last iteration as well. The update rule is shown in equation 4.24. The formula shows that if the gradient of this time is the same as the last time, the gradient of this time is strengthened, and on the contrary, the gradient is weakened (as in Figure 4.6).

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{4.24}$$

```
for i in range (epoch):
    gd = grad( loss , data , weights )
    weights = momentum + weights - lr * gd
```

Listing 4.3: Momentum SGD optimizer.

(3) Nesterov Optimizer

Nesterov is a way to update parameters with a predictive nature. It 'looks ahead' and knows that in the next update iteration the weight would be updated to $\theta - \gamma v_{t-1}$, then the next 'weight' is incorporated into the current iteration. In this way, the optimizer can prevent the gradient descent from going too fast.

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J\left(\theta - \gamma v_{t-1}\right) \\ \theta &= \theta - v_t \end{aligned} \tag{4.25}$$

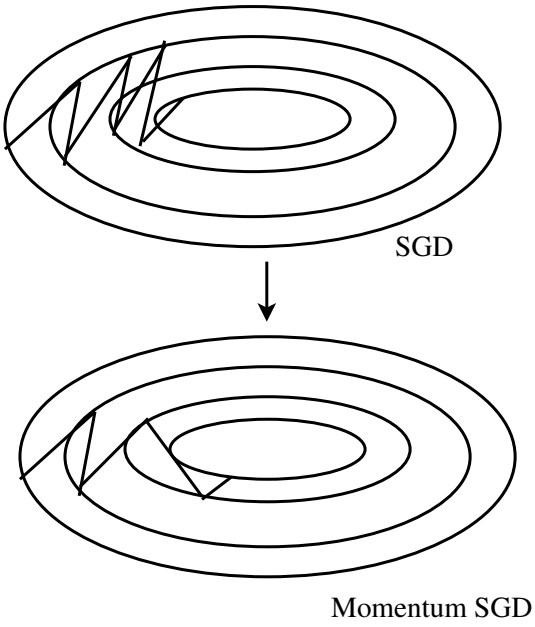


Figure 4.6: Comparison of Momentum SGD and SGD.

```
for i in range (epoch):
    gd = grad( loss , data , weights )
    weights = momentum + weights - lr * gd_ahead
```

Listing 4.4: Nesterov optimizer.

(4) Adagrad

Adagrad offers an adaptive learning rate for each weight in the update iteration. As shown in equation 4.5, for each parameter θ_i, the learning rate is computed with attention to other weights in the layer. Specifically, the $G_{t,i}$ is squared sum of the past gradient of θ_i. In this way the optimizer updates each weight based on its past gradients, and assigns larger gradients it the weight moves too slow in the past. The ϵ parameter in the optimizer is for avoiding dividing by 0.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \tag{4.26}$$

```
for i in range (epoch):
    gd = grad( loss , data , weights )
    weights = weights - lr * gd / (sqrt(gd_cache) + eps)
```

Listing 4.5: Adagrad optimizer.

(5) Adam

Adam (adaptive moment estimation) [8] also computes gradient adaptively for each weight. The differences are twofold. 1) Adam places more importance on the most recent part of the historical gradient than on treating past gradients as equally important. 2) Adam does not only refer to squared gradients, but to gradients as well. The update logic is as follows:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1)\, g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2)\, g_t^2 \end{aligned} \tag{4.27}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{4.28}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t \tag{4.29}$$

```
for i in range (epoch):
    m = beta1*m + (1-beta1)*gd
    v = beta2*v + (1-beta2)*(gd**2)
    weights = weights - lr * m / (sqrt(v) + eps)
```

Listing 4.6: Adam optimizer.

The trigonometric inference is compatible with other gradient descent optimizers such as AdaDelta, Adam, Nesterov Accelerated Gradient (NesterovAG), etc. To show this, we evaluated trigonometric inference on different gradient descent optimizers. Apart from SGD, we tested Adam and NesterovAG with ResNet50 and GoogleNet on Cifar-100, respectively. Figure 4.7 demonstrates the training curves and validation loss descent curves. As illustrated in Figure 4.7, on both optimizers trigonometric inference can achieve the same accuracy as conventional CNNs.

Table 4.7: Evaluation results of GoogleNet & ResNet50 on Cifar-100.

Model	Accuracy	Training Loss	Validation Loss
ResNet50 CNN	0.7245	0.0056	1.2234
ResNet50 TrigInf	0.7311	0.0158	1.3095
GoogleNet CNN	0.7418	0.0089	0.9284
GoogleNet TrigInf	0.7479	0.0170	0.9570

4.3.5 Future Works of Trigonometric Inference

To the best of our knowledge, this is the first study to utilize a trigonometric approximation of all parameters in the training of deep neural networks. However, on modern GPUs and CPUs, we were unable to show the efficiency of our training methods because they are specifically optimized for multiplication. We set as our future study the design of a hardware computation engine for trigonometric inference. Here, we briefly discuss the challenges. Although a simple serial CORDIC module requires significantly fewer logic resources than a multiplier, it has certain flaws in terms of speed. To accelerate the CORDIC module for neural network inference, the following two schemes are worth applying:

(1) CORDIC optimization for small values:

Although randomly distributed, layer inputs, weights, and back-propagated errors tend to have small variances in large deep neural networks. Starting the rotation from the mean value can reduce the number of iterations for the CORDIC modules because the mean value can be selected as a pre-computed angle.

(2) Neural network training with lower-precision CORDIC:

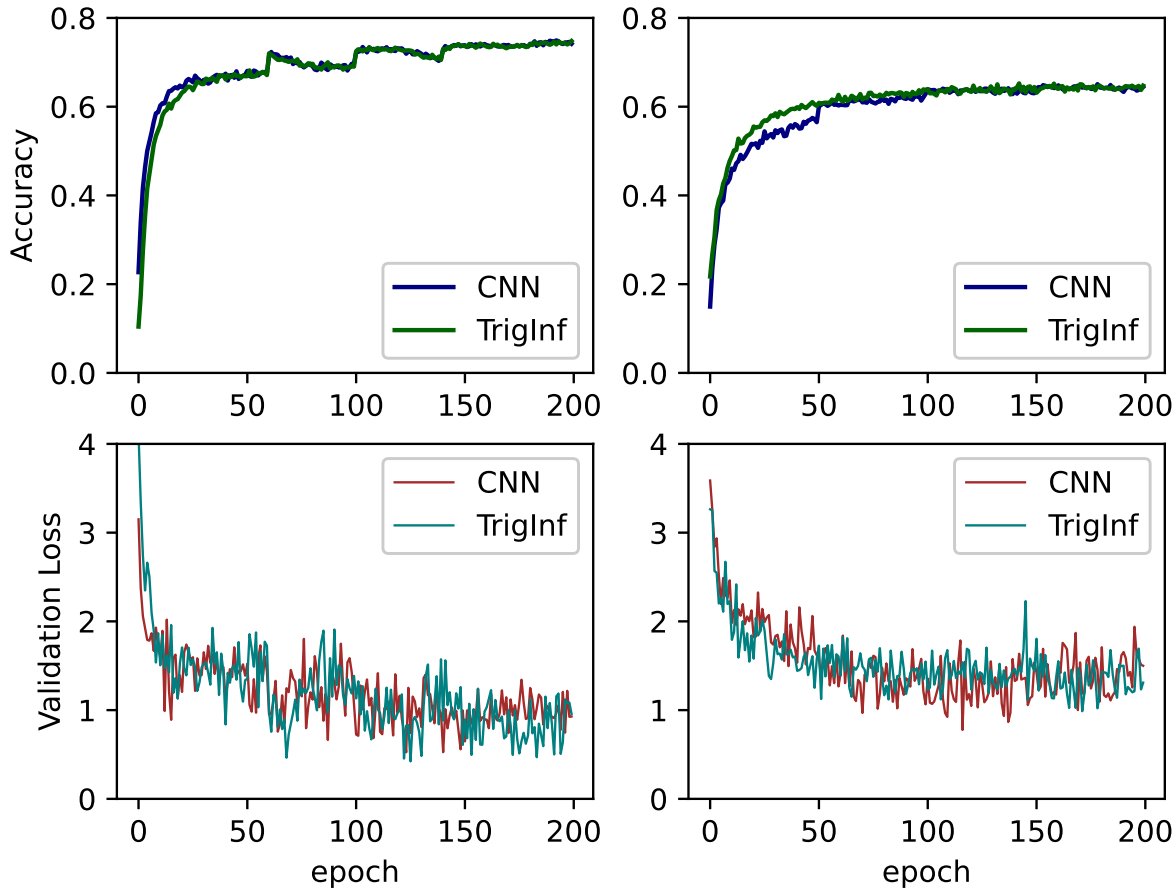


Figure 4.7: Training curves and Validation loss of GoogleNet and ResNet50 on CIFAR-100 dataset. The left subplots denote the validation accuracy and loss records on GoogleNet and the right subplots denote those on ResNet50.

As mentioned in Section 4.2, training with a lower precision can provide a comparable performance under many scenarios. Reducing the model precision to 16-bits, 8-bits, or even fewer bits will significantly decrease the CORDIC iteration times. Therefore, the training efficiency is higher when a lower precision is required.

Chapter 5

Conclusion

In this research, our main contribution is twofold:

First, we have analysed the disadvantages of existing quantization algorithms and proposed a new logarithmic algorithm that is simple yet powerful. We applied our algorithm to all layers of several popular neural networks showing its generality of handling layers of both large and small complexity. In convolutional neural networks such as GoogLeNet that use small filters, our algorithm performs substantially better than other algorithms without retraining. We also pointed out that quantization error is not suitable for evaluating the performance of quantization algorithms since it treats strong and weak connections equally.

By quantifying the importance of each weight to its magnitude, we present a weight-aware quantization optimization objective. We combined this weight-aware methodology with existing state-of-the-art quantization algorithms and achieved superior performance. For the additive power-of-two quantization method that requires retraining, our algorithm can significantly reduce the training time and eliminate the need for weight normalization. For the post-training quantization algorithm, our method can push the performance to an even better degree.

Although previous studies have realized that the importance of each weight is different, to the best of our knowledge, this is the first study to propose a weight-aware quantization optimization objective. We hope that there will be more research to use WAE (weight aware error) to replace the original MSE (mean squared error), and we believe that this simple modification will yield very good results for the algorithm.

Second, a novel technique called trigonometric inference for neural network training was also introduced in this study. Trigonometric inference is promising in removing multipliers altogether in deep neural network training and inference. We analyzed the precision of the sine approximation for deep neural network parameters. In addition, we exploited a rectified sine activation function to remove multiplications in inference and training. Trigonometric inference is suitable for scenarios in which the hardware multiplier is insufficient and is preferred when a hyperbolic activation function is utilized. Our experimental results demonstrate that a comparable performance was achieved using this approach.

The results of this research shed new light on future hardware customization for deep neural networks that aim to take advantage of the CORDIC computation engine and reduce the logic resources and power consumption. It is also practical for applications on weak terminals, such as microcontrollers equipped with CORDIC modules.

Acknowledgment

I would like to express my profound gratitude to my supervise Prof. Nakajo for his support and advice. I offer my sincere appreciation for the reviewers of this thesis: Prof. Hotta, Prof. Kaneko, Prof. Kondo and Prof. Yamai. I have benefited a lot from your valuable comments.

I would also like to acknowledge the lab secretaries Kaneko-san and Okita-san. I am highly indebted to all members in Nakajo lab especially Takemoto-san and Teruya-san. I would not be able to make it without your help. And finally, I would like to express my special gratitude to Ms. Sarocha for her endless support during my doctoral studies.

Bibliography

[1] Tsuguo Mogami. Deep neural network training without multiplications. *arXiv preprint arXiv:2012.03458*, 2020.

[2] Alex Krizhevsky. Learning multiple layers of features from tiny images. pages 32–33, 2009.

[3] Inc. GitHub. Open source survey. `https://github.com/huawei-noah/AdderNet`, 2021.

[4] Jinghui Li and Peiyu Fang. Hdrnet: Single-image-based hdr reconstruction using channel attention cnn. In *Proceedings of the 2019 4th International Conference on Multimedia Systems and Signal Processing*, pages 119–124, 2019.

[5] Fengxiang He, Tongliang Liu, and Dacheng Tao. Control batch size and learning rate to generalize well: Theoretical and empirical evidence. 2019.

[6] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

[7] Igor Gitman, Hunter Lang, Pengchuan Zhang, and Lin Xiao. Understanding the role of momentum in stochastic gradient methods. *arXiv preprint arXiv:1910.13962*, 2019.

[8] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[9] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.

[10] Alexandre Défossez, Léon Bottou, Francis Bach, and Nicolas Usunier. On the convergence of adam and adagrad. *arXiv preprint arXiv:2003.02395*, 2020.

[11] Jianzhi Yan, Bruno Andriamanalimanana, Chen-Fu Chiang, and Jorge Novillo. Non-convex optimization: Rmsprop based optimization for long short-term memory network. 2020.

[12] Pavlo Molchanov, Arun Mallya, Stephen Tyree, Iuri Frosio, and Jan Kautz. Importance estimation for neural network pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11264–11272, 2019.

[13] Gil Shomron, Ron Banner, Moran Shkolnik, and Uri Weiser. Thanks for nothing: Predicting zero-valued activations with lightweight convolutional neural networks. In *European Conference on Computer Vision*, pages 234–250. Springer, 2020.

[14] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[15] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 806–813, 2014.

[16] Ishani Patel, Virag Jagtap, and Ompriya Kale. A survey on feature extraction methods for handwritten digits recognition. *International Journal of Computer Applications*, 107(12), 2014.

[17] Mamta Garg and Deepika Ahuja. A novel approach to recognize the off-line handwritten numerals using mlp and svm classifiers. *International Journal Of Computer Science & Engineering Technology (IJCSET) ISSN*, pages 2229–3345, 2013.

[18] Johnson Loh, Jianan Wen, and Tobias Gemmeke. Low-cost dnn hardware accelerator for wearable, high-quality cardiac arrythmia detection. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 213–216. IEEE, 2020.

[19] Rahul Duggal, Scott Freitas, Cao Xiao, Duen Horng Chau, and Jimeng Sun. Rest: Robust and efficient neural networks for sleep monitoring in the wild. In *Proceedings of The Web Conference 2020*, pages 1704–1714, 2020.

[20] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[21] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *Pattern Recognition*, 105:107281, 2020.

[22] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[23] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[24] Wonyong Sung, Sungho Shin, and Kyuyeon Hwang. Resiliency of deep neural networks under quantization. *CoRR*, abs/1511.06488, 2015.

[25] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[26] Frederick Tung and Greg Mori. Clip-q: Deep network compression learning by in-parallel pruning-quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7873–7882, 2018.

[27] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, 2018.

[28] Junyan Dai. Real-time and accurate object detection on edge device with tensorflow lite. In *Journal of Physics: Conference Series*, volume 1651, page 012114. IOP Publishing, 2020.

[29] Brett Koonce. Squeezenet. In *Convolutional Neural Networks with Swift for Tensorflow*, pages 73–85. Springer, 2021.

[30] Seyyed Hossein Hasanpour, Mohammad Rouhani, Mohsen Fayyaz, and Mohammad Sabokrou. Lets keep it simple, using simple architectures to outperform deeper and more complex architectures. *arXiv preprint arXiv:1608.06037*, 2016.

[31] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *Proceedings of the IEEE international conference on computer vision*, pages 5058–5066, 2017.

[32] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. Incremental network quantization: Towards lossless cnns with low-precision weights. *CoRR*, abs/1702.03044, 2017.

[33] Aojun Zhou, Anbang Yao, Kuan Wang, and Yurong Chen. Explicit loss-error-aware quantization for low-bit deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 9426–9435, 2018.

[34] Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. Training quantized nets: A deeper understanding. *arXiv preprint arXiv:1706.02379*, 2017.

[35] Daisuke Miyashita, Edward H. Lee, and Boris Murmann. Convolutional neural networks using logarithmic data representation. *CoRR*, abs/1603.01025, 2016.

[36] Mostafa Elhoushi, Zihao Chen, Farhan Shafiq, Ye Henry Tian, and Joey Yiwei Li. Deepshift: Towards multiplication-less neural networks. *arXiv preprint arXiv:1905.13298*, 2019.

[37] Yuhang Li, Xin Dong, and Wei Wang. Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks. *arXiv preprint arXiv:1909.13144*, 2019.

[38] Shumeet Baluja, David Marwood, Michele Covell, and Nick Johnston. No multiplication? no floating point? no problem! training networks for efficient inference. *arXiv preprint arXiv:1809.09244*, 2018.

[39] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[40] Jun Fang, Ali Shafiee, Hamzah Abdel-Aziz, David Thorsley, Georgios Georgiadis, and Joseph H Hassoun. Post-training piecewise linear quantization for deep neural networks. In *European Conference on Computer Vision*, pages 69–86. Springer, 2020.

[41] Yoni Choukroun, Eli Kravchik, Fan Yang, and Pavel Kisilev. Low-bit quantization of neural networks for efficient inference. In *ICCV Workshops*, pages 3009–3018, 2019.

[42] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International conference on machine learning*, pages 2849–2858. PMLR, 2016.

[43] Ritchie Zhao, Yuwei Hu, Jordan Dotzel, Chris De Sa, and Zhiru Zhang. Improving neural network quantization without retraining using outlier channel splitting. In *International conference on machine learning*, pages 7543–7552. PMLR, 2019.

[44] Han Vanholder. Efficient inference with tensorrt, 2016.

[45] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. *Advances in neural information processing systems*, 29, 2016.

[46] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. In *Proceedings of the European conference on computer vision (ECCV)*, pages 722–737, 2018.

[47] Sajad Darabi, Mouloud Belbahri, Matthieu Courbariaux, and Vahid Partovi Nia. Bnn+: Improved binary network training. *arXiv preprint arXiv:1812.11800*, 2018.

[48] Jack Volder. The cordic computing technique. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 257–261, 1959.

[49] Francisco J Jaime, Miguel A Sánchez, Javier Hormigo, Julio Villalba, and Emilio L Zapata. Enhanced scaling-free cordic. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 57(7):1654–1662, 2010.

[50] ASN Mokhtar, MBI Reaz, K Chellappan, and MA Mohd Ali. Scaling free cordic algorithm implementation of sine and cosine function. In *Proceedings of the World Congress on Engineering (WCE13)*, volume 2, 2013.

[51] Kui-Ting Chen, Ke Fan, Xiaojun Han, and Takaaki Baba. A cordic algorithm with improved rotation strategy for embedded applications. *Journal of Industrial and Intelligent Information*, 3(4), 2015.

[52] Yuan-Ho Chen, Szi-Wen Chen, and Min-Xian Wei. A vlsi implementation of independent component analysis for biomedical signal separation using cordic engine. *IEEE Transactions on Biomedical Circuits and Systems*, 14(2):373–381, 2020.

[53] Vipin Tiwari and Ashish Mishra. Neural network-based hardware classifier using cordic algorithm. *Modern Physics Letters B*, page 2050161, 2020.

[54] Moslem Heidarpur, Arash Ahmadi, Majid Ahmadi, and Mostafa Rahimi Azghadi. Cordic-snn: On-fpga stdp learning with izhikevich neurons. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66(7):2651–2661, 2019.

[55] Xinyu Hao, Shuangming Yang, Jiang Wang, Bin Deng, Xile Wei, and Guosheng Yi. Efficient implementation of cerebellar purkinje cell with cordic algorithm on lacsnn. *Frontiers in neuroscience*, 13:1078, 2019.

[56] Hanting Chen, Yunhe Wang, Chunjing Xu, Boxin Shi, Chao Xu, Qi Tian, and Chang Xu. Addernet: Do we really need multiplications in deep learning? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1468–1477, 2020.

[57] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[58] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[59] Min Soo Kim, Alberto A Del Barrio, Leonardo Tavares Oliveira, Roman Hermida, and Nader Bagherzadeh. Efficient mitchells approximate log multipliers for convolutional neural networks. *IEEE Transactions on Computers*, 68(5):660–675, 2018.

[60] John N Mitchell. Computer multiplication and division using binary logarithms. *IRE Transactions on Electronic Computers*, (4):512–517, 1962.

[61] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015.

[62] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2002–2011. ACM, 2019.

[63] Haoran You, Xiaohan Chen, Yongan Zhang, Chaojian Li, Sicheng Li, Zihao Liu, Zhangyang Wang, and Yingyan Lin. Shiftaddnet: A hardware-inspired deep network. *arXiv preprint arXiv:2010.12785*, 2020.

[64] Yunhe Wang, Mingqiang Huang, Kai Han, Hanting Chen, Wei Zhang, Chunjing Xu, and Dacheng Tao. Addernet and its minimalist hardware design for energy-efficient artificial intelligence. *arXiv preprint arXiv:2101.10015*, 2021.

[65] Yixing Xu, Chang Xu, Xinghao Chen, Wei Zhang, Chunjing Xu, and Yunhe Wang. Kernel based progressive distillation for adder neural networks. *arXiv preprint arXiv:2009.13044*, 2020.

[66] Jianping Gou, Baosheng Yu, Stephen John Maybank, and Dacheng Tao. Knowledge distillation: A survey. *arXiv preprint arXiv:2006.05525*, 2020.

[67] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *European conference on computer vision*, pages 184–199. Springer, 2014.

[68] Jiwon Kim, Jung Kwon Lee, and Kyoung Mu Lee. Accurate image super-resolution using very deep convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1646–1654, 2016.

[69] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 136–144, 2017.

[70] Dehua Song, Yunhe Wang, Hanting Chen, Chang Xu, Chunjing Xu, and DaCheng Tao. Addersr: Towards energy efficient image super-resolution. *arXiv preprint arXiv:2009.08891*, 2020.

[71] Marco Bevilacqua, Aline Roumy, Christine Guillemot, and Marie Line Alberi-Morel. Low-complexity single-image super-resolution based on nonnegative neighbor embedding. 2012.

[72] Roman Zeyde, Michael Elad, and Matan Protter. On single image scale-up using sparse-representations. In *International conference on curves and surfaces*, pages 711–730. Springer, 2010.

[73] David Martin, Charless Fowlkes, Doron Tal, and Jitendra Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 2, pages 416–423. IEEE, 2001.

[74] Jia-Bin Huang, Abhishek Singh, and Narendra Ahuja. Single image super-resolution from transformed self-exemplars. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5197–5206, 2015.

[75] Timothy P Lillicrap, Daniel Cownden, Douglas B Tweed, and Colin J Akerman. Random synaptic feedback weights support error backpropagation for deep learning. *Nature communications*, 7(1):1–10, 2016.

[76] Arild Nøkland. Direct feedback alignment provides learning in deep neural networks. In *Advances in neural information processing systems*, pages 1037–1045, 2016.

[77] Maria Refinetti, Stéphane d'Ascoli, Ruben Ohana, and Sebastian Goldt. The dynamics of learning with feedback alignment. *arXiv preprint arXiv:2011.12428*, 2020.

[78] Arild Nøkland and Lars Hiller Eidnes. Training neural networks with local error signals. In *International Conference on Machine Learning*, pages 4839–4850. PMLR, 2019.

[79] Hesham Mostafa, Vishwajith Ramesh, and Gert Cauwenberghs. Deep supervised learning using local errors. *Frontiers in neuroscience*, 12:608, 2018.

[80] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[81] Julian Faraone, Nicholas Fraser, Michaela Blott, and Philip HW Leong. Syq: Learning symmetric quantization for efficient deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4300–4309, 2018.

[82] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[83] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. 29:141–142, 11 2012.

[84] K Simonyan and A Zisserman. Very deep convolutional networks for large-scale image recognition. arxiv 1409.1556 (09 2014). *URL https://arxiv. org/abs/1409.1556. Accessed: February*, 2020.

[85] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[86] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed <today>].

[87] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[88] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[89] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[90] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.

[91] Eunhyeok Park, Sungjoo Yoo, and Peter Vajda. Value-aware quantization for training and inference of neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 580–595, 2018.

[92] Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. Towards the limit of network quantization. *arXiv preprint arXiv:1612.01543*, 2016.

[93] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M Bronstein, and Avi Mendelson. Loss aware post-training quantization. *arXiv preprint arXiv:1911.07190*, 2019.

[94] Lu Hou and James T Kwok. Loss-aware weight quantization of deep networks. *arXiv preprint arXiv:1802.08635*, 2018.

[95] Lu Hou, Quanming Yao, and James T Kwok. Loss-aware binarization of deep networks. *arXiv preprint arXiv:1611.01600*, 2016.

[96] Zhongnan Qu, Zimu Zhou, Yun Cheng, and Lothar Thiele. Adaptive loss-aware quantization for multi-bit networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7988–7997, 2020.

[97] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Hawq: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 293–302, 2019.

[98] Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382, 2018.

[99] Mingbao Lin, Rongrong Ji, Zihan Xu, Baochang Zhang, Yan Wang, Yongjian Wu, Feiyue Huang, and Chia-Wen Lin. Rotated binary neural network. *arXiv preprint arXiv:2009.13055*, 2020.

[100] Wen-Pu Cai and Wu-Jun Li. Weight normalization based quantization for deep neural network compression. *arXiv preprint arXiv:1907.00593*, 2019.

[101] Hadi Pouransari, Zhucheng Tu, and Oncel Tuzel. Least squares binary quantization of neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 698–699, 2020.

[102] Haotong Qin, Ruihao Gong, Xianglong Liu, Mingzhu Shen, Ziran Wei, Fengwei Yu, and Jingkuan Song. Forward and backward information retention for accurate binary neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2250–2259, 2020.

[103] Jingyong Cai, Masashi Takemoto, and Hironori Nakajo. A deep look into logarithmic quantization of model parameters in neural networks. In *Proceedings of the 10th International Conference on Advances in Information Technology*, pages 1–8, 2018.

[104] Tim Salimans and Diederik P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. *arXiv preprint arXiv:1602.07868*, 2016.

[105] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[106] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[107] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[108] Vipin Tiwari and Nilay Khare. Hardware implementation of neural network with sigmoidal activation functions using cordic. *Microprocessors and Microsystems*, 39(6):373–381, 2015.

[109] Ismail Koyuncu. Implementation of high speed tangent sigmoid transfer function approximations for artificial neural network applications on fpga. *Advances in Electrical and Computer Engineering*, 18(3):1–8, 2018.

[110] Gil Shomron and Uri Weiser. Spatial correlation and value prediction in convolutional neural networks. *IEEE Computer Architecture Letters*, 18(1):10–13, 2018.

[111] Gil Shomron and Uri Weiser. Non-blocking simultaneous multithreading: Embracing the resiliency of deep neural networks. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 256–269. IEEE, 2020.

[112] Arnab Sanyal, Peter A Beerel, and Keith M Chugg. Neural network training with approximate logarithmic computations. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3122–3126. IEEE, 2020.

[113] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[114] Jiajun Wu, Yi Zhan, Zixuan Peng, Xinglong Ji, Guoyi Yu, Rong Zhao, and Chao Wang. Efficient design of spiking neural network with stdp learning based on fast cordic. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 68(6):2522–2534, 2021.

[115] Josep M Sopena, Enrique Romero, and Rene Alquezar. Neural networks with periodic and monotonic activation functions: a comparative study in classification problems. In *In Proceedings of the 9th International Conference on Artificial Neural Networks*, pages 323–328. IET, 1999.

[116] Giambattista Parascandolo, Heikki Huttunen, and Tuomas Virtanen. Taming the waves: sine as activation function in deep neural networks. 2016.

[117] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. *arXiv preprint arXiv:1710.05941*, 2017.

[118] Yunong Zhang, Lu Qu, Jinrong Liu, Dongsheng Guo, and Mingming Li. Sine neural network (snn) with double-stage weights and structure determination (ds-wasd). *Soft Computing*, 20(1):211–221, 2016.

[119] Brian Alexander Crafton, Abhinav Parihar, Evan Gebhardt, and Arijit Raychowdhury. Direct feedback alignment with sparse connections for local learning. *Frontiers in neuroscience*, 13:525, 2019.

[120] Moslem Heidarpour, Arash Ahmadi, and Rashid Rashidzadeh. A cordic based digital hardware for adaptive exponential integrate and fire neuron. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 63(11):1986–1996, 2016.

[121] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[122] Muhammad Awais, Md Tauhid Bin Iqbal, and Sung-Ho Bae. Revisiting internal covariate shift for batch normalization. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.