Improving Compression Pipelines For Convolutional Neural Networks

Yiren (Aaron) Zhao
Corpus Christi College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: yaz21@cam.ac.uk

June 11, 2017

Declaration

I Yiren (Aaron) Zhao of Corpus Christi College, being a candidate for the M.Phil in Advanced Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 14,235

Signed:

Date:

Abstract

This is the abstract. Write a summary of the whole thing. Make sure it fits in one page.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Neural Networks have achieved outstanding accuracies in large-scale image classifications and they are becoming a state-of-art technique for solving problems in computer vision [2, 6, 7].

It is the rise of deep neural network, in particular, that has helped researchers to achieve outstanding accuracies on many popular datasets. In 1998, *Lecun et al.* proposed a 5-layer network called *LeNet5* for recognising hand-written digits [6]. *Lecun et al.* utilised around 1M parameters to achieve a good accuracy on this particular recognition task. *AlexNet*, designed by *Krizhevsky et al.* for the 2012 ImageNet competition, trained on 1.2 million images and was used for classification with 1000 categories [2]. The 1000 classes include every-day objects (cat, dog, leopard and etc), and the proposed *AlexNet* utilized around 60 million parameters [2]. *Coates et al.* scaled up the learning algorithm to utilise over 11 billion parameters and ran it on 16 machines to recognise unlabelled human faces [8]. Neural networks are becoming deeper and are utilizing a larger number of parameters for harder recognition problems.

Table 1.1 summarized some popular networks, the results show the trend that neural networks are going deeper. Before 2014, researchers focus on designing large neural networks to improve error rates on the *ImageNet* dataset

[1]. *AlexNet* and *VGG16* both use stacked fully connected layers and convolutional layers [2, 9], and consumed a larger number of parameters. After 2014, researchers starts to develop more efficient network architectures and training methodologies. *InceptionV3* uses *Batch normalization* [10] and factorizes traditional large convolutional kernels into small convolutional kernels [11]. *ResNet200* makes use of shortcut connections, these connections skip one or multiple layers [12]. Figure 1.1 gives an overview of the networks mentioned above by considering their top-5 error rates and sizes when used on the *ILSVRC 2012* dataset [1]. *ResNet200* strikes the best balance between accuracy and sizes. It is possible for neural networks to have an elegant architecture and thus achieves a good accuracy at a reasonable size. However, *ResNet200* has a size of around 100MB which is still hard to execute on power sensitive devices. In addition, in the future, neural networks will be applied on harder datasets than *ImageNet*. It is reasonable to assume how to execute neural networks on power-sensitive and memory-limited devices can be a serious challenge in the near future.

Model	*AlexNet*	*VGG16*	*InceptionV3*	*ResNet200*
Layers	8	16	22	200
Year	2012	2014	2015	2015

Table 1.1: Number of layers and proposed years of various networks.

While the parallelisms offered by GPUs can be exploited to accelerate neural network training in a large scale , deploying a network with a large number of parameters on a memory-limited power-sensitive device becomes increasingly difficult. To constraint the problem space, this project focuses on neural network inference rather than training. There are some recent proposed algorithms, such as *Deep Q Learning* [13] and *Asynchroous Q Learning* [14], that require training of a neural network on a local embedded system. This need of executing network training in local embedded systems is still only essential for a small set of learning algorithms. The more appealing problem currently is how to execute network inference in an efficient manner. To execute network inference, although it is a simpler task than training, mobile systems struggle to achieve a reasonable power efficiency. Two major constraints pre-

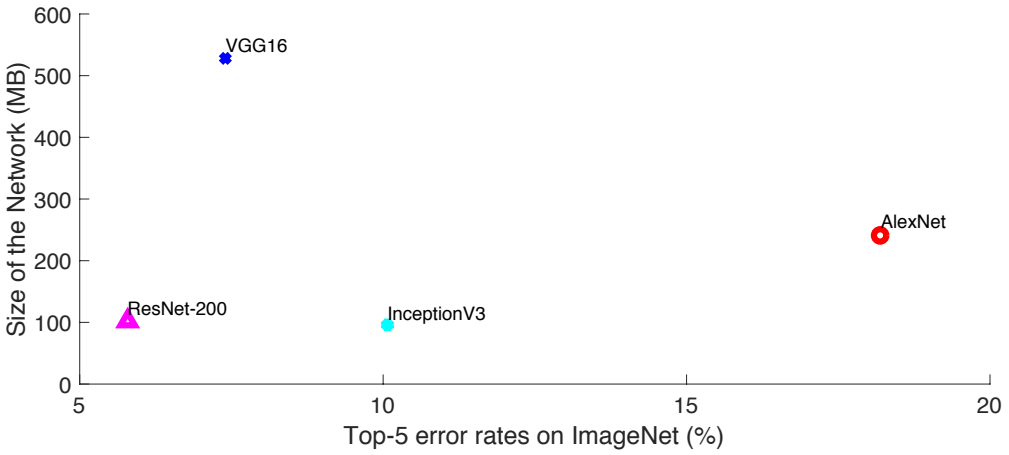

Figure 1.1: Popular networks targeting on *ImageNet ILSVRC 2012* dataset [1].

vent neural network inference from being applicable in real-life. First, the large size of a neural network makes storing its parameters in embedded systems very challenging. For example, the *AlexNet* model is over 200MB and *VGG16* model is over 500MB [4, 9]. These large storage overheads post a challenge to the fundamental limit of memory size and memory bandwidth of embedded systems. Second, energy consumption is dominated by memory accesses [15], and accessing a large number parameters of a neural network can exceed the energy envelope of many power sensitive devices. For instance, the battery of a smartphone struggles with running object classification using *AlexNet* in real-time for more than an hour [15].

To resolve the above problems, the computer architecture community is now actively researching on novel hardware architectures for neural network inference. Many novel custom hardware architectures have been proposed for neural network inference [16, 17, 18, 19]. Various FPGA-based accelerators have been recently applied on neural network inference [20, 21]. In addition, there is an increasing number of ASIC designs for deep neural network inference [18, 22]. These accelerators normally utilize a large on-chip memory and have custom computing units to calculate matrix dot-products. A custom accelerator is definitely beneficial for running neural network inference

efficiently, but accessing and storing the large number of parameters in the memory is still a limit for these accelerators.

To build a hardware accelerator, the first step is to consider how to condense the neural network to a reasonable size. A compressed neural network is more amenable to memory-limited systems and also reduces the energy cost of data movements. Neural network compression is recently catching attentions and many research works have made significant contributions in this field. I would like to systematically summarize and evaluate state-of-art neural network compression techniques. These compression techniques are categorized into three groups: pruning, regularization and quantization. Some of these methods, such as pruning and regularization, encourage sparsity in a neural network; in other words, these methods reduce the number of parameters in a neural network. Other methods, such as quantization, takes a different approach for condensing a neural network: they reduce the number of bits required to represent a single parameter in a neural network. In this report, the aim is to combine several state-of-art compression techniques to build a complete compression pipeline. To further constraint the problem, one of the targets is to condense networks with no test accuracy loss. The proposed compression pipeline has achieved a compression rate of 403x on *LeNet5* and a compression rate of 82.2x on *CifarNet*. These compression results outperformed many existing compression techniques by considerable margins [4, 3].

Chapter 2

Background

2.1 Convolutional Neural Networks

Neural networks have been biologically inspired by the actual neural systems in human brains. Neurons are basic units in a neural network and each neuron provides an output based on its connections (weights) to neurons of the previous layer and an additional offset value (bias). A number of neurons consists a layer, and a neural network is a layer-wise architecture. Two types of layers are the major components in a convolutional neural networks: fully connected layers and convolutional layers. A fully connected layer, as its name states, has neurons that are fully connected to all neurons in the previous layer and neurons in a single layer do not share any connections. However, using only fully connected layers lose particular feature information for image recognition. Convolutional layers requires neurons to be connected to a local region of the previous layer for extracting specific features of a given image. Convolutional layers normally have 3D volumes of neurons [2]. Convolutions leverage three important ideas: *parameter sharing*, *sparse interactions* and *equivariant representations* [23]. The use of small kernels reduces connections since kernels are much smaller than inputs (*sparse interactions*), and each member of the kernel is used at every pixel of the input (*parameter sharing*). *Equivariant representations* help convolutional neural

networks to extract features at each layer and sometimes even abstract features when the layer count is large. Figure 2.1 shows a typical neural network structure for image recognition. The first few layers are convolutional layers followed by maxpooling layers to extract features from images. Maxpooling layers downsample the inputs and thus reduce the inputs' dimensionalities. As mentioned before, these convolutional layers are all 3D volumes of neurons. The last convolutional layer then flattens to connect to fully connected layers. These fully connected layers, although are 2D, but normally have a large number of neurons, for instance, the shown figure of AlexNet has 2048 neurons on its first fully connected layer [2]. A typical neural network architecture is illustrated in Figure 2.1, in this case, this neural network has five convolutional layers and three fully-connected layers [2].

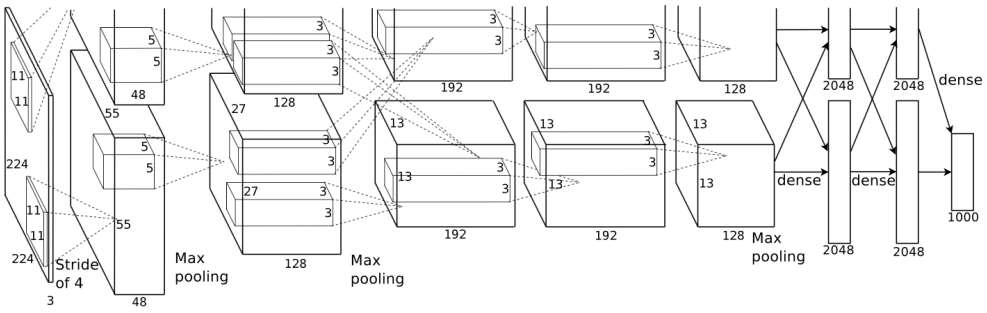

Figure 2.1: AlexNet Network Architecture [2].

From a mathematical perspective, consider w_i to be a vector of weights, b_i to be a single biase and x_i to be the input vector of a particular neuron i, this neuron generates the following output $y_i = w_i x_i + b_i$. This mathematical model of a single neuron feeding forward is the same as the model of a linear classifier [24]. As illustrated in Figure 2.2, the basic setup of neurons can be seen as a mathematical model of actual neurons in the human brain system. In a neural network, each output y_i of a single neuron has to go through an activation function, this added non-linearity becomes a key for neural network to achieve good performance. Popular activation functions include *sigmoid* ($f(x) = \frac{1}{1+e^{-x}}$), *tanh* ($f(x) = tanh(x)$) and *relu* ($f(x) = max(0, x)$).

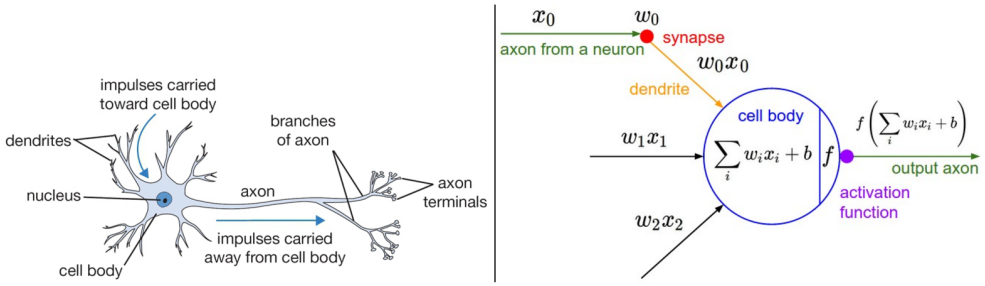

Figure 2.2: Fully connected layer followed by an activation function [2].

Inference and training are two important operations that a convolutional neural network can perform. Network inference refers to a process of feeding image data to a network model and the network produces a classification result based on its assigned parameters. In a typical convolutional neural network, such a process only requires values from the previous layer to be fed into the following layer, so it is also known as forward propagation. Network training, on the other hand, performs the same forward propagation process but has an extra process called backpropagation [25]. In the training stage, the predicted classification result generated from forward propagation is compared to a target for producing an error term (loss function). Backpropagation works by propagating this error term from the last output layer to the first input layer. As shown in Equation (2.1), $\mathbf{W_n}$ are weights and $\mathbf{B_n}$ are biases for a given layer n, α is a hyperparameter called the leanring rate. A gradient value is calculated by differentiating the loss value $L(W_n + B_n)$ with respect to each individual parameters, this first directive determines how parameters should adjust their values for achieving a better classification result. The backpropogation process then updates each parameter based on their original values and gradients, notice the learning rate α determines how aggressive this update is.

$$\mathbf{W_n} \leftarrow \mathbf{W_n} - \alpha \frac{\partial}{\partial(\mathbf{W_n} + \mathbf{B_n})} L(\mathbf{W_n} + \mathbf{B_n}) \tag{2.1}$$

The current research trend suggests convolutional neural networks will use

more layers for achieving a higher accuracy on more complicated problems [6, 2, 8].

2.2 Related Work

A variety of methods have been proposed in the research community for reducing the memory requirement of a neural network. Although redesign the original network to a smaller network is directly beneficial, designing an efficient network topology normally requires a large design-time. Because efficient topologies require experimenting a set of complex connections such as skipping layers (*ResNet200*) and combing multiple parallel layers (*InceptionV3*) [12, 11]. Another method of reducing the size of a neural network is network compression. In general, most of compression techniques focus on the following two paths.

1. Reduce the number of parameters in a neural network.
2. Reduce the bit-width of number representations.

The first methodology focuses on helping a neural network to use fewer parameters. The second path is to reduce the bit-width of each individual parameter in a neural network, this could be done by simply quantizing the parameters or by exploring alternative encodings or by adding a level of indirection. Figure 2.3 provides a taxonomy of compression techniques that follow these two paths.

Network pruning has been widely used for reducing the number of parameters in convolutional neural networks [6, 26, 27]. Pruning methods can be classified based on their granularities (Figure 2.3) [28]. Fine-grained pruning refers to pruning methods that delete individual weights. Starting from *LeCun et al.*, their idea is to use the second derivative of the network's loss function to balance the predication accuracy and model complexity [6]. *Hassibi et al.* proposed to add the non-diagonal elements of the Hessian matrix into the pruning metric, this is proven to be helpful in reducing excess degrees

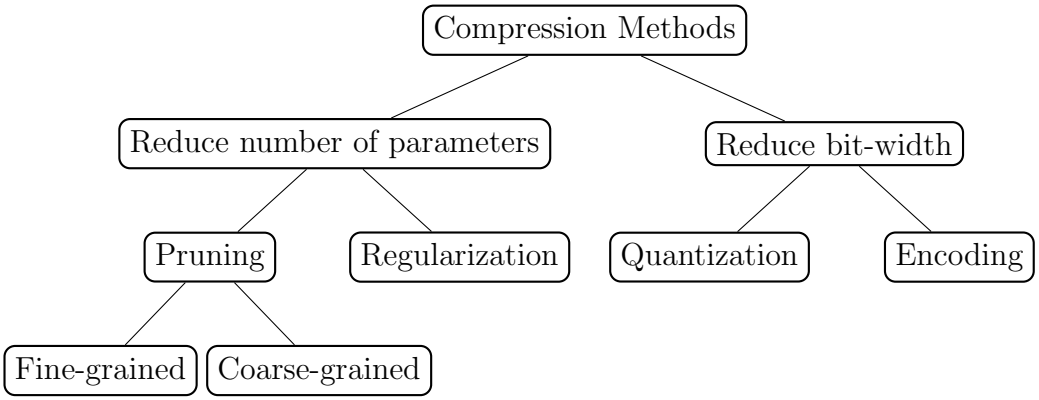

Figure 2.3: Taxonomy of compression techniques.

of freedom in the neural network model [26]. Recently, *Han et al.* utilised fine-grained pruning and achieved high compression rates on many popular convolutional neural networks [4]. Their pruning technique is tightly coupled with training, after every pruning process, a retraining takes place to bring back the lost accuracies. This class of pruning is also called *Iterative pruning*. Inspried by *Han et al.'s* work, later on, *Guo et al.* implemented *Dynamic network surgery* that combines fine-grained pruning with splicing. In the splicing phase of *Dynamic network surgery*, some weights that was previously pruned away would have a chance to recover. *Dynamic network surgery* is able to prune more aggressively and achieved better results than traditional iterative pruning [3]. Coarse-grained pruning, on the other hand, focus on pruning filters of convolutional neural networks [29]. Such coarse-grained pruning eases the design of hardware accelerators due to regularities [30, 31]. Given the requirement of no loss of test accuracy is allowed, fine-grained pruning normally works better than coarse-grained pruning, because the later suffers from a structural loss by pruning away complete filters [29].

Apart from pruning, there are other existing techniques that focus on reducing the number of parameters of a neural network model. Regularization focuses on encouraging sparsity of the neural network while training the network. It has long been known that adding l_1 norm and l_2 norm encourages sparsities in a neural network. However, these regularizers only restrict the magnitudes of weights and thus induces sparsity in a less controlled man-

ner. *Srinivas et al.* proposed to use *spike-and-slab* regularizers for achieving pruning on-the-fly with the training process [27]. *Kang et al.* proposed a new regularizer called *Shakeout*. In *Shakeout*, the activation functions are customized at each neuron. The practical results of using *Shakeout* suggest that this regularizer induces more sparsity than traditional regularization methods.

Reducing the number of bits required to represent each individual parameter also directly compresses a given neural network. *Han et al.* used *Weights sharing* to group parameters with similar values [4]. *Weights sharing* works like an encoding scheme and it normally considers weights in a layer-wise fashion. For weights in each layer, a clustering algorithm is applied on these weights to group them with various centroid values [4]. These centroids are encoded using a hash function. For later inference operation, the centroid values are stored on-chip and each weight is represented using a hash key that is normally small in terms of bit-width. This idea origins from *HashNet* proposed by *Chen et al.* [32], but is different from the original work since now the hash function applies only on a trained network rather than changing the entire network architecture. Quantization is another popular method for reducing the bit-width of individual parameters. Fixed-point arithmetic has been confirmed to be a more energy efficient arithmetic for neural network inference [33]. Quantized networks, also known as low precision networks, utilizes low-precision fixed-point arithmetic to reduce the bit-width of individual parameters in the neural network [34]. Reaching an extreme, neural networks with parameters of only 1-bit width are known as *Binarized neural networks* [35]. *Ternary weight networks*, neural networks with parameters constrained to 1, 0 and -1, have been recently proposed and demonstrated an advance in performance compared to *Binarized neural networks* [36]. However, *Binarized neural networks* and *Ternary weight networks* normally require re-implementations to recover its accuracy loss and thus is beyond the scope of this project. For reducing the size of individual parameters, in this project, I will focus on various quantization methods and *Weights sharing*.

Although a large number of compressing techniques have been proposed, most of these techniques are proposed in isolations – they do not consider how to combine with each other in an optimal manner. Recently, *Han et al.* firstly proposed *Deep Compression* that is a complete compression pipeline utilizing several different compression techniques. It is possible to build a compression chain that makes use of compression techniques from orthogonal optimization spaces. *Deep Compression* offers a large compression rate on many popular networks combing *Deterministic pruning*, *Fixed-point quantization*, *Weights sharing* and *Huffman encoding*. This project aims to evaluate further in this compression optimization space. A larger number of state-of-art compression techniques, including different pruning schemes and quantization schemes are considered for building a better compression pipeline. In addition, this project puts a focus on developing novel pruning and quantization strategies based on some existing compressing techniques.

2.3 Experiment Setups, Datasets and Trained Networks

In this project, I select *TensorFlow* to be the implementation tool [37]. This package has python APIs that are easy to use, and *CUDA* backend for efficient GPU utilizations. Several local GPU machines and Amazon Cloud machines are used to train the networks.

Two datasets, *MNIST* [38] and *CIFAR10* [39], are considered in this project. Both datasets are smaller than the popular *ImageNet* dataset [1]. Sine the optimization space is large and training a large network on a large dataset is time-consuming, the *ImageNet* dataset is not considered in this project. *MNIST* dataset is a relatively small dataset for experimenting ideas. *CIFAR10* dataset serves as a representative for large networks.

The first dataset, *MNIST*, consists of handwritten digits. *MNIST* has a training set of $60,000$ images and a test set of $10,000$ images. These images

are a subset of the bigger *NIST* dataset [40], and all the digts in this dataset have been normalized in size and centered in the middle [38]. I used the *LeNet5* model [41] to recognize *MNIST* images.

Another dataset considered in this project is the *Cifar10* dataset. *Cifar10* is a subset of the *80 million tiny images dataset* [42]. The *Cifar10* dataset is divided into five training batches and one test batch, each batch contains $10,000$ images [39]. *CifarNet* is the neural network architecture used to recognize images in the *Cifar10* dataset [43]. It is important to note that, all these datasets are carefully designed so that all classes are mutually exclusive.

Layer	cov1	cov2	fc1	fc2	total
Params	0.5K	25K	400K	5K	431K

Table 2.1: Number of parameters in LeNet5-431K.

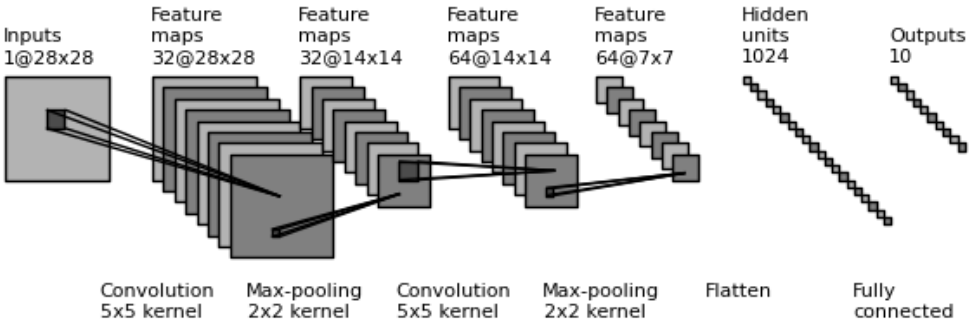

Figure 2.4: LeNet5 Network Architecture

Firstly, I am using a trained *LeNet5* model with 431K parameters that has an error rate of only 0.64% on the *MNIST* dataset. The original *LeNet5* model proposed by *LeCun et al.* has around 1000K parameters, however, later research works all use an implementation of *LeNet5* with 431K parameters [44, 4, 3]. In this project, the 431K implementation of *LeNet5* is chosen to enable a fair comparisons to be made with published results from related research works. The detailed information regarding the number of parameters at each layer of this *LeNet5* network is shown in Table 2.1. The network's

architecture is shown in Figure 2.4. As shown in the figure, this architecture includes two convolutional layers and three fully connected layers.

The *Cifar10* dataset is larger compared to *MNIST*. A *CifarNet* architecture is constructed and trained to an error rate of 18%. As expected, since this larger dataset implies a harder image recognition task, the trained model uses a larger number of parameters and more layers to achieve a reasonable accuracy. A graphical illustration of the *CifarNet* architecture is shown in Figure 2.5, The *CifarNet* architecture has two convolutional layers and three fully connected layers. The input images of the dataset now have three channels. The number of parameters at each layer of the *CifarNet* is summarized in Table 2.2.

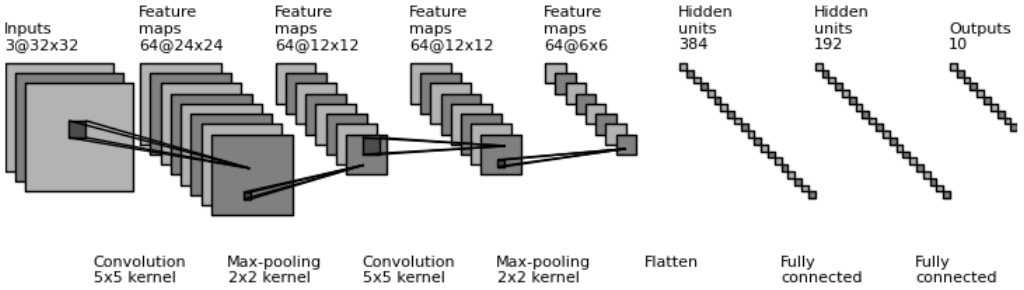

Figure 2.5: CifarNet Network Architecture

Layer	cov1	cov2	fc1	fc2	fc3	total
Params	4.8K	102.4K	885K	74K	2K	1068K

Table 2.2: Number of parameters in CifarNet.

Chapter 3

Pruning

Pruning is an effective method for reducing the redundancies in a network. I mainly focus on fine-grained pruning methods in this project since they achieve best compression results [28]. Fine-grained pruning inspects all parameters in a layer and moves away individual connections that have values lower than a certain threshold. Pruning is normally combined with iterative retraining so that lost accuracies caused by pruning can be recovered directly. In this chapter, I aim to reproduce existing fine-grained pruning techniques: *Deterministic pruning* [45] and *Dynamic Network Surgery* [3]. At the end of this chapter, I compared the performances of various existing fine-grained pruning strategies on selected neural network models.

3.1 Deterministic Pruning

3.1.1 Pruning Both Weights and Biases

Pruning reduces the total number of parameters and finds the minimal neural network topology, but an overly pruned network suffers from a significant accuracy drop and even divergence [46]. Consider a set of weights for a neural network of N layers, the weights can be expressed as $\{\mathbf{W_n} : 0 \leq n \leq N\}$, so

that $\mathbf{W_n}$ is the weights for a given layer n. Similarly, biases are expressed as $\{\mathbf{B_n} : 0 \leq n \leq N\}$. To represent a sparse model, a binary matrix $\{\mathbf{M_n} : 0 \leq n \leq N\}$ for weighs and a binary matrix $\{\mathbf{M_n^b} : 0 \leq n \leq N\}$ for biases are used to indicate connections that are kept. For simple *Deterministic pruning*, Equation (3.1) shows how this binary mask is computed using the weights. An arbitrary threshold value t_n is used to determine whether a certain weight variable should be pruned away. I use $h_n(*)$ to denote the discriminative function that produces a binary mask matrix based on the weight matrix.

$$h_n(\mathbf{W_n}^{(i,j)}) = \begin{cases} 0, & \text{if } t_n < |\mathbf{W_n}^{(i,j)}| \\ 1, & \text{if } t_n \geq |\mathbf{W_n}^{(i,j)}| \end{cases} \tag{3.1}$$

The appealing question now is how to determine the threshold value t_n. Instead of picking threshold values in an arbitrary fashion [45] like *Han et al.* , motivated by the implementation of *Dynamic network surgery* [3], the following metric is used for determining the pruning threshold t_n:

$$t_n = u_n + c\sigma_n \tag{3.2}$$

u_n is the mean value and σ_n is the standard deviation of the parameters in layer n. In Equation (3.2), c is a hyperparameter used to define how aggressive this pruning is. The threshold determination in Equation (3.2) works better than defining an arbitrary threshold value since threshold can be determined automatically without manually inspecting the weights distributions.

In a sparse neural network model, weights of each layer n can be easily expressed using an element-wise product between W_n and M_n. The loss function, expressed as $L_n(*)$, is a metric that measures the optimisation target of the neural network training phase. For a given layer of n, the

following equations set the optimization objective:

$$
\begin{aligned}
&min(L_n(\mathbf{W_n} \odot \mathbf{M_n} + \mathbf{B_n} \odot \mathbf{M_n^b})) \\
&\mathbf{M_n} = h_n(\mathbf{W_n}) \\
&\mathbf{M_n^b} = h_n(\mathbf{B_n})
\end{aligned}
\tag{3.3}
$$

For simplicity, the above optimization target is only for a given layer n, a complete model optimization target is different from this single layer target. Nonetheless, Equation (3.3) characterizes a sparse model. The $\odot$ symbol represents *Hadamard product*, which is a binary operation that takes two matrices and produces another matrix that has elements are the product of the elements of the original two matrices. In *Deterministic pruning*, pruned weights cannot recover their values. However, *Deterministic pruning* occurs iteratively – a number of weights are pruned away at each iteration and then retraining occurs to bring back the lost test accuracy caused by pruning. Although pruned weights cannot recover their values, other existing weights would have a chance to re-learn the correlations between existing connections following Equation (3.4) [3]. The weights are updated concurrently by the pruned model with a learning rate of α.

$$
\mathbf{W_n} \leftarrow \mathbf{W_n} - \alpha \frac{\partial}{\partial(\mathbf{W_n} \odot \mathbf{M_n} + \mathbf{B_n} \odot \mathbf{M_n^b})} L(\mathbf{W_n} \odot \mathbf{M_n} + \mathbf{B_n} \odot \mathbf{M_n^b})
\tag{3.4}
$$

Figure 3.1 shows how pruning affects the weight distribution. In these plots, pruned weights with a value of zero are not plotted. The plots focus on the first fully connected layer of a *LeNet5* model. The unpruned network has weights that are normally distributed and center at zero. The weights spread out to both positive and negative ends, and the values of weights are normally small. In contrast, the second plot shows how pruned weights look like: they gather at two centers – leaving a blank region around zero.

Table 3.1 shows the detailed pruning information at each layer of the *LeNet5*. Pruning reduces the number of parameters of *LeNet5* architecture without dropping the test accuracy. The compressed network is only 3.79% of the

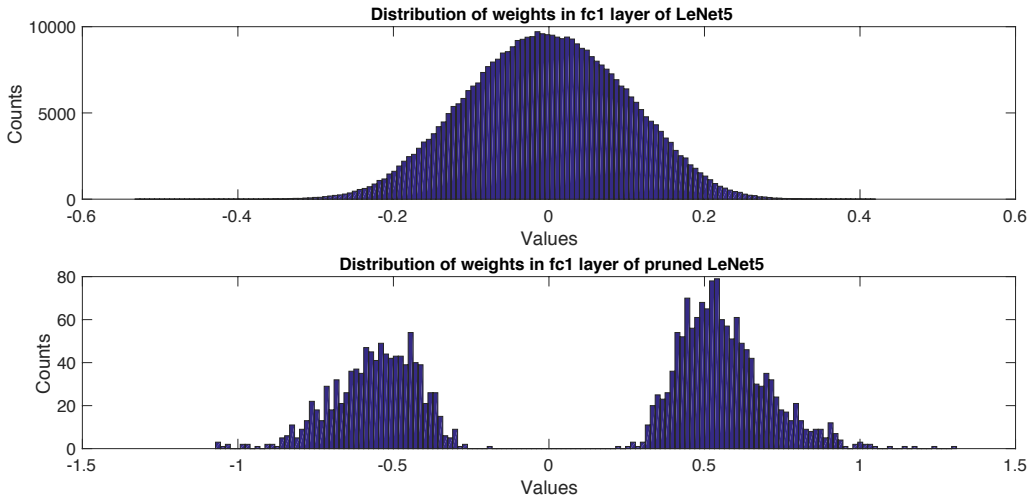

Figure 3.1: Distribution of weights in the first fully-connected layer of pruned and unpruned LeNet5

Layer	cov1	cov2	fc1	fc2
Params	0.5K	25K	400K	5K
Prune %	40%	80%	98%	40%

Table 3.1: Number of parameters of pruned LeNet5-431K using *Determinisitc pruning*, and the pruning includes weights and biases.

size of the original network, which indicates a compression rate of 26.38x.

Table 3.2 shows how the pruning strategy performs on a *CifarNet*. As shown in Table 3.2, each layer of the *CifarNet* achieves a lower pruning percentage compared to *LeNet5*. This is a reasonable result, because recognizing images in *MNIST* is an easier task and the *LeNet5* network contains more redundancies than *CifarNet*.

Layer	cov1	cov2	fc1	fc2	fc3
Params	4.8K	102.4K	885K	74K	2K
Prune %	30%	66%	85%	66%	30%

Table 3.2: Number of parameters of pruned *Cifarnet* using *Determinisitc pruning*, and the pruning includes weights and biases.

3.1.2 Pruning Weights Only

The number of biases of a neural network is normally small compared to the number of weights. More importantly, biases serve as offset values for each individual neuron, these offsets stay invariant to different input images and could have significant impacts on the accuracies of a neural network. It is therefore reasonable to consider pruning only the weights but leaving the biases unpruned.

Similar to the previous section, I compute the layer-wise optimization target of pruning without biases in Equation (3.5).

$$min(L(\mathbf{W_n} \odot \mathbf{M_n} + \mathbf{B_n})) \tag{3.5}$$

As the equation states, now the binary mask matrix only applies on weights variables, the biases are kept unchanged. Table 3.3 and Table 3.4 show the pruning results of *LeNet5* and *CifarNet* respectively.

Layer	cov1	cov2	fc1	fc2
Params	0.5K	25K	400K	5K
Prune(a) %	40%	80%	98%	40%
Prune(b) %	51.92%	79.84%	99.38%	49.90%

Table 3.3: Number of parameters of pruned LeNet5-431K using *Deterministic pruning*. Prune(a) is pruning both weights and biases, Prune(b) is pruning weights only.

Layer	cov1	cov2	fc1	fc2	fc3
Params	4.8K	102.4K	885K	74K	2K
Prune(a) %	30%	66%	85%	66%	30%
Prune(b) %	40%	69%	85%	69%	40%

Table 3.4: Number of parameters of pruned CifarNet using *Deterministic pruning*. Prune(a) is pruning both weights and biases, Prune(b) is pruning weights only.

The results of pruning without biases demonstrate itself to be a more efficient pruning strategy compared to the previous methodology shown in Sec-

tion 3.1.1. Numerically, the compression rate of *LeNet5* when pruned with biases is 26.38x, and now it increases to 37.75x. Similarly, the compression rate of *CifarNet* has increased from 6.10x to 6.19x.

To conclude, pruning with only weights gives better results and later on in this project, all pruning methods only apply on the weights of the neural networks.

3.2 Dynamic Network Surgery

Dynamic network surgery is a pruning method proposed by *Guo et al.* recently [3]. *Guo et al.* proved experimentally that *Dynamic network surgery* outperforms other existing pruning methods by considerable margins [3]. This method combines pruning with splicing. Splicing refers to a procedure where some pruned weights are selected to rejoin the network for the next pruning iteration. The complete procedure of *Dynamic network surgery* is shown in Figure 3.2.

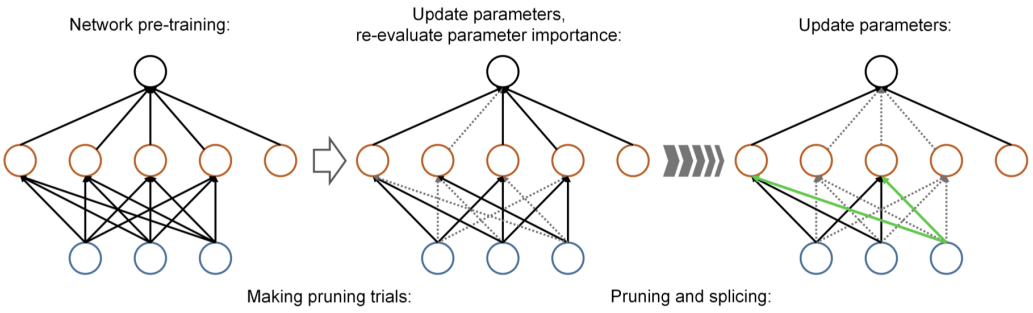

Figure 3.2: Mechanism of *Dynamic network surgery* [3].

In the previous sections, all pruning strategies are deterministic, meaning that a network might suffer from an irretrievable damage once some important weights are lost. In contrast, *Dynamic network surgery* provides a chance for pruned weights to recover. As shown on the rightmost plot in Figure 3.2, the green connections are weights that have been recovered from splicing. The splicing process minimizes the risk of causing an irretrievable

damage to the neural network. Consider the parameters defined previously, the major difference now is on the discriminative function:

$$h_n(\mathbf{W_n}^{(i,j)}) = \begin{cases} 0, & \text{if } t_{an} > |\mathbf{W_n}^{(i,j)}| \\ \mathbf{M_n}, & \text{if } t_{an} \leq |\mathbf{W_n}^{(i,j)}| \leq t_{bn} \\ 1, & \text{if } t_{bn} < |\mathbf{W_n}^{(i,j)}| \end{cases} \tag{3.6}$$

The discriminative function now have two threshold values, namely t_{an} and t_{bn} for a given layer n. Different from the previous discriminative function (Equation (3.2)), for values that stay in between two thresholds, their corresponding binary masks stay unchanged. For weights that have absolute values larger than t_{bn}, their binary masks become one which means these weights are recovered, in other words, these weights are spliced. Weights with absolute values smaller than t_{an} are turned off.

For determining the threshold values, t_{an} and t_{bn}, *Guo et al.* used the following equations in their code:

$$\begin{aligned} t_{an} &= 0.9(u_n + c\sigma_n) \\ t_{bn} &= 1.1(u_n + c\sigma_n) \end{aligned} \tag{3.7}$$

Similar as before, Equation (3.7) made use of the mean and standard deviation at each layer to help determine the pruning thresholds. To test the performance of *Dynamic network surgery*, this pruning method is then applied on the two selected neural networks (*LeNet5* and *CifarNet*). The pruning results of both *LeNet5* and *CifarNet* are displayed in Table 3.5 and Table 3.6 respectively.

The use of *Dynamic network surgery* significantly improves the results of pruning. The compression rate of *LeNet5* is now 49.05x, and the compression rate of *CifarNet* is 17.66x. In the previous section, *Determnisitc pruning* only achieved a compression rate of 37.75x on *LeNet5*, and 6.19x on *CifarNet*. The significant increases in compression rates imply that *Dynamic Network Surgery* is a better pruning method. Pruning happens in a non-deterministic

Layer	cov1	cov2	fc1	fc2
Params	0.5K	25K	400K	5K
Prune(a) %	40%	80%	98%	40%
Prune(b) %	51.92%	79.84%	99.38%	49.90%
Prune(c) %	36.54%	87.90%	99.66%	18.42%

Table 3.5: Number of parameters of pruned LeNet5-431K. Prune(a) is *Deterministic pruning* with both weights and biases, Prune(b) is *Deterministic pruning* with weights only and Prune(c) is *Dynamic network surgery*.

Layer	cov1	cov2	fc1	fc2	fc3
Params	4.8K	102.4K	885K	74K	2K
Prune(a) %	30%	66%	85%	66%	30%
Prune(b) %	40%	69%	85%	69%	40%
Prune(c) %	53%	87%	95%	82%	26%

Table 3.6: Number of parameters of pruned CifarNet. Prune(a) is *Deterministic pruning* with both weights and biases, Prune(b) is *Deterministic pruning* with weights only and Prune(c) is *Dynamic network surgery*.

fashion in *Dynamic Network Surgery*: if a pruned weight finds its importance at later stages of the iterative pruning, it will have a chance to recover. *Guo et al.* proposed inspecting the values of weights as a representative of their importance. As shown in Equation (3.7), if a pruned weight has a relatively large value after retraining, it will be turned on again.

3.3 Comparison to Existing Works

In this section, I would like to compare the implemented pruning methods to their original implementations.

Table 3.7 shows how the original implementations of various pruning methods compared to my implementations. (a) is *Deterministic pruning* with weights and biases, (b) is *Deterministic pruning* with weights only. (c) is my implementation of the *Dynamic network surgery* method. (*Han*) is the *Deterministic pruning* strategy used by *Han et al.* in their *Deep Compression* framework [4]. (*Guo*) is the original implementation of *Dynamic network*

Model	Layer	Params	%(*Han*[4])	%(*Guo*[3])	%(a)	%(b)	%(c)	%(d)
LeNet5	cov1	0.5K	66%	14.2%	60%	48.1%	63.5%	10%
	cov2	25K	12%	3.1%	20%	20.2%	12.1%	6%
	fc1	400K	8%	0.7%	2%	0.6%	0.4%	5.05%
	fc2	5K	19%	4.3%	60%	50.1%	82.6%	5.84%
	total	431K	8%	0.9%	3.8%	2.4%	2.0%	0.89%
	CR	-	12.5x	108x	26x	42x	49x	111x
	ER	-	0.8%	0.91%	0.64%	0.64%	0.64%	0.91%

Table 3.7: *LeNet5* pruning summary, CR is the compression rate, ER is the error rate. (*Han*) is *Deterministic pruning* used by *Han et al.*. (*Guo*) is the original *Dynamic network surgery* implemented by *Guo et al.*. (a), (b), (c), (d) are my implementations of various methods. (a) is *Deterministic pruning* with weights and biases, (b) is *Deterministic pruning* with weights only, (c) is *Dynamic network surgery*, (d) is also *Dynamic network surgery* but with the same error rate as (*Guo*).

surgery [3]. The first few rows show the percentages of parameters left at each layer of a *LeNet5* model. The row named *total* displays the total percentages of parameters that are left in the entire neural network. The row *CR* shows the compression rates achieved using various techniques and *ER* illustrates the error rates of the pruned networks. The compression rates achieved in my implementations are generally better than *Han et al.'s* both in terms of compression rate and test accuracies. Comparing to the original implementation of *Dynamic network surgery* ((*Guo*)), (c) shows a smaller compression rate but also lower error rate. Since the targeting network and dataset are the same, this drop in compression rate can be seen as a trade-off between compression rate and error rate. When I choose to tolerate an error rate of 0.91%, as shown in (d), the compression rate reaches 111x, which is very close to the original implementation proposed by *Guo et al.* (108x).

Table 3.8 shows the pruning results on *CifarNet*, all the pruning methods are implemented by myself. (a) is *Deterministic pruning* with weights and biases, (b) is *Deterministic pruning* with only weights. (c) is my implementation of the *Dynamic network surgery* method. *Han et al.* and *Guo et al.* do not provide results for *CifarNet*. Nonetheless, this comparison demonstrates that *Dynamic network surgery* is superior to other pruning strategies.

Model	Layer	Params	(a)	(b)	(c)
CifarNet	cov1	4.8K	70%	60%	47%
	cov2	102.4K	34%	31%	13%
	fc1	885K	15%	15%	5%
	fc2	74K	34%	31%	18%
	fc3	2K	70%	60%	74%
	total	1068K	18.5%	17.9%	7.0%
	CR	-	5.41x	5.58x	14.3x
	ER	-	18%	18%	18%

Table 3.8: *CifarNet* pruning summary, CR is the compression rate, ER is the error rate. (a) is *Deterministic pruning* with weights and biases, (b) is *Deterministic pruning* with weights only, (c) is *Dynamic network surgery*.

In this section, by comparing various pruning strategies, two important observations can be summarized. First, biases of a neural network should not be pruned since they are only offset values of each neuron. Second, *Dynamic network surgery* is proven to have the best performance. Intuitively, the splicing technique allows recovering of pruned weights, which helps to search more exhaustively for a minimal network topology.

Chapter 4

Optimized Pruning Strategies

In this chapter, I will mainly describe two novel pruning strategies. The first proposed pruning technique is called *Gradient profiling*, this method benefits pruning with limited retraining resources. The second method is *Regularization aided pruning*. By putting extra regularization terms into the cost function of a neural network, later in this chapter, I demonstrate that regularizers encourage sparsity of a neural network and thus induce better pruning results.

4.1 Gradient Profiling

All pruning methods in the previous chapter only focus on pruning parameters based on absolute values. These implementations, however, have two problems. First, weights with small absolute values are pruned, but it is possible for a weight with small absolute value to stay at an important location so pruning it might have a large impact on accuracy. Second, large retraining time is allowed. Consider an epoch to be a complete traverse of the training dataset. *LeNet5* is allowed to a maximum retraining of 200 epochs, and *CifarNet* is allowed to have 300 epochs. The number of epochs allowed for retraining is identical to the number of epochs spent in training the unpruned

baseline models. In this section, I would like to propose a new method called *Gradient profiling* for finding weights that are small but important for test accuracies using limited retraining resources.

4.1.1 Method Description

Pruning with *Gradient profiling* is nearly identical to *Dynamic network surgery*, apart from that it inspects the gradient changes for one epoch. The reason for inspecting all gradients of one epoch is to find enough observations on the entire training dataset. The description of the method is illustrated in Figure 4.1.

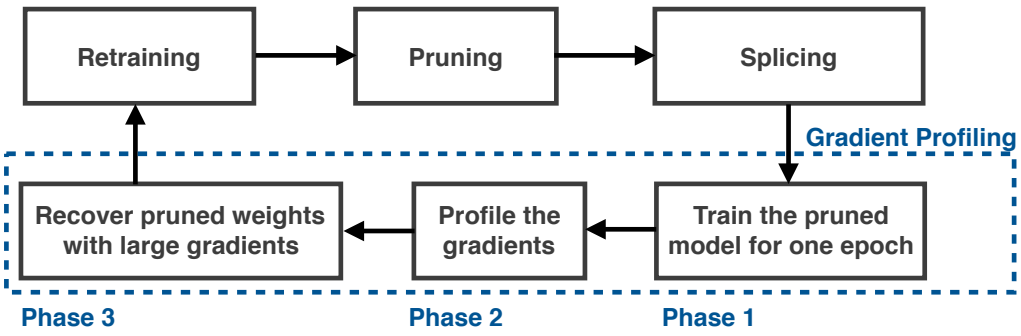

Figure 4.1: Pruning with *Gradient profiling*. The top row is identical to *Dynamic network surgery*, the second row is the three phases of *Gradient profiling*

The *Gradient profiling* part is broken down into three phases, in the first phase, a pruned neural network is trained only for one epoch. The second phase collects the gradients from that one epoch training. The third phase is to recover pruned weights based on profiled gradients data. The amount of pruned weights that are recovered is controlled by a arbitrarily defined hyperparamter. In this section, the amount of recovered weights is equal to 10% of the total weights remained. The hypothesis is that the gradients of a network serve as indications for weights importance. If an important weight is pruned away, the neural network would keep passing this weight a large

gradient. Later in the gradient profiling mechanism, pruned weights with large gradients are recognized as important weights and recovered.

From a mathematical point of view, let $\{\mathbf{G_n} : 0 \leq n \leq N\}$ to represent the profiled gradients from phase 2 in Figure 4.1. A discriminative function $h_{gn}(*)$ is used to determine a new mask $\{\mathbf{GM_n} : 0 \leq n \leq N\}$ for each layer.

$$\mathbf{GM_n} = h_{gn}(\mathbf{G_n}^{(i,j)}) = \begin{cases} 0, & \text{if } t_{gn} < |\mathbf{G_n}^{(i,j)}| \\ 1, & \text{if } t_{gn} \geq |\mathbf{G_n}^{(i,j)}| \end{cases} \tag{4.1}$$

t_{gn} is a threshold value determined arbitrarily, as mentioned before, the number of weights recovered from this *Gradient profiling* process is equal to a tenth of the unpruned weights. As the number of unpruned weights keep decreasing during the iterative pruning process, the amount of weights recovered from *Gradient profiling* would keep decreasing as well. The loss function therefore becomes:

$$min(L_n(\mathbf{W_n} \odot (\mathbf{M_n} + \mathbf{GM_n}) + \mathbf{B_n})) \tag{4.2}$$

This optimization target now embraces *Gradient profiling*. Notice the optimization target has a term – $\mathbf{W_n} \odot (\mathbf{M_n} + \mathbf{GM_n})$. The elements in $(\mathbf{M_n} + \mathbf{GM_n})$ can only be 1 or 0, because only pruned weights can be reactivated to one in GM_n and pruned weights are zeros in $\mathbf{M_n}$. The training strategy follows the same change:

$$\begin{aligned} \mathbf{W_n} \leftarrow & \mathbf{W_n} - \alpha \frac{\partial}{\partial(\mathbf{W_n} \odot (\mathbf{M_n} + \mathbf{GM_n}) + \mathbf{B_n})} \\ & L(\mathbf{W_n} \odot (\mathbf{M_n} + \mathbf{GM_n}) + \mathbf{B_n}) \end{aligned} \tag{4.3}$$

Follow the same setup as before, α is the learning rate and Equation (4.3) shows how weights are updated when both masks are applied.

4.1.2 Gradient Profiling and Retraining

To fully understand the effect of *Gradient profiling*, I would like to compare it to *Dynamic network surgery* on various levels of retraining. It is common to combine pruning methods with retraining, however, some researchers argue that retraining takes a significant amount of time and thus should be avoided [47]. So, in this section, I consider the following three retraining strategies:

1. No retrain after pruning.
2. Retrain for 10 epochs after pruning.
3. Retrain for 300 epochs after pruning.

Figure 4.2 is the results when both *Gradient profiling* and *Dynamic network surgery* are applied on *LeNet5* without any retraining. The horizontal axis displays the amount of weights that are left compared to the original network as a ratio. For instance, 0.4 means 40% of the parameters are kept and the other 60% are pruned away. The vertical axis shows test accuracies when the network is compressed to various sizes. The dashed line represents *Gradient profiling pruning* and the solid line is *Dynamic network surgery*. As expected, because of the fact that some important weights are recovered by *Gradient profiling*, it shows a greater test accuracy than *Dynamic network surgery* when the size of the neural network is compressed below 0.4 of its original size. When the size of the neural network is relatively large (size above 0.5), the loss of test accuracies are not apparent and therefore both methods show similar performances.

As mentioned previously, retraining recovers the test accuracies by forcing the remaining parameters to learn to adapt with each other. However, the amount of time spent on retraining is a major drawback of this method. It is interesting to observe how pruning strategies are combined with *limited retraining*. I define *limnited retraining* as a concept that only a very small number of retraining epochs are allowed after pruning. In this case, I pick to retrain only 10 epochs after pruning. Two different pruning methods, *Gradient profiling* and *Dyamic network surgery*, are considered and compared on

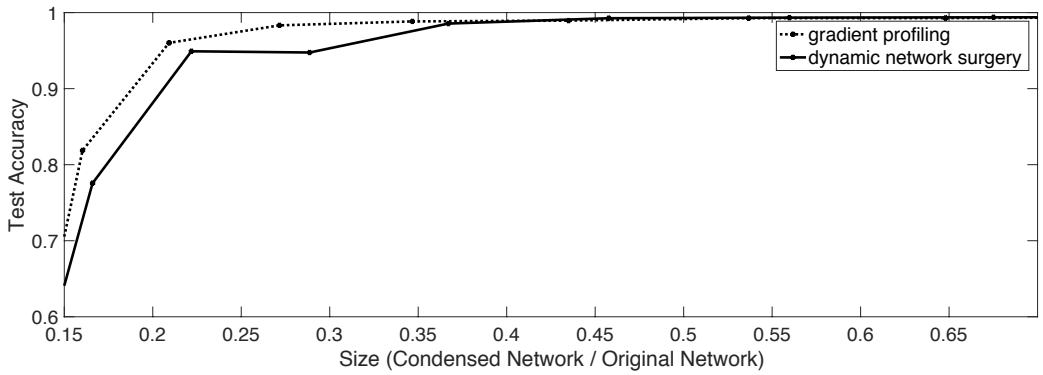

Figure 4.2: *Gradient profiling* and *Dynamic network surgery* without retraining. The figure shows how various compressions affect test accuracies.

the *LeNet5* model. Figure 4.3 shows the results when 10 epochs of retraining is applied. As expected, the test accuracy of *Gradient profiling* drops at a slower rate compared to *Dynamic network surgery*. It is clear that between the size of 0.2 and 0, the test accuracies of *Gradient profiling* is significantly higher.

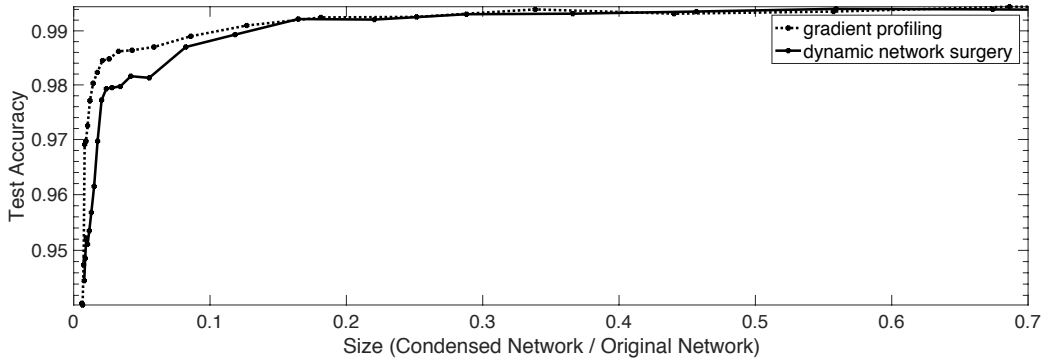

Figure 4.3: *Gradient profiling* and *Dynamic network surgery* with 10 epochs retraining. The figure shows how various compressed sizes affect test accuracies.

Finally, I combine pruning with retraining of 300 epochs, and tested them on *LeNet5*. Table 4.1 shows the pruning results when using *Gradient profiling*. The compression rate is 48x, which is slightly lower than the compression rate of *Dynamic network surgery* (49x). This proves that, if retrained to convergence, *Gradient profiling* is not superior to *Dynamic network surgery*.

The large amount of retraining time give weights the ability to learn to adapt with each other. Especially for *Dynamic network surgery*, incorrect pruning has a chance to be recovered, which gives parameters a larger chance to learn to co-adapt with each other. The large weights now have enough time to learn , and previously important small weights might lose their importance because large valued weights have already had enough time to adjust their values.

Layer	cov1	cov2	fc1	fc2
Params	0.5K	25K	400K	5K
Prune(a) %	44.20%	91.08%	99.31%	28.87%
Prune(b) %	36.54%	87.90%	99.66%	18.42%

Table 4.1: Number of parameters of pruned LeNet5-431K. Prune(a) is *Gradient profiling*, Prune(b) is *Dynamic Network Surgery*.

To summarize, *Gradient profiling* can be a very efficient pruning strategy when retraining resource is limited. Its enhancement on compression rate becomes limited if long retraining time is allowed. In this project, since the topic is to construct a compression pipeline that achieves the best compression rate, I used *Dynamic network surgery* rather than *Gradient profiling* in later sections. However, the importance and effectiveness of *Gradient profiling* under limited retraining resources still worth further evaluations.

4.2 Regularization Aided Pruning

Regularization methods are popular for preventing the neural networks from overfitting. Some regularization methods, such as l_1 norm, l_2 norm and *Shakeout*, achieve regularization by encouraging sparsities in the model. The use of regularizers leads to a sparse model, and this might benefit the pruning process. *Han et al.* previously showed how to combine regularization with *Deterministic pruning* [45], but the use of regularizers is absent in *Dynamic Network Surgery* [3]. In this section, I would like to investigate how regularization could be combined with *Dynamic Network Surgery*.

4.2.1 l_1 and l_2 Norms

l_1 and l_2 norms, also known as least absolute errors (LAE) and least squares respectively, are popular regularization terms that could be appended to a neural network's cost function [48]. Mathematically, the l_p norm of a given vector v is:

$$||v||_p = (\sum_{i=1}^{n} |v_i|^p)^{\frac{1}{p}} \tag{4.4}$$

l_1 and l_2 norms have been proven to be robust to outliers in data and also encourages sparsity via training [48]. The intuitive explanation is to view these additional norms as penalties to large weights, so they prevent particular weights from dominating network. For the combined l_1 and l_2 norms, I add the following term into the cost function of a neural network:

$$\lambda_1|w| + \lambda_2|w|^2 \tag{4.5}$$

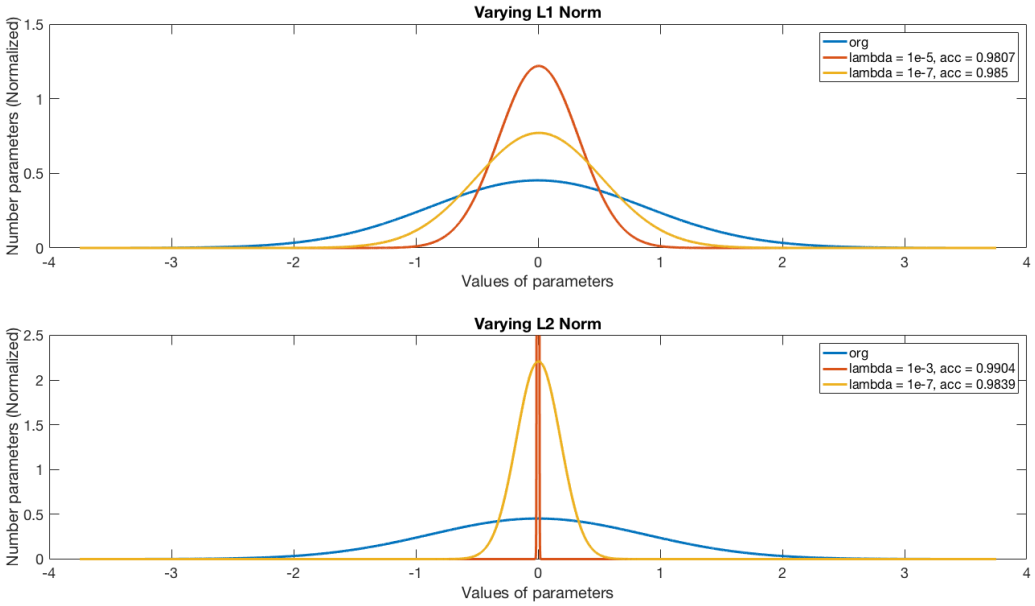

Figure 4.4: Effects of $l1$ and $l2$ norms with different hyperparameters.

$\lambda 1$ and $\lambda 2$ are two hyperparamters that are normally chosen arbitrarily. To view the effect of various norms on the weights distribution of a given neural

network, Figure 4.4 shows how the weights distribution of a *LeNet5* model varies when applied with different norms. With larger $\lambda 1$ and $\lambda 2$ values, the weights distributions are more concentrated to zero.

Notice in Figure 4.4, $l1$ and $l2$ norms are shown separately, meaning that λ_2 and λ_1 in Equation (4.5) are set to zeros respectively when chosen to demonstrate the effect of $l1$ norm and $l2$ norm respectively. For the first plot in Figure 4.4, l_2 norm is set to zero and it is obvious that when λ_1 reaches $1e^{-5}$, the weights distribution are more concentrated at zero (red) compard to the original weights distribution (blue). For the original distribution, both λ_1 and λ_2 are zeros. A similar pattern is observed for the second plot in Figure 4.4: with a larger λ_2 value, the weights distribution is more concentrated at zero.

For the hyperparameters of the regularizer, I pick $\lambda_1 = 1e^{-4}$ and $\lambda_2 = 1e^{-7}$ for *LeNet5*. The model is then pruned and retrained to an error rate of 0.64%. The detailed layer-wise pruning information is shown in Table 4.2 and compared to the best performance pruning method from the previous chapter (*Dynamic network surgery*).

Layer	cov1	cov2	fc1	fc2
Params	0.5K	25K	400K	5K
Prune(a) %	22.11%	92.47%	99.70%	32.91%
Prune(b) %	36.54%	87.90%	99.66%	18.42%

Table 4.2: Number of parameters of pruned LeNet5-431K. Prune(a) is *Regularization aided pruning* with $l1$ and $l2$ norms, $\lambda_1 = 1e^{-4}$ and $\lambda_2 = 1e^{-7}$. Prune(b) is *Dynamic network surgery*.

For *CifarNet*, $\lambda_1 = 1e^{-5}$ and $\lambda_2 = 1e^{-5}$ are selected after exploring a range of values. The layer-wise pruning results of *CifarNet* are shown in Table 4.3. This trained *CifarNet* remains a test accuracy of 0.82, which is the same as the original network.

From the results in both Table 4.2, Table 4.3, *Regularization aided pruning* demonstrates its performance by showing greater compression rates on both networks. For the *LeNet5* model, the compression rate now reaches 63x and

Layer	cov1	cov2	fc1	fc2	fc3
Params	4.8K	102.4K	885K	74K	2K
Prune(a) %	45%	88%	96%	76%	39.6%
Prune(b) %	53%	87%	95%	82%	26%

Table 4.3: Number of parameters of pruned CifarNet. Prune(a) is *Regularization aided pruning* with $l1$ and $l2$ norms, $\lambda_1 = 1e^{-5}$ and $\lambda_2 = 1e^{-5}$. Prune(b) is *Dynamic Network Surgery*.

CifarNet achieves a compression rate of 15.4x. It can be concluded that the use of regularizers encourage sparsity in the network and thus provide better pruning results.

4.3 Summary of Pruning Methods

In this section, I would like to summarize all pruning methods implemented in this project. Some pruning methods haven been compared to their original implementations in Section 3.3, this section only focuses on comparing my implementations of various pruning strategies on selected datasets.

Model	Layer	Params	(a)	(b)	(c)	(d)	(e)
LeNet5	cov1	0.5K	60%	48.1%	63.5%	65.8%	78.9%
	cov2	25K	20%	20.2%	12.1%	8.9%	7.5%
	fc1	400K	2%	0.6%	0.4%	0.7%	0.3%
	fc2	5K	60%	50.1%	82.6%	72.2%	67.1%
	total	431K	3.8%	2.4%	2.0%	2.0%	1.6%
	CR	-	26x	42x	49x	48x	63x
	ER	-	0.64%	0.64%	0.64%	0.64%	0.64%

Table 4.4: LeNet5 Pruning Summary, CR is the compression rate, ER is the error rate. (a) is *Deterministic pruning* with weights and biases, (b) is *Deterministic pruning* with weights only, (c) is *Dynamic network surgery*, (d) is *Gradient profiling* and (e) is *Regularization aided pruning*.

Similar to the previous setup, Table 4.4 shows the pruning results. The percentiles showing for each layer represents the amount of parameters left in that layer. To prune *LeNet5*, it is important to achieve a small percent-

Model	Layer	Params	(a)	(b)	(c)	(d)
CifarNet	cov1	4.8K	70%	60%	47%	55%
	cov2	102.4K	33%	31%	13%	12%
	fc1	885K	15%	15%	5%	4%
	fc2	74K	33%	31%	18%	24%
	fc3	2K	70%	60%	74%	60%
	total	1068K	18.5%	17.9%	7.0%	6.5%
	CR	-	5.41x	5.58x	14.3x	15.4x
	ER	-	18%	18%	18%	18%

Table 4.5: *CifarNet* Pruning Summary, CR is the compression rate, ER is the error rate. (a) is *Deterministic pruning* with weights and biases, (b) is *Deterministic pruning* with weights only, (c) is *Dynamic network surgery* and (d) is *Regularization aided pruning*.

age on the first fully-connected layer (fc1), since it contains a large number of parameters. Method (a) is *Deterministic pruning* with weights and biases, (b) is *Deterministic pruning* with weights only, (c) is *Dynamic network surgery*, (d) is *Gradient profiling* and (e) is *Regularization aided pruning*. The proposed pruning strategy, *Regularization aided pruning*, achieves the best compression rate: it shows a 1.3x increase in compression rate compared to *Dynamic network surgery* on the *LeNet5* model.

Similarly, the pruning summary of *CifarNet* demonstrated that *Regularization aided pruning* achieves the best compression result. The compression rate now reaches 15.4x without any loss of test accuracies. The following important observations can be summarized from comparing a range of pruning methods:

1. Pruning with only weights is more efficient.
2. Non-deterministic pruning achieves better compression rates since pruned weights now have chance to recover.
3. Gradients can serve as indications for identifying important weights.
4. The use of regularizers helps pruning by encouraging sparsity in the network, and thus *Regularization aided pruning* shows the best pruning results.

Chapter 5

Quantization

The objective of this chapter is to discover an efficient number representation system. An efficient number representation system reduces the number of bits required to represent individual parameters, and thus offers a compression on the top of pruning. In this section, I aim to investigate various existing quantization methodologies to further compress the chosen neural networks. Traditionally, a neural network is trained on GPUs in 32-bit floating-point representation, however, this representation becomes problematic on low power devices. First, using 32 bits to store a single parameter is proven to be redundant [49]. Second, floating-point arithmetic operations are generally more power-consuming than fixed-point operations. Methods such as *Weights sharing* are discussed in this section [4]. Low-precision arithmetics, including *Fixed-point arithmetic* and *Dynamic fixed-point arithmetic*, are also compared and evaluated.

5.1 Weights Sharing

Weights sharing compresses the bit-width of parameters using a codebook and stores only the indexes. Similar to *HashNet* [32], *Weights sharing* used in *Deep Compression* [4] can reduce the number of bits required to represent

a parameter by employing a hash function and store weights as hash keys (indexes). Designing the hash function firstly requires grouping of weights. In this case, following the implementation of *Han et al.* [4], I performed *K-means* clustering [50] to group weights into n clusters, and therefore n centroid values are produced.

The inference and retraining of grouped weights are illustrated in Figure 5.1. Consider a small four-by-four weight matrix, and weights are grouped into 4 groups. Only the cluster indexes and codebook (effective weights) need to be stored for network inference. For this particular example, to represent 4 clusters, each weight can be represented using only a 2-bit cluster index. The use of index and codebook directly compressed the size of a neural network, and *Han et al.* have summarized an equation for the compression rate [4]. Given k clusters, $\log_2(k)$ bits are required for encoding the index. Consider a neural network with N parameters, and each parameter is b bits wide, the following equation summarizes the compression rate CR:

$$CR = \frac{Nb}{N\log_2(k) + kb} \tag{5.1}$$

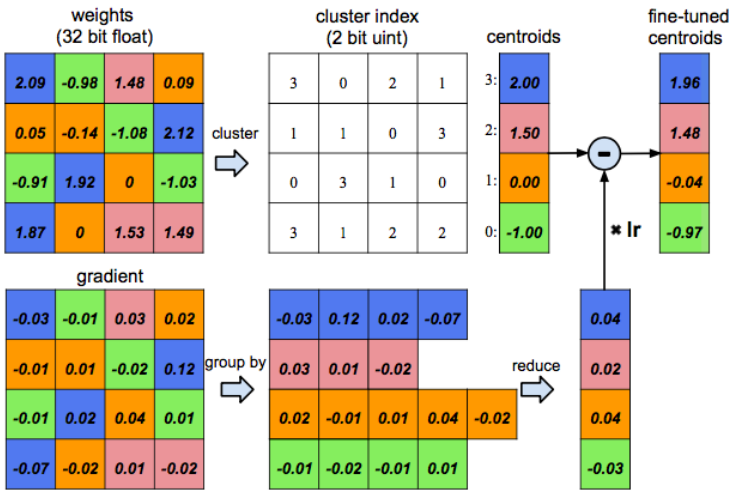

Figure 5.1: Weights sharing: training (bottom) and inference (top) [4].

This *Weights sharing* technique is applied on both *LeNet5* and *CifarNet*.

Table 5.1 shows the quantization results of *Weights sharing* on the targeting neural networks. *Weights sharing* is also combined with retraining to bring back the lost test accuracies. Table 5.1 uses two rows for each neural network, one row represents the test accuracies before retraining and one row shows the accuracies after retraining. First, as expected, test accuracies are significantly larger when the number of clusters is large. Second, retraining is demonstrated to be an efficient method of brining back the lost accuracies. *Weights sharing* worked very well on *LeNet5*, with 8 clusters at each layer, *LeNet5* achieves no accuracy loss after retraining. It is important to note that, for 8 clusters, only 3 bits are required to represent the cluster index of each parameter.

Number of clusters	64	32	16	8	4	2
Number of bits	6	5	4	3	2	1
LeNet5, 32-bit floating point accuracy: 99.36%						
Before Retrain	99.36%	99.36%	99.36%	99.29%	98.90%	94.67%
After Retrain	99.36%	99.36%	99.36%	99.36%	99.15%	98.66%
CifarNet, 32-bit floating point accuracy: 82.00%						
Before Retrain	79.94%	79.81%	72.78%	69.00%	25.25%	9.94%
After Retrain	82.21%	82.01%	82.02%	79.05%	65.70%	17.2%

Table 5.1: *Weights sharing* summary for *LeNet5* and *CifarNet*.

CifarNet is a larger network compared to *LeNet5*. As a result of its larger size, the network requires a larger number of clusters at each layer. According to Table 5.1, at a cluster count of 16, the network achieves no accuracy loss. This means, each weight parameter can be represented using 4 bits, and the neural network is able to achieve the same accuracy as the uncompressed one.

Although *Weights sharing* demonstrated good compression rates on both networks, the arithmetic operations are still in floating-point. Since floating-point arithmetic operations are power consuming, turning into small fixed-point numbers can reduce both energy consumption and circuitry area. In *Han et al.'s* imeplementation, they used *Weights sharing* on top of *Fixed-point quantization*. In later sections, the focus stays on exploring fixed-point

based quantization methods.

5.2 Fixed-point Quantization

One simple quantization strategy when using fixed-point arithmetic is to truncate the least significant bits (LSBs) but leave the most significant bits (MSBs) unchanged. This method is straightforward and easy to implement.

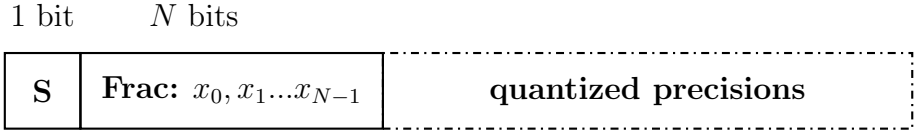

Figure 5.2: Number representation system for fixed-point quantization with 1-bit sign and N-bit fraction.

Figure 5.2 shows the number representations of a quantized number. The number representation system has 1 bit for sign and N bits for fractions. The number of fractional bits can be therefore changed to track numbers to various levels of precisions. Equation (5.2) shows how the number representation of Figure 5.2 can be converted to decimal representation. In this case, S stands for the sign bit, N is the total number of fractional bits. x_i represents the fractional bit values. The representation used in this section is a traditional signed fixed-point representation with N-bit fractions [51].

$$D = (-1)^S (\sum_{i=0}^{N-1} 2^{-i} x_i) \tag{5.2}$$

Table 5.2 shows how fixed-point quantization performs on *LeNet5* and *Cifar-Net* when the bit-width changes. It is important to note that, the bit-width in this case refers to total bit-width. Total bit-width is $N + 1$, including N-bit fractions and 1-bit sign.

Bit width	32-bit	16-bit	8-bit	4-bit	2-bit
LeNet5, 32-bit floating point accuracy: 99.36%					
Before Retrain	98.98%	97.08%	75.63%	12.74%	9.78%
After Retrain	99.36%	99.36%	99.26%	98.88%	9.78%
CifarNet, 32-bit floating point accuracy: 82.00%					
Before Retrain	32.84%	21.58%	12.16%	10.22%	9.97%
After Retrain	82.00%	79.21%	68.76%	65.76%	9.95%

Table 5.2: Fixed-point quantization summary for *LeNet5* and *CifarNet*.

The results in Table 5.2 suggests that 16 bits for *LeNet5* achieves best quantization results without any accuracy loss. 32-bit is redundant since the test accuracy remains the same as 16-bit when retraining is applied. Similar to the previous setup, retraining occurs after quantization to bring back test accuracies. Retraining takes a very important role in fixed-point quantization. For instance, when *LeNet*5 is quantized to 4-bit (3-bit fraction and 1-bit sign), its test accuracy before retraining is only 12.74%, but increases to 98.88% after retraining. As illustrated in Table 5.2, *CifarNet* only recovers to an accuracy of 79.21% when the precision is 16-bit and 82.00% when the precision is 32-bit. Since *CifarNet* is a larger neural network compared to *LeNet5*, it requires more bits to retain its test accuracy. To conclude, for *fixed-point quantization*, *LeNet5* is able to use 8 bits to represent the original neural network and 32 bits are needed for *CifarNet* to retain its test accuracies.

5.3 Dynamic Fixed-point Quantization

Dynamic fixed-point arithmetic is a variant of fixed-point arithmetic, where a number of parameters are grouped with a fixed scaling factor [52]. Previously, *Courbariaux et al.* has implemented both *LeNet5* and *CifarNet* using *Dynamic fixed-point* arithmetic [53]. They proposed an algorithm (Algorithm 1) for computing the scaling factors. The scaling factors define the dynamic range for a selected group of weights.

To use *Dynamic fixed-point* in a neural network, Algorithm 1 is applied on each layer of parameters to help weights with various dynamic ranges. Algorithm 1 works by setting an arbitrarily defined overflow rate, in this project, the overflow rate is set to be 10^{-3}. It then inspect the weights matrix and converges to a scaling factor after a few iterations. It is possible for weights in convolutional layers to have a completely different dynamic range from weights in fully connected layers. The number representation system is the same as *fixed-point* arithmetic shown in Figure 5.2. The major difference now is each group of weights has a fixed dynamic range. In Equation (5.3), it shows how a *Dynamic fixed-point* number can be converted to a decimal value. DR represents the fixed dynamic range, consider the scaling factor (s_t) computed in Equation (5.3), the following relation holds: $s_t = 2^{DR}$. These dynamic ranges, or can be viewed as scaling factors, are stored globally for each layer and thus saves the bit-width for the number representation system.

$$D = (-1)^S 2^{DR} (\sum_{i=0}^{M-1} 2^{-i} x m_i) \tag{5.3}$$

Algorithm 1 Scaling Factor Update

Initialize: matrix M, scaling factor s_t and maximum overflow rate r_{max}
while s_t doe not converge **do**
  if the overflow rate of $M > r_{max}$ **then**
    $s_{t+1} \leftarrow 2s_t$
  else if the overflow rate of $2M \leq r_{max}$ **then**
    $s_{t+1} \leftarrow s_t/2$
  else
    $s_{t+1} \leftarrow s_t$
  end if
end while

Dynamic Fixed-point quantization has been applied on both $LeNet5$ and $CifarNet$, the results are shown in Table 5.3. For each bit-width count, it includes the sign bit but excludes the dynamic range. If a bit-width of 4 is given, this means it has 1 bit for sign, 3 bits for fractions and a dynamic range that is globally defined for a given layer. Bit-width refers to the num-

ber of total bits required to represent a number using *Dynmaic fixed-point* arithmetic, and the dynamic range is stored as a layer-wise global value. As expected, *LeNet5* reaches the original test accuracy at a bit-width of 4, and *CifarNet* achieves no accuracy loss after retrain at a bit-width of 8. These results agree with the experimental results achieved by *Courbariaux et al.* [53] and *Gysel et al.* [5]. However, the test accuracy of retrained *LeNet5* never reaches its original test accuracy (99.36%). This indicates that *Dynamic fixed-point* hurts test accuracy in a way that even retraining cannot bring back the lost test accuracies. In both *Courbariaux et al.* and *Gysel et al.*'s work, they used a $LeNet5$ with an original test accuracy of 99.17% [53, 5], so they did not observe this effect. Because of the high original test accuracy, my implementation of *LeNet5* is more sensitive to test accuracy deterioration.

Bit width	32-bit	16-bit	8-bit	4-bit	2-bit
LeNet5, 32-bit floating point accuracy: 99.36%					
Before Retrain	88.40%	88.32%	86.92%	79.10%	71.26%
After Retrain	99.29%	99.30%	99.31%	99.31%	99.00%
CifarNet, 32-bit floating point accuracy: 82.00%					
Before Retrain	58.99%	26.25%	10.48%	9.98%	9.98%
After Retrain	82.00%	82.12%	82.03%	72.18%	30.53%

Table 5.3: *Dynamic fixed-point quantization* summary for *LeNet5* and *CifarNet*.

By observing the dynamic range of fixed-point quantization, I can practically find the reason for this loss of accuracy caused by using *Dynamic fixed-point*. Consider the scaling factor determined using Algorithm 1, this scaling factor restricts the range of values that this arithmetic could represent. For instance, a scaling factor of 0.5 is determined for the first fully connected layer by inspecting the pre-quantized *LeNet5* model. However, for a $LeNet5$ model that is quantized using fixed-point arithmetic and achieved no accuracy loss, its maximum value in that layer reaches 0.406 but minimum value reaches -0.55. In this case, if *Dynamic fixed-point* is applied with a scaling factor of 0.5, values that are lower than -0.5 will never be reached. The hypothesis is

that the loss of dynamic range can potentially affect retraining since now the quantized weights is restricted to certain numerical ranges. This hypothesis is not theoretically proven, but is hinted by comparing both the test accuracies and numerical ranges of *Fixed-point* arithmetic and *Dynamic fixed-point* arithmetic. A theoretical proof of this hypothesis is beyond the scope of this project, but could be evaluated in future works.

For the experimental results shown in Table 5.3, *LeNet5* is able to represent the parameters using 4 bits and achieve a test accuracy of 99.31%. Although it suffers from a slight test accuracy drop, it offers a much better quantization result. For *CifarNet*, it is able to quantize to a bit-width of 8 with no loss of test accuracies.

Chapter 6

Optimized Quantization Strategies

In the previous chapter, a number of existing quantization methods have been implemented and evaluated. In this chapter, I would like to propose some new quantization methods for compressing neural networks. First, a *Customized floating-point* arithmetic is proposed and it overcomes certain limitations of *Dynamic fixed-point* arithmetic and also showed better performance compared to *Dynamic fixed-point*. Second, to quantize a pruned model, I applied some bi-centralized arithmetics to fully explore the weights distribution by re-centralizing the precision centres of arithmetics.

6.1 Customized Floating-point Quantization

In this section, I propose a new number representation system that stands in the middle between *Dynamic fixed-point* arithmetic and *Floating-point* arithmetic.

In *Dynamic fixed-point*, each number is represented as: $(-1)^S 2^{-DR} \sum_{i=0}^{M-1} 2^i x_i$. M denotes the mantissa width, S is the sign bit, DR is the dynamic range and x are the mantissa bits [53, 5]. At various layers of the neural network,

Dynamic fixed-point assigns fixed-point numbers with different layer-wise dynamic ranges (DR). This DR value stays unchanged for the grouped weights in one layer. In contrast, *Floating-point arithmetic* has a changing exponent for each individual weight, and therefore the decimal point is floating. A normal floating-point number can be expressed as $(-1)^S 2^{exp-offset} \sum_{i=0}^{M-1} 2^i x_i$. In this case, an *offset* value is included to help the arithmetic to cover a large range of values [51].

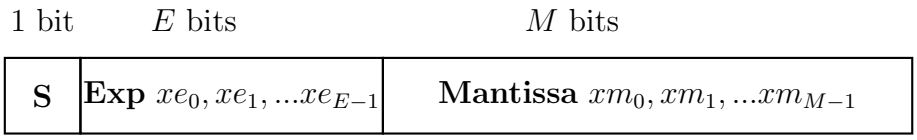

Figure 6.1: Number representation system for customized floating-point quantization with e-bit exponent and m-bit mantissa.

Different from both arithmetics, *Customized floating-point* arithmetic only needs to track values smaller than 1, so the *offset* value in normal floating-point is eliminated. Consider the number representation system in Figure 6.1, the following equation summarizes its conversion to a decimal value:

$$D = (-1)^S 2^{-\sum_{i=0}^{E-1} 2^{-i} xe_i} (\sum_{i=0}^{M-1} 2^{-i} xm_i) \tag{6.1}$$

In Equation (6.1), D represents expected decimal value, S is the sign bit, M is the mantissa width and E is the exponent width. xm_i are the mantissa bits and xe_i are the exponent bits.

As shown in Equation (6.1), each individual weight now takes its own dynamic range, so this is different from *Dynamic fixed-point* where only a fixed dynamic range is allowed for a group of parameters. More importantly, the dynamic range now is restricted to be lower than 1, this makes sure the arithmetic tracks to lowest possible precisions given a limited bit-width.

Table 6.1 shows the quantization results. Using only 6 bits, *Customized floating-point* is able to generate a retrained *LeNet5* that has no accuracy

loss. For *CifarNet*, a bit-width of 8 is required for it to recover to the original accuracy. In Table 6.1, the bit-width includes both a sign bit and a 2-bit exponent. The exponent is chosen to be 2 bits for both *LeNet5* and *CifarNet*. For instance, given a bit width of 6 bits, it consumes 1 bit for sign, 2 bits for exponent and only 3 bits are left for mantissa.

Bit width	32-bit	16-bit	8-bit	6-bit	4-bit
LeNet5, 32-bit floating point accuracy: 99.36%					
Before Retrain	99.36%	99.36%	99.31%	98.4%	9.79%
After Retrain	99.36%	99.36%	99.36%	99.36%	11.35%
CifarNet, 32-bit floating point accuracy: 82.00%					
Before Retrain	34.54%	25.20%	19.15%	15.17%	10.35%
After Retrain	82.02%	82.00%	82.10%	79.82%	70.73%

Table 6.1: *Customized floating-point* quantization summary for *LeNet5* and *CifarNet*, with 2-bit exponent and various mantissa widths.

The results in Table 6.1 demonstrate a good compression rate, however, using *Customized floating-point* arithmetic, the width of exponent has to be determined beforehand, in this case, the arithmetic utilizes a 2-bit exponent. For the results in Table 6.1, only the mantissa bit-width is changing. For an increasing mantissa width, the test accuracy increases as expected. The test accuracy of quantized models stopped increasing after reaching the original test accuracy. At a bit-width of 6, the arithmetic is able to achieve no accuracy loss on a retrained *LeNet5* model. Quantized *CifarNet* achieves no test accuracy loss at a bit-width of 8 after retraining. It can be concluded that a larger mantissa width is beneficial for both pre-retrain and post-retrain test accuracies.

Exponent bit width	4-bit	3-bit	2-bit	1-bit	0-bit
LeNet5, 32-bit floating point accuracy: 99.36%					
Before Retrain	98.41%	98.41%	98.40%	97.91%	9.79%
After Retrain	99.36%	99.36%	99.36%	99.36%	11.35%

Table 6.2: *Customized floating-point* quantization summary for *LeNet5*, with 1-bit sign, 3-bit mantissa and various exponent widths.

It is also interesting to see the effect of changing the exponent width. Table 6.2 shows how varying the exponent width can affect test accuracies. The exponent bit-width has a locally optimal value of 1 bit when the mantissa width is set to 3. Taking the sign bit into account, the arithmetic would utilize 5 bits for recovering to the original accuracy. Using *Customized floating-point*, the quantizatized $LeNet5$ is able to achieve the original test acccuracy using only 5 bits. In addition, unlike the *Dynamic fixed-point* method used in Section 5.3, this method is able to span all numerical ranges and thus the test accuracy is able to recover fully.

6.2 Re-centralized Quantization

All previous quantization methods focused on an unpruned $LeNet5$ model, in this section, I would like to propose a new quantization method specifically designed for quantizing pruned neural networks. I first show the motivation of this proposed method by comparing to the weights distribution of a pruned model and an unpruned model. I then describe the mechanism of the proposed qunatization method and show the quantization results.

Motivation

Previously, all quantization methods focused on unpruned networks and proved good compression rates. The high precision region of these applied arithmetics focuses on the center of the weights distribution.

Given an example in Figure 6.2, the chosen arithmetic has the ability to represent numbers more precisely in the region where most weights exist. The color coded bar is deepest at values near zero, and the histogram shows its peak at zero as well. This partially explains why quantizations with dynamic ranges (*Customized floating-point* and *Dynamic fixed-point*) work well, because now greater precisions can be explored near zero. However, for pruned models, the advantage of using dynamic ranges disappears.

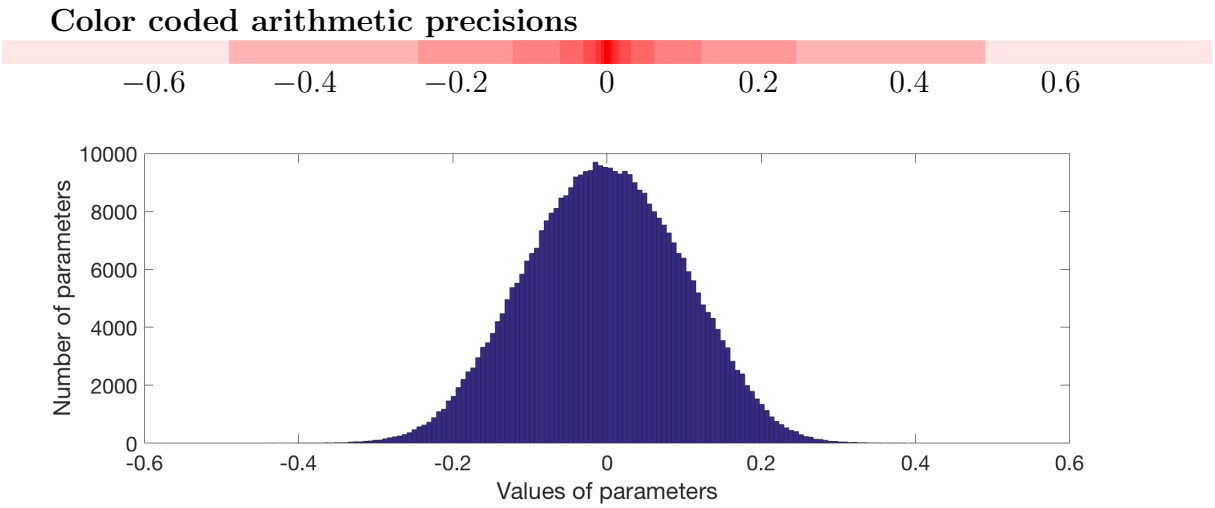

Figure 6.2: Parameter distribution of the first fully connected layer in *LeNet5*, and the color coded arithmetic precisions using *Customized floating-point quantization* with 1-bit sign, 1-bit exponent and 3-bit fraction. A deeper red color corresponds to a greater arithmetic precision, the quantization is non-linear since the representations is focused on a particular range.

As shown in Figure 6.3, the weights distribution has a binomial shape, if the same arithmetic used before applies on this weights distribution, the middle empty region will have the richest number representations. The waste in number representation suggests that a suitable arithmetic for pruned models should have the ability to represent numbers that are away from zeros in high precisions. The color coded arithmetic precision bar suggests the preferred precision intensities. The arithmetic should have high precision on two center points that are away from zero, and have low precision representation for weights that are near zero because these weights have been pruned away already.

Method Description

The methods introduced in this section is called *Re-centralization*. The shape of weights distribution of a pruned model is binomial, however, its central

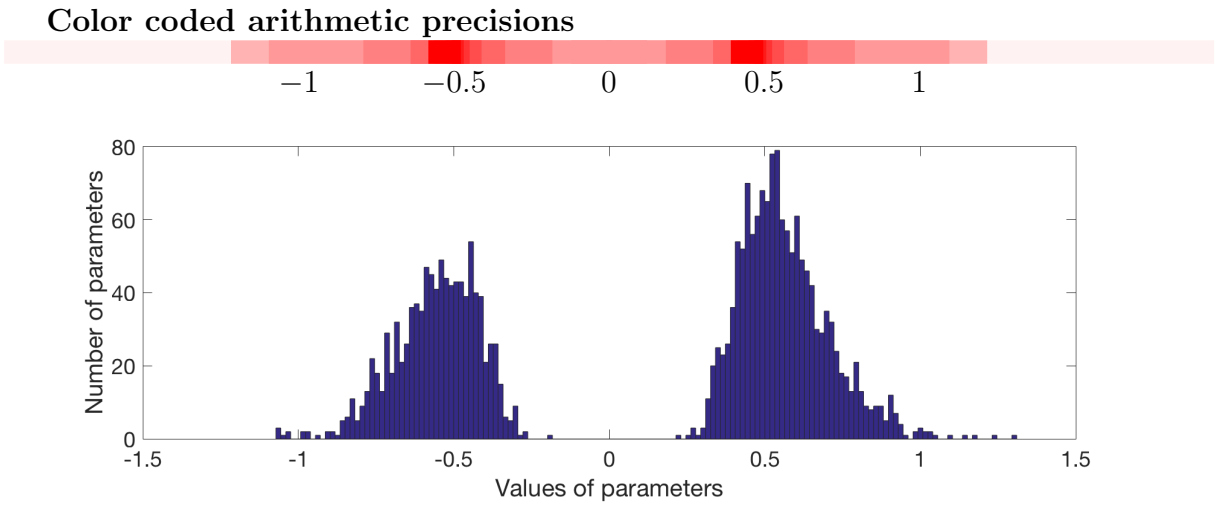

Figure 6.3: Parameter distribution of the first fully connected layer in *LeNet5*, and the color coded arithmetic precisions using *Centralized customized floating-point quantization* with 1-bit sign, 1-bit centre, 1-bit exponent and 3-bit fraction. A deeper red color corresponds to a greater arithmetic precision, the quantization is non-linear since the representations is focused on particular ranges.

values of two peaks can be different for every layer. In this method, the weights parameters are inspected before quantization occurs and two central values are determined layer-wise. Two central values correspond to the two peaks of the binomial distribution and are stored globally for parameters in the same layer. For each layer, two central values are recorded and I use 1 bit (C bit) to help a single parameter to determine which central value it is associated with. This re-centralization technique is applied on both *Dynamic fixed-point* and *Customized floating-point*.

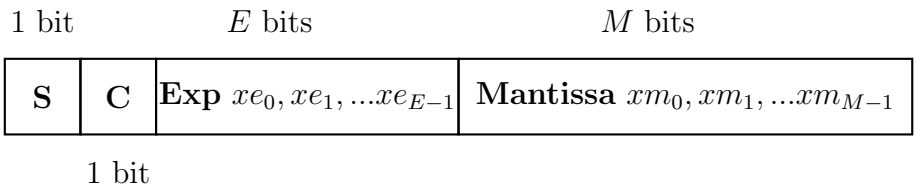

Figure 6.4: Number representation system for *Re-centralized customized floating-point quantization* with 1-bit sign, 1-bit central, E-bit exponent and M-bit mantissa.

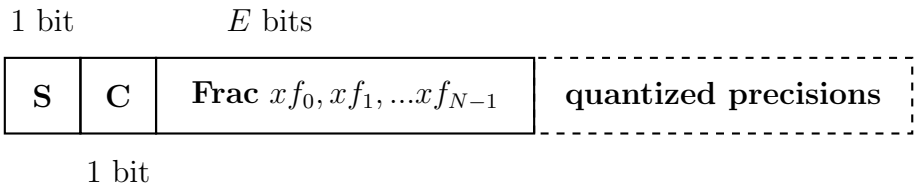

Figure 6.5: Number representation system for *Re-centralized dynmaic fixed-point quantization* with 1-bit sign, 1-bit central and N-bit fraction.

A *Re-centralized customized floating-point* number representation system is displayed in Figure 6.4, the extra bit C is also shown to determine which central value this parameter belongs to. Similarly, a *Re-centralized dynmaic fixed-point* arithmetic is shown in Figure 6.5. To convert from this *Re-centralized customized floating-point* arithmetic to a normal decimal value, consider two layer-wise central values to be C_{pos} and C_{neg}. The bit stored to identify the central value is C, we have:

$$D = (-1)^S 2^{-\sum_{i=0}^{E-1} 2^{-i} xe_i} (\sum_{i=0}^{M-1} 2^{-i} xm_i) + (CC_{pos} + (1-C)C_{neg}) \tag{6.2}$$

Similarly, *Re-centralized customized floating-point* arithmetic can be converted to the decimal representation using the equation below:

$$D = (-1)^S 2^{-DR} (\sum_{i=0}^{N-1} 2^{-i} xf_i) + (CC_{pos} + (1-C)C_{neg}) \tag{6.3}$$

The re-centralization method achieves better compression results by quantizing in a non-linear manner, which is similar to *Weights sharing*. As illustrated in *Deep Compression* [4], *Weights sharing* clusters weights in a given layer and thus achieves nonlinear quantization.

Figure 6.6 shows how *Weights sharing* changes quantization to a non-linear form, the quantized intensities are similar to Figure 6.3. In later sections, a complete comparision between the two compression pipelines will demonstrate that re-centralization works as well as *Weights sharing*. Besides, re-

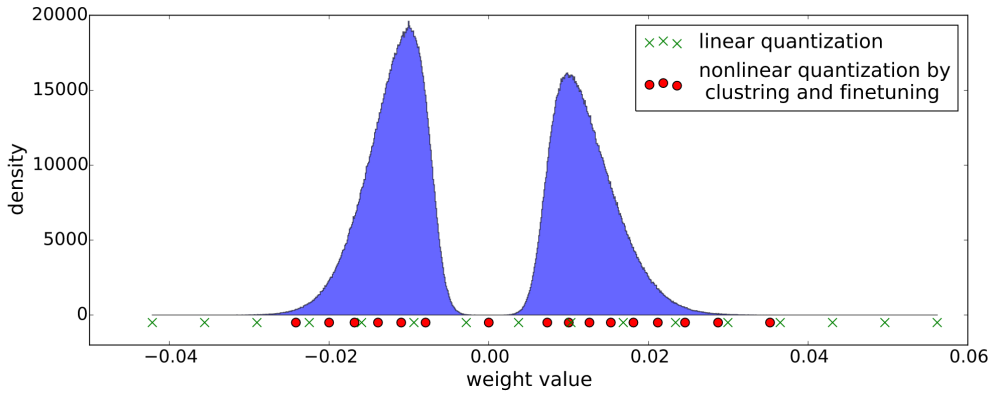

Figure 6.6: Distribution of weights (blue) and distribution of codebook before (green) and after *Weights sharing*(red).

centralization avoids clustering of weights and searching for weights based on their indexes. As mentioend previously, searching weights based on indexes might cause extra data movements and thus reduces energy efficiency.

Results

To demonstrate the performance gain from re-centralization, pruned models of both *LeNet5* and *CifarNet* are considered. I show a comparison between *Customized floating-point quantization*, *Centralized dynamic fixed-point* and *Centralized customized floating-point* to observe the effects of re-centralizations. The results of quantization on a pruned *LeNet5* are shown in Table 6.3, and Table 6.4 shows the quantization results on pruned *CifarNet*.

As shown in Table 6.3, *Customized floating-point* arithmetic has a similar post-retrain accuracy to both *Centralized customized floating-point* and *Centralized dynamic fixed-point* on *LeNet5*. However, the pre-retrain results of using *Centralized dynamic fixed-point* is actually the best one in these three arithmetics. The limited mantissa width causes *Centralized customized floating-point* to have a bad pre-train accuracy. The pre-train accuracy of this simple network is tightly linked to the number of mantissa (or fractional) bits available. The underlying issue is that *LeNet5* is too simple, it seems

Bit width	32-bit	16-bit	8-bit	6-bit	4-bit
LeNet5, *Customized floating-point*					
Before Retrain	99.36%	99.36%	99.31%	98.40%	73.94%
After Retrain	99.36%	99.36%	99.36%	99.36%	99.36%
LeNet5, *Centralized dynamic fixed-point*					
Before Retrain	99.04%	99.04%	99.06%	98.93%	91.86%
After Retrain	99.36%	99.36%	99.36%	99.36%	99.36%
LeNet5, *Centralized customized floating-point*					
Before Retrain	98.73%	98.72%	98.67%	98.04%	21.81%
After Retrain	99.36%	99.36%	99.36%	99.36%	99.36%

Table 6.3: Quantization summary on a pruned *LeNet5* model. *Customized floating-point* has 1-bit sign, 1-bit exponent and the rest are mantissa bits. *Centralized customized floating-point* has 1-bit sign, 1-bit central, 1-bit exponent and the rest are mantissa bits. *Centralized dynamic fixed-point* has 1-bit sign, 1-bit central and the rest are fraction bits.

like a large precision near the central values would not introduce critical improvements when no retraining takes place. Another problem with this simple *LeNet5* network is that all arithmetics can achieve the same post-train accuracies, and lower precisions cannot be explored because the sign bit and exponent bits set a minimum total bit width.

Table 6.4 shows the quantization resutls when various quantization methods are applied on the pruned *CifarNet*, *Centralized customized floating-point* and *Centralized dynamic fixed-point* shows a significant improvement in pre-train accuracy compared to *Customized floating-point*. Conceptually, centralized arithmetics re-focus the arithmetic centers to fit the binomial weights distributions, and this should increase the precisions near central values. Using *Centralized customized floating-point* on *CifarNet*, it achieves the reqruied accuracy at a precision of only 6 bits.

Bit width	32-bit	16-bit	8-bit	6-bit
CifarNet, *Customized floating-point*				
Before Retrain	34.30%	29.19%	30.58%	20.56%
After Retrain	82.02%	82.01%	81.85%	81.12%
CifarNet, *Centralized dynamic fixed-point*				
Before Retrain	81.18%	81.17%	80.06%	73.23%
After Retrain	82.02%	82.06%	82.05%	82.10%
CifarNet, *Centralized customized floating-point*				
Before Retrain	59.65%	59.61%	58.23%	44.94%
After Retrain	82.01%	82.11%	82.00%	81.79%

Table 6.4: Quantization summary on a pruned *CifarNet* model. *Customized floating-point* has 1-bit sign, 2-bit exponent and the rest are mantissa bits. *Centralized customized floating-point* has 1-bit sign, 1-bit central, 2-bit exponent and the rest are mantissa bits. *Centralized dynamic fixed-point* has 1-bit sign, 1-bit central and the rest are fraction bits.

6.3 Summary of Quantization Methods

To summarize, I would like to compare the implemented quantization methods on the targeting networks.

Consider unpruned *LeNet5* and *CifarNet* as targeting networks, Table 6.5 shows the bit-widths of various networks after quantizations. As mentioned previously, *Dynamic fixed-point* (*DFP*) restricts the range of weights and thus some weights can never have a chance to recover. As shown in the table, the error rate of using *Dynamic fixed-point* is slightly higher on *LeNet5* due to this effect. In contrast, *CifarNet* shows comparable quantization results for both *Dynamic fixed-point* and *Customized floating-point* (*CFP*). *Gysel et al.* obtained their best quantization results using *Dynamic fixed-point* on the unpruned *LeNet5* and *CifarNet*, I added their results to Table 6.5.

For quantizing unpruned models, *Customized floating-point* shows best results on the two selected networks: it showed a good compression rates on both networks without any loss of test accuracies.

Quantization on pruned networks works differently from the unpruned ones.

Model	*Fixed-point*	*DFP*	*CFP*	(*Gysel*)
LeNet5	16 bits(0.64%)	4 bits(0.69%)	4 bits(0.64%)	4 bits(0.90%)
CifarNet	32 bits(18%)	8 bits(18%)	8 bits(18%)	8 bits(18.3%)

Table 6.5: Quantization summary (bit-width, error rate) on unpruned *LeNet5* and *CifarNet*. *DFP* is *Dynamic fixed-point*, *CFP* is *Customized floating-point* and (*Gysel*) is *Gysel et al.*'s implemenetation of *Dynamic fixed-point* [5].

As illustrated before, pruned network have central values that are non-zeros. Re-centralization significantly helps number representation systems to focus on the region with a large number of parameters using only one extra central bit. Two re-centralization methods, named *Centralized customized floating-point* (*CCFP*) and *Centralized dynamic fixed-point* (*CDFP*) respectively, are applied on the pruned networks. Table 6.6 shows the quantization results, both arithmetics achieve same quantization results on the pruned *LeNet5* model, but *Centered dynamic fixed-point* shows a better performance on the pruned *CifarNet* model by achieving zero accuracy loss using only 6 bits.

Model	*CFP*	*CCFP*	*CDFP*
LeNet5	4 bits(0.64%)	4 bits(0.64%)	4 bits(0.64%)
CifarNet	8 bits(18.2%)	8 bits(18%)	6 bits(18%)

Table 6.6: Quantization summary (bit-width, error rate) on pruned *LeNet5* and *CifarNet*. *CFP* is *Customized floating-point* *CCFP* is *Centralized customized floating-point* and *CDFP* is *Centralized dynmaic fixed-point*.

Chapter 7

Evaluation

7.1 Compression Pipeline

Han et al. proposed a three-stage compression pipeline called *Deep Compression*. This work is most comparable to my compression pipeline. Other research works almost focus only on whether pruning or quantization in an isolated manner. In this section, I would like to compare my compression pipeline to *Deep Compression*

As shown in Figure 7.1, *Han et al.* focused on reducing the number of parameters first using *Deterministic pruning* iteratively. This iterative pruning process is combined with retraining to bring back the lost accuracies. They then utilized both *Fixed-point quantization* and *Weights sharing* to minimize the number representation of each individual parameter. Weights are quantized to limited precisions and encoded using a codebook to reduce the amount of representations. Finally, they tried to compress the number representations further using an encoding scheme [4].

Deep Compression is proven successful on many popular neural networks. To give a fair comparison between my compression pipeline and *Deep Compression*, the encoding stage is not considered for two reasons. First, encoding schemes normally encode symbols by their occurrences, such an encoding

scheme can be attached to the end of any compression pipelines. Second, the use of encoding indicates a need for encoder and decoder in hardware, this might affect energy efficiency of the hardware platform.

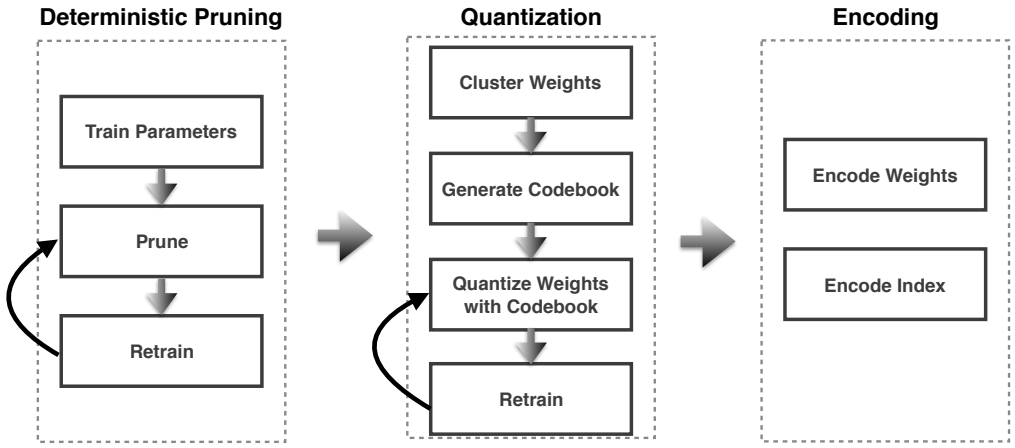

Figure 7.1: Overview of *Deep Compression* [4].

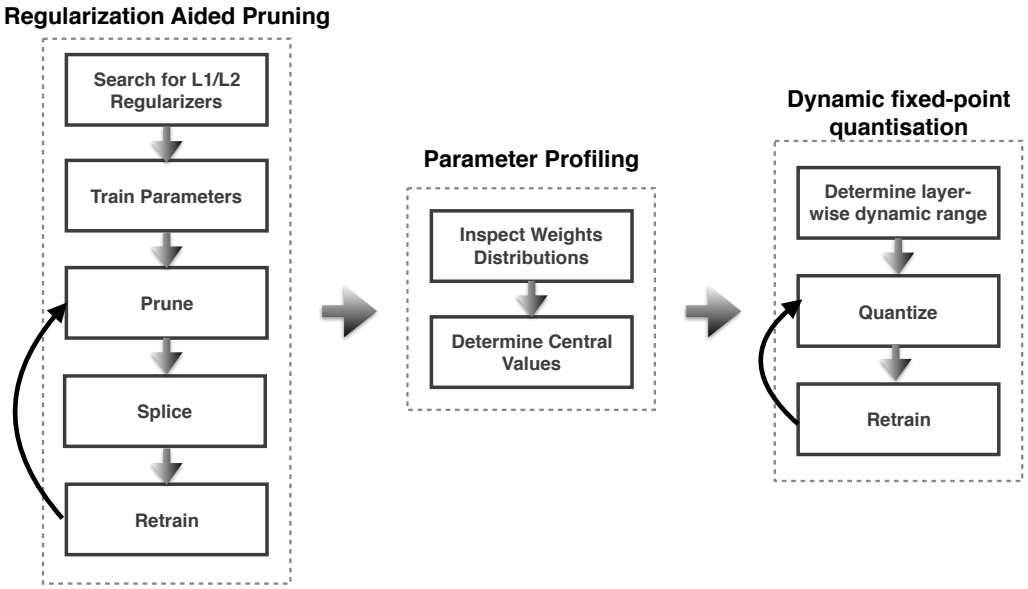

Figure 7.2: Overview of the proposed compression pipeline.

In my implementation, the focus stays on pruning and quantization, an overview of the complete compression pipeline is shown in Figure 7.2. *Regu-*

larization aided pruning is performed to reduce the number of parameters in a neural network. The best hyperparameter combinations for the regularizer come from an exhaustive search. The rest of the pruning process is identical to *Dynamic network surgery*, where pruned weights have a chance to recover. The second phase is to profile the parameters, this profiling serves quantization since some important information is extracted from the pre-quantized model and the central values are determined. The quantization method used is *Centralized dynamic fixed-point quantization*. The dynamic ranges are stored in a layer-wise fashion, each quantization is combined with retraining to recover the test accuracies.

7.2 Evaluation of Performance

To fully understand the performance difference between my proposed compression pipeline and *Deep Compression*, Table 7.1 summarizes the performances of the two compression pipelines on *LeNet5*.

Model	Layer	Params	%(*Han*, P)	%(*Han*, P+Q)	%P	%P+Q
LeNet5	cov1	0.5K	66%	78.5%	78.9%	12.2%
	cov2	25K	12%	6.0%	7.5%	1.2%
	fc1	400K	8%	2.7%	0.3%	0.047%
	fc2	5K	19%	6.9%	67.1%	10.5%
	total (CR)	431K	8%(12x)	3.05%(33x)	1.6%(63x)	0.25%(403x)
	ER	-	0.8%	0.8%	0.64%	0.64%

Table 7.1: *LeNet5* compression summary, Pruning and Quantization, CR is the compression rate, ER is the error rate. P represnets pruning and Q reporesents quantization.

As shown in Table 7.1, putting a *LeNet5* into the proposed compression pipeline, the neural network obtains a compression rate of 403x. This compression rate shows an 12x increase compared to *Deep Compression*. In terms of pruning, *Regularization aided pruning* provides a better result, showing an increase of 5.25x compared to *Determnisitc pruning*. The proposed compression pipeline uses *Centralized dynamic fixed-point arithmetic* and quantized

the entire model to 5 bits.

It is also important to note that the use of *Weights sharing* in *Deep Compression* gives significant energy overhead. Each weight now is encoded using a codebook. When neural network inference occurs, each weight fetches the correct value using the stored index as an address. This extra weights fetching process is the cost of using *Weights sharing*. *Yang et al.* suggested one of the major energy consumption in network inference is data movement [15]. Since *Weights sharing* added this redundant data movement, its energy consumption should be evaluated more carefully for understanding the trade-off between energy savings and compression rates. In contrast, using *Centralized dynamic fixed-point arithmetic*, the arithmetic operators can be designed specifically for this kind of arithmetic and thus avoids the weights fetching stage.

Model	Layer	Params	%P	%P+Q
CifarNet	cov1	4.8K	55%	10.3%
	cov2	102.4K	12%	2.3%
	fc1	885K	4%	0.75%
	fc2	74K	24%	4.5%
	fc3	2K	60%	11.2%
	total(CR)	1068K	6.5%(15.4x)	1.2%(82.2x)
	ER	-	18%	18%

Table 7.2: *CifarNet* compression summary, Pruning and Quantization, CR is the compression rate, ER is the error rate. P represnets pruning and Q reporesents quantization.

As stated in Table 7.2, pruning offers a compression rate of 15.4x. Combinging pruning with quantization, the proposed compression pipeline offers a compression rate of 82.2x on *CifarNet* without any loss of test accuracy.

7.3 Evaluation of Compression Techniques

In my compression pipeline, two main phases are pruning and quantization. The experimental results of various pruning techniques are summarized in

Section 4.3. Non-deterministic pruning methods are generally better than deterministic pruning. The best performance pruning strategy is *Regularization aided pruning*, however, it might take a significantly larger amount of time depending on how experienced the designer is in finding the correct regularization parameters. In contrast, *Dynamic network surgery* also proves to have good compression rates and avoids the efforts of searching for the regularization hyperparameters.

In terms of quantization, the suitable methods for pruned and unpruned models are different. For unpruned models, *Customized floating-point* proves its performance is comparable to the popular *Dynamic fixed-point* arithmetic, moreover, it does not restrict any dynamic ranges in the number representation system. Consider pruned models, an idea of re-centralizing the arithmetic representations is suggested, and *Centralized dynamic fixed-point* arithmetic achieved the best performance.

Chapter 8

Summary and Conclusions

8.1 Conclusion

In this project, I have investigated a number of pruning and quantization methods and built a complete compression pipeline based on the methods that have the best performances. A number of existing pruning and quantization methods have been explored in this project, and they are listed below:

1. Deterministic pruning
2. Dynamic network surgery
3. Fixed-point quantization
4. Dynamic fixed-point quantization

Inspired by these existing compression techniques, some novel pruning and quantization methods are also proposed:

1. Gradient profiling pruning
2. Regularization aided pruning
3. Customized floating-point quantization
4. Re-centralized quantization

The large range of exploration of compression techniques provided some important empirical results. For pruning, as summarized previously in Section 3.3 and Section 4.3, it can be concluded that pruning only weights and pruning in a non-deterministic manner give significantly better compression results. Regularizations also help improving the compression rates. The proposed pruning strategy, *Regularization aided pruning*, becomes the best performance pruning method. In terms of quantization, the results suggest *Customized floating-point quantization* achieves best quantization results on unpruned models due to its ability of tracking near-zero numbers to high precisions. Later, for pruned neural networks, re-centralization of arithmetics gives significant improvement on pruned model. The practical results proved that the preferred arithmetics are different for dense and sparse network models because of the difference in weights distributions.

At the end, I selected *Regularization aided pruning* and *Centralized dynamic fixed-point quantization* for building a complete compression pipeline. The proposed compression pipeline achieves a better compression results than the existing compression pipeline (*Deep Compression*). On the *LeNet5* model, the proposed compression pipeline (403x) shows a significant advance in compression rate compared to *Deep Compression* (33x). On the *CifarNet* model, the proposed compression pipeline achieved a compression rate of 82.2x.

8.2 Future Works

Possible future works can be broken down into two parts: pruning and quantization. One possible future work is to focus on reducing the retraining time in pruning. Most pruning methods require at least one complete retraining after the pruning process [46, 54]. Shorten the retraining time can significantly reduce the computation time, but it might cause a bad compression result. *Gradient profiling pruning* shows promising results with limited retraining resources. It obtains information of the importance of weights during retraining and uses this information to prune weights in a more selec-

tive manner. However, its performance should be further verified on other networks. *Regularization aided pruning* is proven useful in achieving high compression rates. The amount of time spent on choosing feasible hyperparameters for regularizers are entirely empirical at this stage. One possible future work is to automate this process. It has been empirically observed that the feasible regularization loss (loss caused by regularizers) is around ten percents of the value of the cost function. If such a numerical ratio can be confirmed on a large range of networks, it is easy to automate the hyperparameter definitions by inspecting this numerical ratio at run-time. Another potential future work is to explore more efficient regularizers. Regularizers help pruning because they encourage sparsity during the retraining phase. It is therefore logical to consider using regularizers that are better at encouraging zeros for achieving a better compression. *Shakeout* is a new regularizer proposed by *Kang et al.* [55]. *Kang et al.* proved *Shakeout* encourages more sparsity than traditional l_1 and l_2 norms [55]. It is possible to use *Shakeout* as a regularizer for *Regularization adided pruning* in future developments.

With respect to quantization, I have demonstrated that *Customized floating-point* can be more efficient than the popular *Dynamic fixed-point* method when the neural network model is dense. It is noted that *Dynamic fixed-point* arithmetic fails to recover test accuracies if the trained network is sensitive to reduced dynamic ranges. A theoretical proof and more throughout experimentations should carry on in this direction to prove whether this is a general phenomenon. Some novel arithmetics, such as logarithmic arithmetics, are catching attentions recently [56] because they can compute multiplications in a power-efficient manner. Part of the future works is to propose energy consumption models for the proposed arithmetic operators in this project. These energy estimations could be helpful for exploring possible hardware accelerator architectures.

In this project, pruning and quantization are treated as individual stages and retraining takes place in each of these stages. One interesting future work is to combine pruning, quantization and retraining all in one stage. Previously, a number of research works have tried to combine pruning with retraining

into one process and thus eliminates the need of pruning iteratively [26, 57]. The strategy is to add the number of zeros of weights masks into the cost function so that the network encourages more values to be pruned away. It is interesting to see whether there is a chance to fit quantization into this method, for instance, the bit-width might be considered as a metric in the cost function and a lower bit-width is preferred.

Another important future work is to apply the proposed compression pipeline to a larger number of neural networks. Networks, like *AlexNet* [2], *VGGNet* [9] and *InceptionNet* [58], should be tested on the proposed compression pipeline.

Bibliography

[1] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[2] 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.

[3] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. *CoRR*, abs/1608.04493, 2016.

[4] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *CoRR*, abs/1510.00149, 2015.

[5] Philipp Gysel. Ristretto: Hardware-oriented approximation of convolutional neural networks. *CoRR*, abs/1605.06402, 2016.

[6] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[7] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[8] Adam Coates, Brody Huval, Tao Wang, David J. Wu, Bryan Catanzaro, and Andrew Y. Ng. Deep learning with COTS HPC systems. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1337–1345, 2013.

[9] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[10] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[11] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[14] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

[15] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing energy-efficient convolutional neural networks using energy-aware pruning. *CoRR*, abs/1611.05128, 2016.

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

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

[18] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.

[19] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with compressed lstm on fpga. *arXiv preprint arXiv:1612.00694*, 2016.

[20] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[21] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. Can fpgas beat gpus in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, pages 5–14, New York, NY, USA, 2017. ACM.

[22] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[24] Dunja Mladenić, Janez Brank, Marko Grobelnik, and Natasa Milic-Frayling. Feature selection using linear classifier weights: interaction with classification models. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 234–241. ACM, 2004.

[25] Robert Hecht-Nielsen et al. Theory of the backpropagation neural network. *Neural Networks*, 1(Supplement-1):445–448, 1988.

[26] Babak Hassibi, David G Stork, et al. Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, pages 164–164, 1993.

[27] Suraj Srinivas and R Venkatesh Babu. Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*, 2015.

[28] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J Dally. Exploring the regularity of sparse structure in convolutional neural networks. *arXiv preprint arXiv:1705.08922*, 2017.

[29] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *CoRR*, abs/1608.08710, 2016.

[30] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *CoRR*, abs/1608.03665, 2016.

[31] Vadim Lebedev and Victor S. Lempitsky. Fast convnets using group-wise brain damage. *CoRR*, abs/1506.02515, 2015.

[32] Wenlin Chen, James Wilson, Stephen Tyree, Kilian Weinberger, and Yixin Chen. Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294, 2015.

[33] Bert Moons and Marian Verhelst. An energy-efficient precision-scalable convnet processor in a 40-nm cmos. *IEEE Journal of Solid-State Circuits*, 2016.

[34] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016.

[35] 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.

[36] Fengfu Li, Bo Zhang, and Bin Liu. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016.

[37] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and

Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

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

[39] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. The CIFAR-10 dataset, 2014.

[40] Patrick J Grother. NIST special database 19 handprinted forms and characters database. *National Institute of Standards and Technology*, 1995.

[41] Yann LeCun et al. LeNet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 2015.

[42] Antonio Torralba, Rob Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 30(11):1958–1970, 2008.

[43] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[44] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[45] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[46] Georg Thimm and Emile Fiesler. Pruning of neural networks. Technical report, IDIAP, 1997.

[47] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint arXiv:1611.06440*, 2016.

[48] Feiping Nie, Heng Huang, Xiao Cai, and Chris H Ding. Efficient and robust feature selection via joint l2, 1-norms minimization. In *Advances in neural information processing systems*, pages 1813–1821, 2010.

[49] Naveen Mellempudi, Abhisek Kundu, Dipankar Das, Dheevatsa Mudigere, and Bharat Kaul. Mixed low-precision deep learning inference using dynamic fixed point. *arXiv preprint arXiv:1701.08978*, 2017.

[50] Tapas Kanungo, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*, 24(7):881–892, 2002.

[51] Miloš D Ercegovac and Tomas Lang. *Digital arithmetic*. Elsevier, 2004.

[52] Darrell Williamson. Dynamically scaled fixed point arithmetic. In *Communications, Computers and Signal Processing, 1991., IEEE Pacific Rim Conference on*, pages 315–318. IEEE, 1991.

[53] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[54] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*, 2016.

[55] Guoliang Kang, Jun Li, and Dacheng Tao. Shakeout: a new regularized deep neural network training scheme. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 1751–1757. AAAI Press, 2016.

[56] Hokchhay Tann, Soheil Hashemi, Iris Bahar, and Sherief Reda. Hardware-software codesign of accurate, multiplier-free deep neural networks. *arXiv preprint arXiv:1705.04288*, 2017.

[57] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning convolutional neural networks for resource efficient transfer learning. *CoRR*, abs/1611.06440, 2016.

[58] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.