Karlsruher Institut für Technologie

Institut für Telematik
Pervasive Computing Systems / TECO
Leiter: Prof. Dr.-Ing. Michael Beigl

Training Quantized Neural Networks with ADMM Approach

Master's Thesis

by

Xue Ma

Department of Informatics

Responsible Supervisor:	Prof. Dr. Michael Beigl
Supervising Staff:	Haibin Zhao
Project Period:	01.02.2023 - 01.08.2023

Contents

List of Figures

List of Tables

Abstract

Deep learning models have achieved remarkable success in various learning tasks, but their high computational costs pose challenges for deployment in resource-limited scenarios. In this paper, we focus on addressing this issue by quantizing deep learning models, where network weights are represented by a smaller number of bits. We formulate this problem as a discrete optimization problem and draw inspiration from the Alternating Direction Method of Multipliers to optimize the parameters in a neural network. We introduce two approaches to quantize neural networks using the Alternating Direction Method of Multipliers algorithm. The first approach is a gradient-free optimization method for training the quantized neural network. It avoids many problems of gradient descent, such as saturation effects and saddle points. In contrast, the second approach is a gradient-based method to quantize the neural network layerwisely using Alternating Direction Method of Multipliers based on a pre-trained neural network. After each layer is quantized, the parameters of non-quantized layers are updated to compensate for the loss of accuracy.

Notation

In this chapter we show some important notations and their descriptions in the following paper.

Notations	Descriptions
a_l	The output of the activation function for the l-th layer
a_l^k	The updated value of a_l in the k-th iteration
α_l	The scaling factor that separated from the dequantized weight matrix G_l
β	The penalty factor to control weight of constraints $z_l = W_l a_{l-1}$
C	The discrete set filled with integer values
C_l	The discrete set in which dequantized weight matrix are constrained
$F(\cdot)$	The objective function for layerwise quantization
$h_l(\cdot)$	The activation function for l-th layer
G_l	The dequantized weight matrix for l-th layer
$J_a(\cdot)$	The objective function of the sub-problem a_l
$J_w(\cdot)$	The objective function of the sub-problem W_l
$J_z(\cdot)$	The objective function of the sub-problem z_l
κ	The scaling factor in backtracking algorithm for updating W_l
k	k-th iteration in complete parameters updating process
$\mathcal{L}(\cdot)$	The Loss function of neural network about z_L
η	The coefficient of quadratic term for updating z_L
Q_l	The quantized weight matrix for l-th layer
q_l^{ij}	The element of Q_l in the i-th row and j-th column
ρ	The penalty factor in layerwise quantization
R	The loss function about weight
γ	The penalty factor to control weight of constraints $a_l = h_l(z_l)$
$S(\cdot)$	The softmax function
τ	The scaling factor in the backtracking algorithm for updating a_l
μ	The penalty factor to control weight of constraints $W_l = G_l$ in Chapter 4
υ_l	The coefficient of the quadratic term of $\psi(\cdot)$
W_l	The weight matrix for the l-th layer
W_l^k	The updated value of W_l in the k-th iteration
w_l^{ij}	The element of W_l in the i-th row and j-th column
$\overline{W}$	The already quantized weight matrix
y	The desired output of the neural network

Table 1: Important notations and descriptions

Notations	Descriptions
z_l	The output of linear operator for the l-th layer
$\hat{z_L}$	The start point of each FISTA iteration
z^{ij}	The element of the matrix z_l in the i-th row and j-th column
θ_l	The coefficient of the quadratic term of $\phi(\cdot)$
λ	Lagrange multiplier
$\varphi(\cdot)$	The approximated quadratic function for updating z_l
$\psi(\cdot)$	The approximated quadratic function of $J_w(\cdot)$
$\phi(\cdot)$	The approximated quadratic function of $J_a(\cdot)$

Table 2: Important notations and descriptions

1. Introduction

In recent years, neural networks have achieved remarkable success in many emerging fields, such as natural language processing [12] and computer vision [11]. This success can be attributed to increasing larger model sizes and the corresponding development of computing resources. However, large models impose a heavy storage footprint due to the enormous amounts of network parameters. For example, the 16-layer VGG network has a storage footprint of 528 megabytes. This high cost can be a significant barrier to deploying deep neural networks on edge devices, whose memory or computing resources are limited. As a result, there is a growing interest in compressing deep models to reduce their computational cost and memory footprint.

Neural network quantization [21] is widely used to reduce the precision of weights and activations in a deep neural network to lower memory and computational costs. By representing weights and activations with fewer bits, model quantization can significantly reduce the model size with little loss of accuracy. This technique has been shown to be highly effective in achieving both memory and computational efficiency in deep neural networks, making it a popular choice for deploying artificial intelligence applications in resource-constrained environments [31, 19, 5]. However, after model quantization, the weights and activation values are often discretized into fixed numbers of bits, that are no longer continuously differentiable. Since the training of neural networks is based on the gradient descent algorithm, calculating the gradients of quantized network during backpropagation becomes a challenge.

To address this problem, we focus on training quantized neural networks using the gradient-free optimization method, Alternating Direction Method of Multiplier, also known as ADMM. By introducing the augmented Lagrangian, we can decompose the large neural network into smaller optimization problems which are easier to be solved with global optimum. It obviates multiple drawbacks of gradient-based optimization methods, such as saddle points and vanishing gradients.

We begin in Chapter 2 by providing background about neural networks, convolutional neural networks, and some optimization algorithms, such as ADMM and fast iterative shrinkage-thresholding algorithm. Chapter 3 proposes the mathematical notation context and the algorithm for training the quantized neural network with

ADMM approach. Chapter 4 describes the optimization approach of parameters in the quantized neural network.

Moreover, we also propose a method, layerwise quantization for neural networks using ADMM. Chapter 5 introduces the algorithm to quantize the neural network layer by layer.

2. Background & Related Work

2.1 Neural network

A neural network is a model that contains a series of operations that aims to identify underlying relationships in a given dataset by assembling the functionality of the human brain. However, unlike the human brain, the artificial neural network is is simplified to a layered structure. A typical neural network is composed of multiple

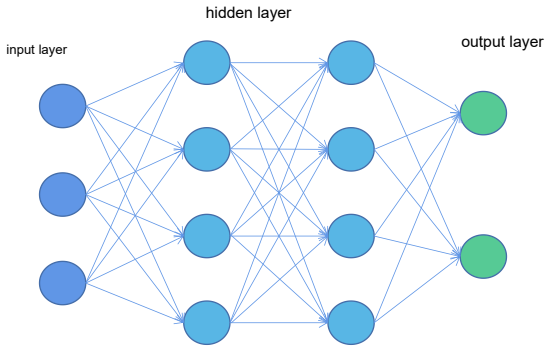

Figure 2.1: Example of a 3-layer neural network.

layers, including an input layer, an output layer, and some hidden layers. Each layer consists of multiple neurons. The neurons in a layer receive output from the neurons in the previous layer, perform a linear transformation followed by a non-linear activation function, and pass its output to the neurons in the next layer. This kind of neural network is also known as Multilayer Perceptron (MLP) [26]. Figure 2.1 shows the structure of a 3-layer neural network. Now, we suppose that the neural network has L layers. We denote the linear operator of the l-th layer as W_l and the non-linear activation function of the l-th layer as $h_l(\cdot)$. W_l is also known as the weight for l-th layer of the neural network. Given a vector of input activations a_{l-1}, a single layer computes and outputs the non-linear function $a_l = h_l(W_l a_{l-1})$. A network is formed by layering these units together in a nested fashion to compute a composite function [29]. For instance, in the 3-layer case, this would be

$$f(W) = W_3(h_2(W_2 h_1(W_1 a_0))), \tag{2.1}$$

where $W = \{W_l, l = 1, 2, 3, \ldots, L\}$ denotes the set of weight matrices, and a_0 contains input activations for every training sample.

During training, the weights of the connections between neurons are adjusted to minimize the error between the network's output $f(W)$ and the desired output y, which is generally the label provided by the target dataset. Using the loss function $\mathcal{L}(\cdot)$, the training process can be modeled as an optimization problem:

$$\underset{W}{\text{minimize}}\, \mathcal{L}(f(W), y). \tag{2.2}$$

The loss function computes the error between the current output of the neural network and the desired output. By minimizing this error, the network can give the output that is closest to the desired output.

To solve Equation 2.2, gradient-based approaches, such as gradient descent (GD), are usually adopted. The basic idea behind gradient descent is to iteratively update the model parameters in the direction of the steepest descent of the objective function until convergence or a predetermined number of iterations. Different variants of gradient descent vary in the way that the gradient is computed and how the parameters are updated. These variants, like batch gradient descent, stochastic gradient descent, and mini-batch gradient descent, are applied to optimize Equation 2.2. But the convergence of all gradient methods suffers from the problem, such as saturation effects and saddle points. To mitigate the drawbacks in gradient-based methods, Gavin Taylor proposed a method that uses alternating direction methods and Bregman iteration to train the networks without gradient descent steps [29]. This method exhibits good scalability. Although ADMM is a powerful optimization framework that can be applied to large-scale deep learning applications, it usually converges slowly to high accuracy, even for simple examples [8]. It is often the case that ADMM becomes trapped in a modest solution and hence performs worse than stochastic gradient descent [30]. To overcome this problem, Wang et al. proposed a novel and efficient dlADMM algorithm to handle the fully-connected deep neural networks problem and improve the accuracy of the model when training neural networks based on the ADMM approach [30].

2.2 Convolutional neural network

2.2.1 Basic convolutional neural network

Multilayer Perception performs poorly when it comes to handling image problems. For high-resolution images, Multilayer Perception with fully connected layers flatten the image into a long vector, disregarding the spatial relationships between pixels. This can prevent the model from capturing local features and spatial structures, thus impacting its performance. It can also result in a high number of parameters. This increases computational and storage requirements, leading to longer training and inference time. In addition, traditional neural networks are sensitive to changes in the position of objects in the image. To overcome these problems, the convolutional neural network is proposed [22].

Convolutional neural network (CNN) is inspired by the visual system in biological organisms, particularly the working principle of neurons in the visual cortex [18].

Compared to Multilayer Perception, the advantages of CNN, such as local connectivity, parameter sharing, hierarchical feature extraction, and spatial pooling, make them particularly effective at handling grid-like structured data like images, enabling them to achieve state-of-the-art performance in various computer vision tasks.

The core idea of CNN is to extract features from input images through multiple layers of convolution and pooling operations, followed by fully connected layers.

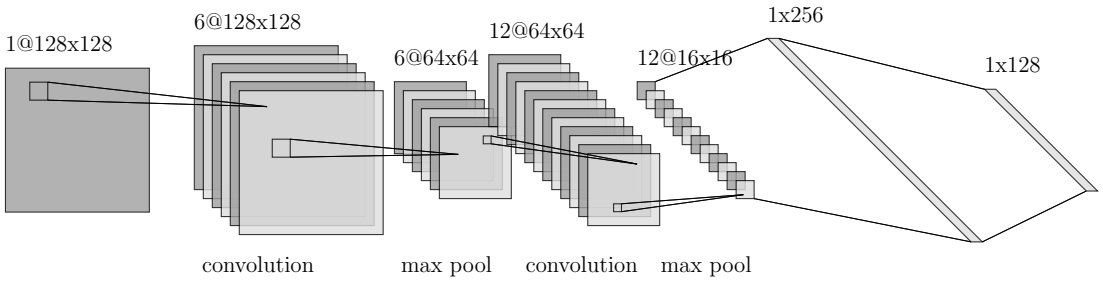

Figure 2.2: Illustration of a convolutional neural network with two convolution operators and two max pool operators, followed by 2 fully connected layers.

The convolutional layer applies a set of convolutional kernels to perform convolutional operations on the input image. The kernels scan the input image with a sliding window in a local receptive field and extract features by element-wise multiplication and summation. Convolutional operations effectively capture local features of the input image by sharing the same filter weights at different positions. Figure 2.3 shows how convolutional layer works.

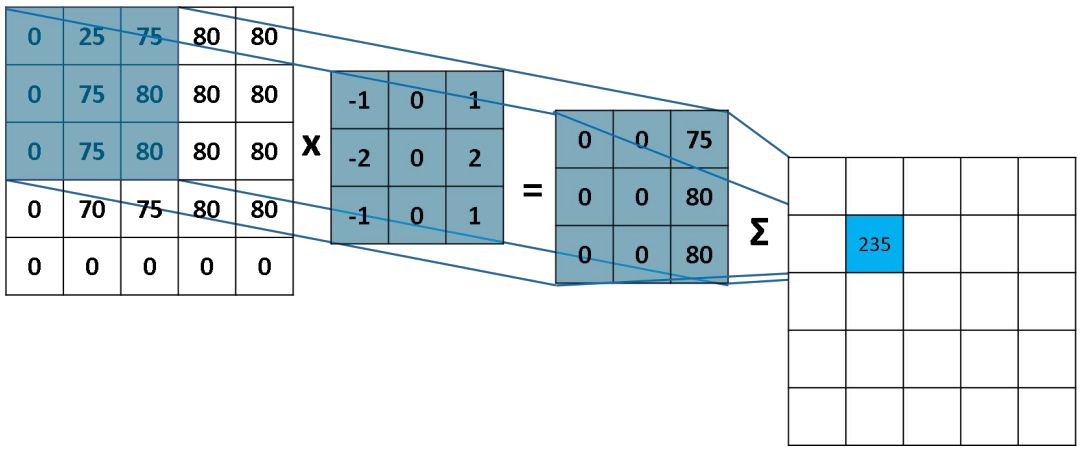

Figure 2.3: Illustration of the convolutional operator [1].

Pooling operations are used to reduce the spatial dimensions of feature maps and decrease computational complexity in subsequent layers. Common pooling operations include max pooling and average pooling, which select the maximum or average value within a local receptive field as the output. By performing pooling operations, CNNs can exhibit invariance to spatial and scale variations, improving the model's robustness. Figure 2.4 shows how max pooling layer works.

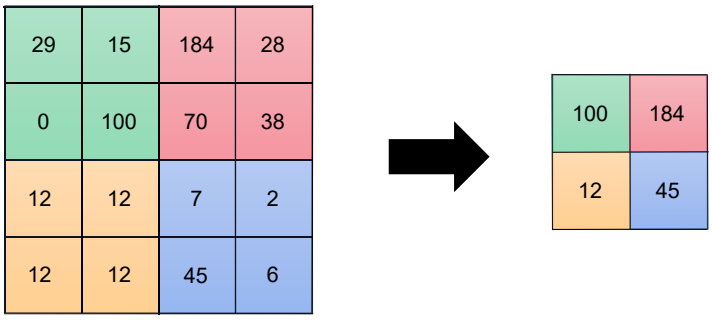

Figure 2.4: Example of max pooling operation with $stride = 2$ and $kernel = (2, 2)$.

In a CNN, multiple convolutional and pooling layers are stacked together, gradually learning higher-level feature representations. The final layer is usually a fully connected layer, which maps the high-level features to the desired classification or regression output.

2.2.2 ResNet

In traditional convolutional neural networks, information flows through the layers, and each layer learns to map the input to the output. However, as the network becomes deeper, the gradients tend to vanish during backpropagation, making the network difficult to train. Additionally, the problem of model degradation occurs, where increasing the network depth does not necessarily lead to better performance and may even lead to worse performance.

To overcome these issues, the residual network was introduced in 2015 by Kaiming He et al. to address the problems of vanishing gradients and model degradation during the training of deep networks [15]. Residual networks introduce residual connections. Residual connections create "residual blocks" (as shown in Figure 2.5) by connecting the input signal directly to the output through skip connections. This allows the network to learn residual mappings, i.e., learning to transform the input residual into the output residual. Since residual learning is easier than learning the full mapping, the network can more easily optimize the residual part without being affected by vanishing gradients and degradation problems.

In a residual network, each residual block typically consists of two or three convolutional layers and a residual connection from the input. Additionally, to reduce the feature map size while increasing the network depth, convolutional or pooling layers with a stride of 2 are used for downsampling. The entire network is composed of multiple residual blocks, where the depth and width can be adjusted based on the specific task.

The residual networks have achieved significant performance improvements in various computer vision tasks such as image classification [23], object detection [3], and image segmentation [25]. Furthermore, the concept of residual networks has been widely applied to other deep learning tasks in different domains, providing an effective solution to the challenges of training deep networks.

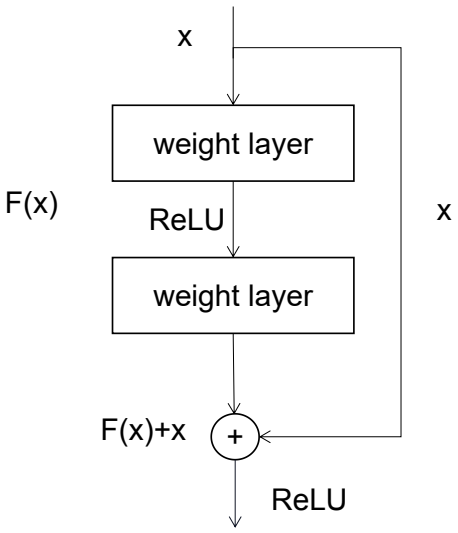

Figure 2.5: A building block of residual block.

In our experiment, we use the model ResNet20, which contains 20 convolutional layers in residual blocks. It is specifically designed for the CIFAR datasets. The architecture of ResNet20 can be divided into several key components. It starts with a convolutional layer, followed by batch normalization and the rectified linear unit (ReLU) activation function. The network then consists of a sequence of residual blocks, each containing two convolutional layers with batch normalization and ReLU activation. Figure 2.6 shows the structure of the residual block of ResNet20. These residual blocks help to capture and learn increasingly complex features as the network deepens.

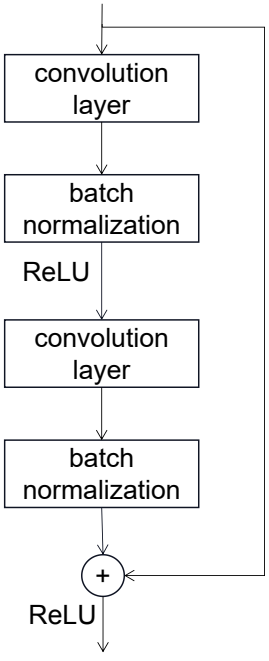

Figure 2.6: Residual block of ResNet20.

ResNet20 also includes downsampling blocks, which reduce the spatial dimensions of the feature maps while increasing the number of filters. This downsampling is typically achieved through convolutional layers with a stride of 2, which reduces the width and height of the feature maps. The downsampling blocks allow the network to learn both low-level and high-level features efficiently. Figure 2.7 shows the structure of residual block of ResNet20.

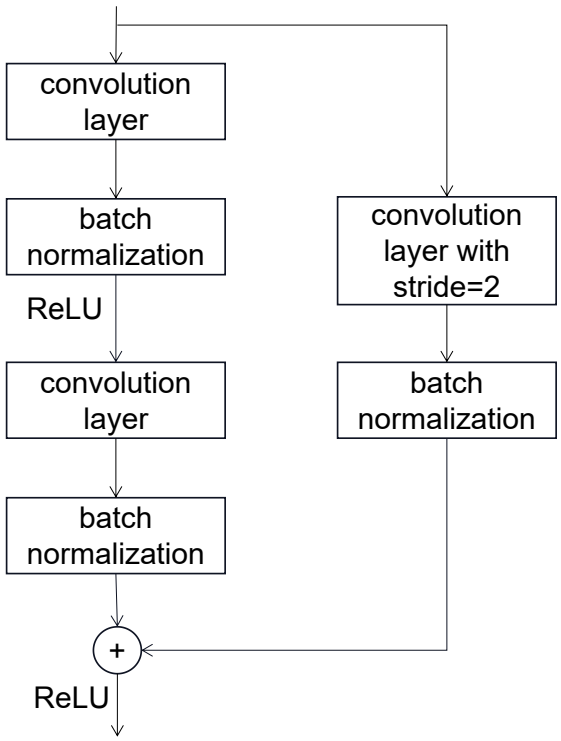

Figure 2.7: Downsampling block of ResNet20.

Finally, ResNet20 ends with an average pooling layer, which reduces the spatial dimensions of the feature maps to a fixed size, and a fully connected layer for classification. The output of the fully connected layer represents the predicted class probabilities. Figure 2.8 shows the network structure of ResNet20.

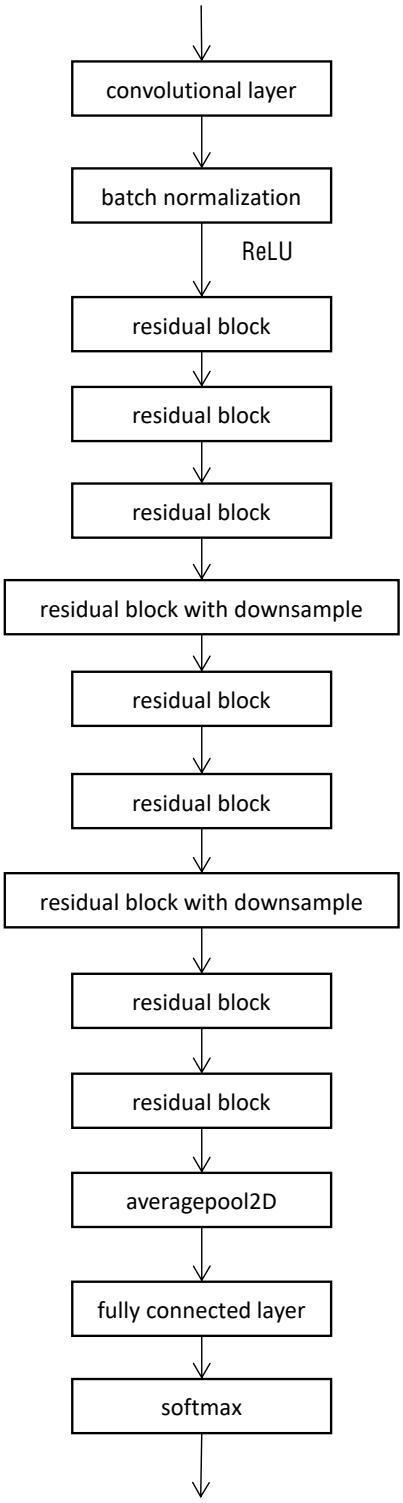

Figure 2.8: Illustration of the structure of ResNet20. It contains 7 residual blocks (with total 14 convolutional layers) and 2 downsampling blocks (with total 6 convolutional layers).

2.3 Quantization of neural network

In recent years, the significance of large neural networks has grown tremendously alongside the advancement of neural networks. However, the utilization of such large models often leads to significant storage requirements and demands substantial computational resources. But in reality, computing resources are often constrained. In order to address the increasing demand for deploying models on resource-constrained devices and improving inference speed, various approaches to model compression have been proposed.

Neural network prunning: One effective strategy to decrease the memory footprint and computational cost of neural networks is to apply pruning [28]. The essential concept behind pruning is to remove redundant or less significant parameters from the network. By pruning neurons with low saliency, a sparse computational graph is created [14]. However, pruning is a non-trivial task and requires careful consideration. Pruning methods should balance the removal of redundant components with preserving the network's representational capacity. Furthermore, strategies such as regularization and fine-tuning after pruning can be employed to mitigate the potential loss of performance due to pruning.

Knowledge distillation: Another method for model compression is knowledge distillation [16]. It involves training a large model and then using it to generate the target label for training a more compact model (like a teacher teaching students) [2]. Despite the extensive research on distillation, a significant challenge lies in achieving high compression ratios through distillation alone [14]. Compared to pruning, which can maintain performance with compression ratios of at least 4 times, knowledge distillation methods often experience noticeable accuracy degradation, especially when aggressively compressed [14]. However, the combination of knowledge distillation with other techniques, such as quantization and pruning, has shown remarkable success in overcoming this limitation [27].

Neural network quantization: Neural network quantization aims to address this problem from another perspective, namely to reduce the memory cost for each parameter by reducing their underlying bits. Because neural networks typically use 32-bit floating point numbers (FP32) to represent weights and activations, which consume significant memory and computational resources. By means of quantization, the network parameters can be represented with fewer bits. This reduces the model size and saves computational resources. Related to neural networks, quantization is work in neuroscience that suggests that the human brain stores information in a discrete/quantized form, rather than in a continuous form [24]. A popular rationale for this idea is that information stored in continuous form will inevitably get corrupted by noise (which is always present in the physical environment, including our brains, and which can be induced by thermal, sensory, external, synaptic noise, etc.) [9]. However, discrete signal representations can be more robust to such low-level noise [20]. Neural networks mimic the working principle of the human brain. Therefore, this theory is applied to artificial Neural Networks.

There are different approaches to quantization. **Post training quantization** is a common method in that the pre-trained neural network is quantized after it has been trained using standard techniques like backpropagation. However, given a trained model, Post training quantization may introduce a perturbation to the trained model

parameters, and this pushes the model away from the point to which it had converged when it was trained with floating point precision [14]. This result in loss of accuracy.

Another popular approach is to use Quantization aware training (QAT). Unlike Post training quantization, which quantizes a network after training, QAT incorporates the quantization process during the training phase. The forward and backward passes are executed on the floating-point quantized model, but the model parameters are quantized in the forward pass [13]. However, the gradient of the quantization operator is either zero or infinity, i.e., the parameters can not be updated through gradient-based methods. In order to solve this problem, the Straight through estimator (STE) [7] is introduced. STE approximates the gradients of the non-differentiable function using continuous relaxation. During the forward pass, the STE applies the non-differentiable operation to the input, obtaining a continuous relaxation of the discrete variable. During the backward pass, instead of applying the derivative of the non-differentiable function, the gradients are directly passed through as if the operation were differentiable [32]. However, the target use cases of STE are strongly limited. Firstly, STE performs badly for ultra low-precision quantization, such as binary quantization [4]. Secondly, the relaxed problem may have a different optimal solution than the actual problem, especially for large, deep networks.

We propose to use ADMM, which is a gradient-free optimization method, to easily integrate quantization into the training process.

Then we propose a layerwise quantization approach using ADMM. For each layer of the neural network, we quantize the parameters through the ADMM approach, followed by training parameters of non-quantized layers.

2.4 Alternating Direction Method of Multipliers

Alternating direction method of multipliers (ADMM) is an algorithm that is intended to blend the decomposability of dual ascent with the superior convergence properties of the method of multipliers [20]. The algorithm solves problems in the form:

$$\begin{aligned} &\text{minimize } f(x) + g(z) \\ &\text{subject to } Ax + Bz = c, \end{aligned} \tag{2.3}$$

with variables $x \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$, where $A \in \mathbb{R}^{p \times n}$, $A \in \mathbb{R}^{p \times m}$, and $c \in \mathbb{R}^p$. We form the augmented Lagrangian:

$$L(x, z, \lambda) = f(x) + g(z) + \frac{\omega}{2}\|Ax + Bz - c\|_2^2 + \lambda(Ax + Bz - c), \tag{2.4}$$

where λ is Lagrangian multiplier, ω is the penalty factor, and $L(\cdot)$ is the objective function.

To solve this problem, the variables can be optimized iteratively and alternatively

$$x^{k+1} := \arg\min_x L(x, z^k, \lambda^k), \tag{2.5}$$

$$z^{k+1} := \arg\min_z L(x^{k+1}, z, \lambda^k), \tag{2.6}$$

$$\lambda^{k+1} := \lambda^k + \omega(Ax^{k+1} + Bz^{k+1} - c). \tag{2.7}$$

2.5 FISTA algorithm

Fast iterative shrinkage-thresholding algorithm (FISTA) is an iterative optimization method used for solving sparse regularization problems [6]. It is particularly effective in dealing with data that exhibit sparse structures. Suppose that the optimization problem with regularization terms is defined as

$$\text{minimize}\{\text{F(x)} = \text{f(x)} + \text{g(x)}\}, \tag{2.8}$$

where the function g(x) denotes the regularization term.

Gradient descent algorithm is a general optimization algorithm. But for optimization problems with penalty terms, gradient descent algorithms may require more iterations to handle sparsity.

Then fast iterative shrinkage-thresholding algorithm (FISTA) has been proposed to improve the efficiency of training [6]. FISTA is based on the idea of gradient descent, but the starting point of each iteration is chosen more intelligently in the iterative process to achieve a faster iteration speed.

A quadratic approximation model is generally used to solve the optimization problem Equation 2.8. The Equation 2.8 is approximated at point y. The quadratic approximation function of Equation 2.8 is given by

$$Q(x, y) = f(y) + \langle x - y, \nabla f_y \rangle + \frac{\zeta}{2}\|x - y\|_2^2 + g(x). \tag{2.9}$$

Ignoring its constant term $f(y)$ and ∇f_y, Equation 2.9 can be simplified as

$$Q(x, y) = \frac{\zeta}{2}\|x - y + \frac{1}{\zeta}\nabla f_y\|_2^2 + g(x). \tag{2.10}$$

The minimum of Equation 2.10 is

$$\begin{aligned} P(y) &= \arg\min_x Q(x, y) \\ &= \arg\min_x \frac{\zeta}{2}\|x - y + \frac{1}{\zeta}\nabla f_y\|_2^2 + g(x), \end{aligned} \tag{2.11}$$

where P is the proximal operator of Equation 2.10.

Algorithm 1 FISTA Algorithm

Initialize: $t_1 = 1$, $y_1 = x_0$, $k = 1$
while not converged **do**
  Update: $x_k = P(y_k)$
  Update: $t_{k+1} \leftarrow \frac{1+\sqrt{1+4{t_k}^2}}{2}$
  Update: $y_{k+1} \leftarrow x_k + (\frac{t_k - 1}{t_k})(x_k - x_{k-1})$
  Update: $k \leftarrow k + 1$
end while

3. Objective function and algorithm

3.1 Objective function of neural network

Our approach revolves around decoupling the model into a series of individual subproblems. While passing the output of linear operator W_l through activation function h_l, we use a new variable $z_l = W_l a_l$ to store the output of linear operator and the output of activation function is denoted as $a_l = h_l(z_l)$. And The loss function of our neural network is denoted as $\mathcal{L}(\cdot)$. Then we could regard the original optimization problem as the following objective function:

$$\begin{aligned} &\underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}}\ \mathcal{L}(z_L, y) \\ &\text{subject to } z_l = W_l a_{l-1},\ for\ l = 1, 2, \dots L \\ &\qquad\qquad a_l = h_l(z_l),\ for\ l = 1, 2, \dots L-1. \end{aligned} \tag{3.1}$$

To solve Equation 3.1 we relax the constraints by adding penalty functions to the objective function. The Problem can then be converted to unconstrained problem:

$$\begin{aligned} \underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}}\ &\mathcal{L}(z_L, y) + \frac{\beta}{2}\|z_L - W_L a_{L-1}\|_2^2 \\ &+ \sum_{l=1}^{L-1}[\frac{\beta}{2}\|z_l - W_l a_{l-1}\|_2^2 + \frac{\gamma}{2}\|a_l - h_l(z_l)\|_2^2], \end{aligned} \tag{3.2}$$

where β and γ are constants which control the weight of constraints. To obtain the exact solution, we also need to add a Lagrange multiplier term to Equation 3.2. Then the equation is

$$\begin{aligned} \underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}}\ &\mathcal{L}(z_L, y) + \langle z_L - W_L a_{L-1}, \lambda\rangle + \frac{\beta}{2}\|z_L - W_L a_{L-1}\|_2^2 \\ &+ \sum_{l=1}^{L-1}[\frac{\beta}{2}\|z_l - W_l a_{l-1}\|_2^2 + \frac{\gamma}{2}\|a_l - h_l(z_l)\|_2^2], \end{aligned} \tag{3.3}$$

where λ is a vector of Lagrange multiplier. λ has same dimensions as z_L.

Because the proposed method scheme involves more than two coupled blocks and a non-smooth penalty function, it lies outside the scope of known convergence results for ADMM [29]. If ADMM is applied to Equation 3.2 in a conventional way, each constraint needs one separate Lagrange multiplier vector. This method is highly unstable because of the destabilizing effect of numerous coupled non-smooth, non-convex terms [29]. Taylor proposed that this problem can be solved with the help of Bregman Iteration, which only requires a Lagrange Multiplier [29]. So we only add one Lagrange multiplier for variable z_L. Then we found the objective function with only one Lagrange multiplier is more stable than the classical ADMM.

3.2 Objective function of quantized neural network

In order to quantize the neural network, we restrict the weight to be integer. We denote C as a set of these integers, where $C = \{-(2^N-1), ..., -3, -2, -1, 0, 1, 2, 3, ..., 2^N-1\}$. N represents the number of bits. We also introduce a scaling factor α. α is always positive and could be different in various layers. α_l denotes the scaling factor of l-th layer. The entries of the matrix W_l are constrained into the set $C_l = \{-(2^N-1)\alpha_l, ... -3\alpha_l, -2\alpha_l, \alpha_l, 0, \alpha_l, 2\alpha_l, 3\alpha_l ... + (2^N-1)\alpha_l\}$. As Figure 3.1 shows, taking the case of 1 bit as an example, by introducing scaling factors α_l, the solution space for every entry of the weight matrix changes from points to lines.

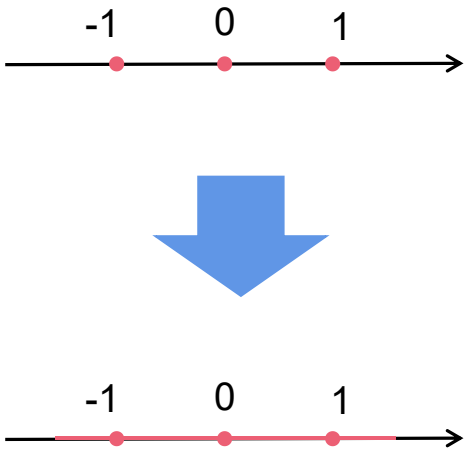

Figure 3.1: After introducing α_l, the solution space of element of weight matrix changes from points to lines.

So we add some new constraints $W_l \in C_l$ on the objective function to represent the quantized neural network. The objective function of quantized neural network is

$$\begin{aligned} \underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} \; & \mathcal{L}(z_L, y) + \langle z_L - W_L a_{L-1}, \lambda \rangle + \frac{\beta}{2}\|z_L - W_L a_{L-1}\|_2^2 \\ & + \sum_{l=1}^{L-1}[\frac{\beta}{2}\|z_l - W_l a_{l-1}\|_2^2 + \frac{\gamma}{2}\|a_l - h_l(z_l)\|_2^2] \\ \text{subject to } & W_l \in C_l \; for \; l = 1, 2, \dots L. \end{aligned} \tag{3.4}$$

This optimization problem is an integer programming problem. It is very complex to solve it using traditional integer programming method, such as branch and bound, especially for the large model with huge solution space. Inspired by Huang et al. [17], we introduce an additional auxiliary variable G which is subject to the discrete set. The objective function can be rewritten as:

$$\begin{aligned} \underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} \; & \mathcal{L}(z_L, y) + \langle z_L - W_L a_{L-1}, \lambda \rangle + \frac{\beta}{2} \|z_L - W_L a_{L-1}\|_2^2 \\ & + \sum_{l=1}^{L-1} [\frac{\beta}{2} \|z_l - W_l a_{l-1}\|_2^2 + \frac{\gamma}{2} \|a_l - h_l(z_l)\|_2^2] \\ \text{subject to } & W_l \; = G_l \; for \; l = 1, 2, \ldots L, \end{aligned} \tag{3.5}$$

where G_l is the dequantized weight of the l-th layer.

Then we add the penalty term on the objective function

$$\begin{aligned} \underset{\{W_l\},\{a_l\},\{z_l\}}{\text{minimize}} \; & \mathcal{L}(z_L, y) + \langle z_L - W_L a_{L-1}, \lambda \rangle + \frac{\beta}{2} \|z_L - W_L a_{L-1}\|_2^2 \\ & + \sum_{l=1}^{L-1} [\frac{\beta}{2} \|z_l - W_l a_{l-1}\|_2^2 + \frac{\gamma}{2} \|a_l - h_l(z_l)\|_2^2] \\ & + \sum_{l=1}^{L} \frac{\mu}{2} \|W_l - G_l\|_2^2 \\ \text{subject to } & G_l \in C_l, \end{aligned} \tag{3.6}$$

where μ is the penalty factor of constraints $W_l = G_l$. The parameters a_l, z_l and W_l can be optimized in continuous space. Then the W_l is projected into discrete space after optimizing in continuous space.

3.3 Algorithm of updating parameters

In order to overcome the problem that ADMM becomes trapped in a modest solution, we adopt the method in [30]. Wang et al. proposed: "If the parameters are updated only from the first layer to final layer, the parameters in the final layer are subject to the parameter update in the first layer. But, the parameters in the final layer contain important information that can be transmitted towards the previous layer to speed up convergence." [30]. According to the method in [30], we update the parameters first in a backward direction and then in a forward direction and use some quadratic approximation to replace the matrix inversion and accelerate the training process. However, as the auxiliary variable, G_l always updates after W_l. It means that we first optimize the W_l in the continuous space and then optimize G_l in the discrete space.

There are two steps involved in updating parameters. The first step is backward updating, where we start updating from the L-th layer and move backward toward the first layer. Within the same layer, the parameters are updated in the following order: $a_l \rightarrow z_l \rightarrow W_l \rightarrow G_l$. The second step is forward updating, where we initiate the update process from the first layer and proceed backward toward the L-th layer. Within the same layer, the parameters are updated in the following order: $W_l \rightarrow G_l \rightarrow z_l \rightarrow a_l$. Figure 3.2 shows the updating process in the case of a 3-layer neural network.

Algorithm 2 Training quantized Neural Network with ADMM approach

Ensure: $\{a_l\}_{l=1}^{L-1}$, $\{W_l\}_{l=1}^{L}$, $\{z_l\}_{l=1}^{L}$, $\{G_l\}_{l=1}^{L}$
Require: training features $\{a_0\}$ and labels $\{y\}$
  Initialize $k = 0$
  while $\{W_l^{k+1}\}_{l=1}^{L}$, $\{G_l^{k+1}\}_{l=1}^{L}$, $\{z_l^{k+1}\}_{l=1}^{L}$, $\{a_l^{k+1}\}_{l=1}^{L}$ not converged **do**
    for l=L to 1 **do**
      if $l \leq L$ **then**
        update a_l^{k+1}
        update z_l^{k+1}
        update W_l^{k+1}
        update G_l^{k+1}
      else
        update z_l^{k+1}
        update W_l^{k+1}
        update G_l^{k+1}
      end if
    end for
    for l=1 to L **do**
      if $l \leq L$ **then**
        update W_l^{k+1}
        update G_l^{k+1}
        update z_l^{k+1}
        update a_l^{k+1}
      else
        update W_l^{k+1}
        update G_l^{k+1}
        update z_l^{k+1}
      end if
    end for
    update λ^{k+1}
    $k \leftarrow k + 1$
  end while

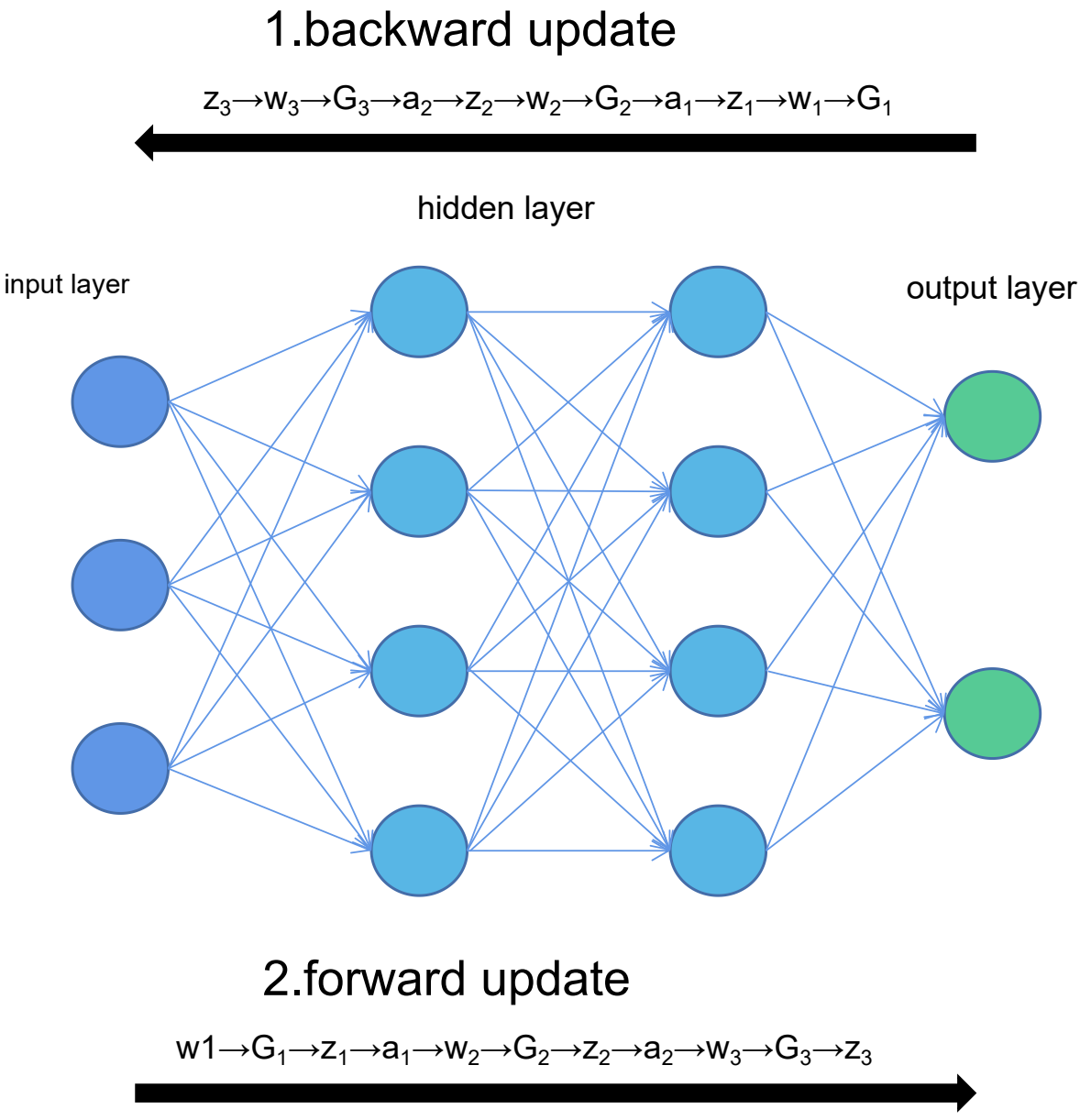

Figure 3.2: The updating process of a 3-Layer Neural Network.

4. Alternating minimization for neural networks

4.1 Update W_l

With the help of ADMM, the original problem can be decoupled into some sub-problems. Firstly, we describe how to minimize the parameter W_l. In the sub-problem of W_l, we consider W_l as the only variable. It means, that z_l, G_l and a_{l-1} are considered to be constants. The sub-problem about W_l is to minimize

$$J_w(W_l) = \frac{\beta}{2}\|z_l - W_l a_{l-1}\|_2^2 + \frac{\mu}{2}\|W_l - G_l\|_2^2, \tag{4.1}$$

where $J_w(\cdot)$ denotes the objective function of the sub-problem of parameter W_l.

We can observe that the function $J_w(\cdot)$ is convex. The minimum of W_l must satisfy the optimality condition:

$$\frac{\mathrm{d}J_w(W_l)}{\mathrm{d}W_l} = 0. \tag{4.2}$$

So the derivative of Equation 4.1 should be zero. We can obtain that W_l must satisfy

$$-\beta(z_l - W_l a_{l-1})a_{l-1}^T + \mu(W_l - G_l) = 0. \tag{4.3}$$

Then we obtain the update of W_l:

$$W_l \leftarrow (\mu G_l + \beta z_l a_{l-1}^T)(\beta a_{l-1} a_{l-1}^T + \mu I)^{-1}. \tag{4.4}$$

We found there is matrix inversion in Equation 4.4. The computation of matrix inversion needs further sub-iterations, produces high time complexity, and leads to a large amount of computing time. Inspired by Wang et al. [30], we use quadratic function and backtracking techniques to approximate the $J_w(W_l)$.

We approximate the $J_w(W_l)$ at the point W_l^k, where W_l^k represents the value of W_l in the k-th iteration. We define $\psi(W_l, \upsilon_l)$ as the quadratic approximation of $J_w(W_l)$. The function $J_w(W_l)$ could be reformulated as

$$\psi(W_l, \upsilon_l) = J_w(W_l^k) + (\nabla_{W_l^k} J_w)^T(W_l - W_l^k) + \frac{\upsilon_l}{2}\|W_l - W_l^k\|_2^2, \tag{4.5}$$

where υ_l denotes the coefficient of quadratic term and $(\nabla_{W_l^k} J_w)$ is the gradient of $J_w(\cdot)$ at the point W_l^k. $\upsilon_l > 0$ is a scalar. Instead of minimizing Equation 4.1, we solve the following problem:

$$W_l^{k+1} \leftarrow \arg\min_{W_l} \psi(W_l, \upsilon_l). \tag{4.6}$$

The function $\psi(W_l, \upsilon_l)$ also must satisfy:

$$\frac{\partial \psi}{\partial W_l} = 0. \tag{4.7}$$

The partial derivative of $\psi(\cdot)$ to W_l is

$$\frac{\partial \psi}{\partial W_l} = \nabla_{W_l^k} J_w + \upsilon_l (W_l - W_l^k). \tag{4.8}$$

So the solution can be obtained by

$$W_l{}^{k+1} \leftarrow W_l^k - \nabla_{W_l^k} J_w / \upsilon_l, \tag{4.9}$$

where the gradient of $J_w(W_l)$ at the point W_l^k is

$$\nabla_{W_l^k} J_w = -\beta (z_l - W_l^k a_{l-1}) a_{l-1}^T + \mu (W_l^k - G_l). \tag{4.10}$$

Now, the key problem is how to choose an appropriate parameter υ_l. We use the backtracking algorithm. The Algorithm 3 shows pseudocode of updating W_l^{k+1}, where W_l^{k+1} denotes the value of W_l in $(k+1)$-th iteration. We set κ as the scaling factor for the backtracking algorithm and κ is always greater than one. Then We set up a loop that stops until condition $J_w(W_l{}^{k+1}) \leq \psi(W_l{}^{k+1}, \upsilon_l)$. Because $\kappa > 1$, the υ_l becomes larger and larger as the number of iterations increases. The W_l^{k+1} is getting close to W_l^k. As $\psi(W_l{}^k, \upsilon_l) = J_w(W_l^k)$, the point W_l^k satisfy the stopping condition. This prevents Algorithm 3 from falling into an infinite loop.

Algorithm 3 The Backtracking Algorithm to update W_l^{k+1}

Ensure: υ_l, W_l^{k+1}
Require: z_l, a_{l-1}, G_l, W_l^k, β, μ
  Initialize υ_l and choose $\kappa > 1$
  $W_l{}^{k+1} \leftarrow W_l^k - \nabla_{W_l^k} L / \upsilon_l$
  while $J_w(W_l{}^{k+1}) > \psi(W_l{}^{k+1}, \upsilon_l)$ **do**
    $\upsilon_l \leftarrow \kappa \upsilon_l$
    $W_l{}^{k+1} \leftarrow W_l^k - \nabla_{W_l^k} J_w / \upsilon_l$
  end while
  Output $W_l{}^{k+1}$
  Output υ_l

4.2 Update G_l

After optimizing W_l in continuous space, we need to optimize the discrete values G_l. The update for G_l requires minimizing

$$\begin{aligned} J_G(G_l) =& \frac{\mu}{2}\|W_l - G_l\|_2^2 \\ &\text{subject to } G_l \in C_l, \end{aligned} \tag{4.11}$$

where C_l is the set $\{0, \pm\alpha_l, \pm 2\alpha_l, ..., \pm(2^N - 1)\alpha_l\}$ and $J_G(\cdot)$ is the objective function of sub-problem G_l. We could observe that all parameters in Equation 4.11 are decoupled. So its solution does not contain any matrix inversion. We could solve it directly. First, We introduce an auxiliary variable Q_l for each layer. Q_l has same dimension as W_l. And Q_l is constrained in the discrete set $C = \{0, \pm 1, \pm 2, ..., \pm(2^N - 1)\}$. Then the Equation 4.11 can be reformulated as following equation:

$$\begin{aligned} J_G(Q_l, \alpha_l) =& \frac{\mu}{2}\|W_l - \alpha_l Q_l\|_2^2 \\ &\text{subject to } Q_l \in C. \end{aligned} \tag{4.12}$$

We use an iterative approach to alternatively optimize these two variables. It means we fix one variable and optimize another variable. Firstly, we consider Q_l as constant and optimize α_l. As the penalty factor μ does not affect the result of the optimization, it can be ignored. We take a two-dimensional W_l matrix as an example. We assume that the size of W_l is c and d. w_l^{ij} represents the entry of the matrix W_l in the i-th row and j-th column. q_l^{ij} represents the entry of the matrix Q_l in the i-th row and j-th column. We can rewrite Equation 4.12 as

$$\begin{aligned} J_G(Q_l, \alpha_l) &= \|W_l - \alpha_l Q_l\|_2^2 \\ &= \sum_{i=1}^{c}\sum_{j=1}^{d}((w_l^{ij})^2 + \alpha_l^2(q_l^{ij})^2 - 2\alpha_l(w_l^{ij})(q_l^{ij})) \\ &\text{subject to } Q_l \in \{-(2^N - 1), ..., -2, -1, 0, 1, 2, ..., (2^N - 1)\}. \end{aligned} \tag{4.13}$$

If Q_l is fixed, the function $J_G(\cdot)$ is convex. And α_l is continuous. To minimize Equation 4.13, it must satisfy the optimality condition:

$$\frac{\partial J_G}{\partial \alpha_l} = 0. \tag{4.14}$$

The derivative of $J_G(\cdot)$ with respect to α_l is

$$\frac{\partial J_G}{\partial \alpha_l} = \sum_{i=1}^{c}\sum_{j=1}^{d}(2\alpha_l(q_l^{ij})^2 - 2w_l^{ij}q_l^{ij}) = 0. \tag{4.15}$$

Then we obtain the update of α_l

$$\alpha_l \leftarrow \frac{\sum_{i=1}^{c}\sum_{j=1}^{d} w_l^{ij}q_l^{ij}}{\sum_{i=1}^{c}\sum_{j=1}^{d}(q_l^{ij})^2}. \tag{4.16}$$

Secondly, we optimize the parameter Q_l. Now α_l is considered as constant. All entries of Q_l must be integers, so the function $J_G(\cdot)$ is not differentiable. Therefore,

the optimality condition is not suitable for optimizing Q_l. The optimal Q_l can be obtained by projection of $\frac{W_l}{\alpha_l}$

$$Q_l = \Pi_{\{0,\pm1,\pm2,\ldots,\pm(2^N-1)\}}(\frac{W_l}{\alpha_l}), \tag{4.17}$$

where Π represents the projection operator. The projection onto a discrete set corresponds to finding the point within the set that is nearest to the given point.

Algorithm 4 shows the pseudocode of updating G_l. We adopt the early stop to stop the loop.

Algorithm 4 The Algorithm to update Q_l and α_l

Ensure: Q_l, α_l
Require: W_l
  Initialize Q_l and α_l, maximum_patience, best_loss $= \infty$, patience $= 0$
  while the maximum iteration is not reached **do**
    $\alpha_l \leftarrow \frac{\sum_{i=1}^{c}\sum_{j=1}^{d} w_l^{ij} q_l^{ij}}{\sum_{i=1}^{c}\sum_{j=1}^{d}(q_l^{ij})^2}$
    $Q_l = \Pi_{\{0,\pm1,\pm2,\ldots,\pm2^N-1\}}(\frac{W_l}{\alpha_l})$
    if $\|W_l - \alpha_l Q_l\|_2^2 <$ best_loss **then**
      best_loss $= \|W_l - \alpha_l Q_l\|_2^2$
      $patience = 0$
    else
      $patience = patience + 1$
    end if
    if $patience \geq$ maximium_patience **then**
      break
    end if
  end while
  $G_l = \alpha_l Q_l$
  Output G_l

4.3 Update z_l

The update for z_l ($l < L$) requires minimizing

$$J_z(z_l) = \frac{\beta}{2}\|z_l - W_l a_{l-1}\|_2^2 + \frac{\gamma}{2}\|a_l - h_l(z_l)\|_2^2. \tag{4.18}$$

This problem exhibits non-convexity and non-quadratic behaviour due to the presence of the non-linear term $h(\cdot)$. So the optimality condition can not be used to optimize the z_l. However, a notable advantage is that the non-linearity $h(\cdot)$ operates element-wise on its argument, allowing the individual entries in z_l to be treated independently. Solving Equation 4.18 is easy when $h(\cdot)$ is a piecewise linear function, as it can be solved in close form; common piecewise linear choices for $h(\cdot)$ include rectified linear units and non-differentiable sigmoid functions [29]. The rectified linear units function is

$$h(x) = \begin{cases} x, \ if \ x > 0 \\ 0, \ otherwise. \end{cases} \tag{4.19}$$

The non-differentiable sigmoid equation is

$$h(x) = \begin{cases} 1, \ if \ x \geq 1 \\ x, \ if \ 0 < x < 1 \\ 0, \ otherwise. \end{cases} \tag{4.20}$$

For more sophistical choices of $h(\cdot)$, including smooth sigmoid curves, the problem can be solved quickly with a lookup table of pre-computed solutions [29].

We define the entry of matrix z_l as z_l^{ij}. Taking rectified linear units as an example, suppose $z_l > 0$ (all entries of z_l are greater than 0), the $h_l(z_l)$ is equal to z_l. If $z_l > 0$, the objective function of optimizing z_l is

$$\underset{z_l}{\text{minimize}} \frac{\beta}{2}\|z_l - W_l a_{l-1}\|_2^2 + \frac{\gamma}{2}\|a_l - z_l\|_2^2. \tag{4.21}$$

To satisfy optimality condition, the derivative of the objective equation of updating z_l must be equal to zero:

$$\beta(z_l - W_l a_{l-1}) - \gamma(a_l - z_l) = 0. \tag{4.22}$$

The solution of Equation 4.22 is obtained by

$$z_l = \frac{\beta W_l a_{l-1} + \gamma a_l}{\beta + \gamma}. \tag{4.23}$$

After updating z_l, we must check if the precondition $z_l^{ij} > 0$ for each element is satisfied. We keep the matrix entries that satisfy the precondition.

Suppose $z_l < 0$ (all elements of z_l are smaller than 0), the $h_l(z_l)$ is equal to zero. The objective function of optimizing z_l is

$$\underset{z_l}{\text{minimize}} \frac{\beta}{2}\|z_l - W_l a_{l-1}\|_2^2 + \frac{\gamma}{2}\|a_l\|_2^2. \tag{4.24}$$

Through optimality condition, we can obtain the update of z_l

$$z_l = W_l a_{l-1}. \tag{4.25}$$

Same as the case where z_l is greater than zero, we have to check if the precondition $z_l^{ij} < 0$ for each element is satisfied. We keep the matrix elements that satisfy the precondition.

It is worth noting that there are some special values of entries in the matrix z_l. The first special case is: When these entries are assumed to be greater than zero, the update result is less than zero; when they are assumed to be less than zero, the update result is greater than zero. Both preconditions are not satisfied. In this case, we make these entries equal to the critical value 0. The second special case is: When these entries are assumed to be greater than zero, the update result is greater than zero; when they are assumed to be less than zero, the update result is less than zero. It means two preconditions are satisfied at the same time. In this case, we have to compare which update result has a smaller target value and choose the better one.

Algorithm 5 shows the pseudocode of updating z_l. $\tilde{z_1}$ is the solution of Equation 4.21, if we suppose $z_l > 0$. $\tilde{z_2}$ is the solution of Equation 4.24 if we suppose $z_l < 0$. $\tilde{z_1}^{ij}$ and $\tilde{z_2}^{ij}$ are the elements of $\tilde{z_1}$ and $\tilde{z_2}$.

Algorithm 5 The Algorithm to update z_l

Ensure: z_l
Require: a_l, a_{l-1}, W_l, β, γ
  $\tilde{z_1} = \frac{\beta W_l a_{l-1} + \gamma a_l}{\beta + \gamma}$
  $\tilde{z_2} = W_l a_{l-1}$
  for each element $\tilde{z_1}^{ij}$ in $\tilde{z_1}$ **do**
    if $\tilde{z_1}^{ij} \leq 0$ **then**
      $\tilde{z_1}^{ij} = 0$
    end if
  end for
  for each element $\tilde{z_2}^{ij}$ in $\tilde{z_2}$ **do**
    if $\tilde{z_2}^{ij} \geq 0$ **then**
      $\tilde{z_2}^{ij} = 0$
    end if
  end for
  for each elements $\tilde{z_1}^{ij}$, $\tilde{z_2}^{ij}$ in $\tilde{z_1}$,$\tilde{z_2}$ **do**
    if $\tilde{z_1}^{ij} > 0$ and $\tilde{z_2}^{ij} < 0$ **then**
      $e1 = \frac{\beta}{2}\|\tilde{z_1}^{ij} - w_l^{ij} \cdot a_{l-1}^{ij}\|_2^2 + \frac{\gamma}{2}\|a_l^{ij} - h_l(\tilde{z_1}^{ij})\|_2^2$
      $e2 = \frac{\beta}{2}\|\tilde{z_2}^{ij} - w_l^{ij} \cdot a_{l-1}^{ij}\|_2^2 + \frac{\gamma}{2}\|a_l^{ij} - h_l(\tilde{z_2}^{ij})\|_2^2$
      if $e1 < e2$ **then**
        $z_l^{ij} = \tilde{z_1}^{ij}$
      else
        $z_l^{ij} = \tilde{z_2}^{ij}$
      end if
    end if
  end for

4.4 Update a_l

The update for a_l requires minimizing

$$J_a(a_l) = \frac{\beta}{2}\|z_{l+1} - W_{l+1}a_l\|_2^2 + \frac{\gamma}{2}\|a_l - h_l(z_l)\|_2^2. \tag{4.26}$$

where $J_a(\cdot)$ denotes the objective function of sub-problem of a_l. When the optimality condition is satisfied, we obtain the update for a_l

$$a_l \leftarrow (\beta W_{l+1}^T W_{l+1} + \gamma I)^{-1}(\beta W_{l+1}^T z_{l+1} + \gamma h_l(z_l)). \tag{4.27}$$

There is still the inversion in Equation 4.27. Same as the process of updating W_l, we use quadratic function and backtracking techniques to approximate the Equation 4.26. .

We approximate the $J_a(a_l)$ at the point a_l^k, where a_l^k represents the value of a_l in the k-th iteration. We define $\phi(a_l, \theta_l)$ as the quadratic approximation of $J_a(a_l)$. The $J_a(a_l)$ could be reformulated as

$$\phi(a_l, \theta_l) = J_a(a_l^k) + (\nabla_{a_l^k} J_a)^T(a_l - a_l^k) + \frac{\theta_l}{2}\|a_l - a_l^k\|_2^2, \tag{4.28}$$

where $\theta_l > 0$ is the coefficient of quadratic term of $\phi(\cdot)$ and $(\nabla_{a_l^k} J_a)$ is the gradient of $J_a(\cdot)$ at the point a_l^k. Instead of minimizing the Equation 4.26, we solve the following problem:

$$a_l^{k+1} \leftarrow \arg\min_{a_l} \phi(a_l, \theta_l). \tag{4.29}$$

The function $\phi(\cdot)$ is a convex function. When a_l locates at position of minimum, the function $\phi(a_l, \theta_l)$ must satisfy:

$$\frac{\partial \phi}{\partial a_l} = 0. \tag{4.30}$$

The partial derivative of $\phi(\cdot)$ to a_l is

$$\frac{\partial \phi}{\partial a_l} = \nabla_{a_l^k} J_a + \theta_l(a_l - a_l^k). \tag{4.31}$$

The solution of Equation 4.29 can be obtained by

$$a_l^{k+1} \leftarrow a_l^k - \nabla_{a_l^k} J_a/\theta_l, \tag{4.32}$$

where the gradient of $J_a(a_l)$ at the point a_l^k is

$$\nabla_{a_l^k} J_a = -\beta W_{l+1}^T(z_{l+1} - W_{l+1}a_l^k) + \gamma(a_l^k - h_l(z_l)). \tag{4.33}$$

Same as updating W_l, we use the backtracking algorithm to choose suitable parameter θ_l. The Algorithm 6 shows pseudocode for updating a_l^{k+1}. We set τ as the scaling factor for the backtracking algorithm and τ is always greater than one. Then We set up a loop that stops until condition $J_a({a_l}^{k+1}) \leq \phi({a_l}^{k+1}, \theta_l)$.

Algorithm 6 The Backtracking Algorithm to update a_l^{k+1}

Ensure: θ_l, a_l^{k+1}
Require: z_l, W_{l+1}, z_{l+1}, a_l^k, β, γ
  Initialize θ_l and choose $\tau > 1$
  ${a_l}^{k+1} \leftarrow a_l^k - \nabla_{a_l^k} J_G/\theta_l$
  while $J_a({a_l}^{k+1}) > \phi(\tilde{a_l}^{k+1}, \theta_l)$ **do**
    $\theta_l \leftarrow \tau\theta_l$
    ${a_l}^{k+1} \leftarrow a_l^k - \nabla_{a_l^k} J/\theta_l$
  end while
  Output ${a_l}^{k+1}$
  Output θ_l

4.5 Update z_L

Compared with updating z_l, there is an additional term in the equation for updating z_L:

$$\underset{z_L}{\text{minimize}}\, \mathcal{L}(z_L, y) + \frac{\beta}{2}\|z_L - W_L a_{L-1}\|_2^2 + \langle z_L - W_L a_{L-1}, \lambda\rangle. \tag{4.34}$$

FISTA Algorithm is applied to optimize z_L. We define

$$f(z_L) = \mathcal{L}(z_L, y) + \langle z_L - W_L a_{L-1}, \lambda\rangle. \tag{4.35}$$

Then we approximate the Equation 4.34 at the point $\hat{z}_L$. The approximate function φ is

$$\varphi(z_L, \hat{z}_L) = \frac{\eta}{2}\|z_L - \hat{z}_L + \frac{1}{\eta}\nabla f_{\hat{z}_L}\|_2^2 + \frac{\beta}{2}\|z_L - W_L a_{L-1}\|_2^2, \tag{4.36}$$

where η denotes coefficient of quadratic term of the function $\varphi(\cdot)$. The minimum of Equation 4.36 is

$$P(\hat{z}_L) = \arg\min_{z_L} \varphi(z_l, \hat{z}_L), \tag{4.37}$$

where $P(\cdot)$ denotes the proximal operator.

To obtain the solution of Equation 4.37, the optimal solution must satisfy the optimality condition:

$$\frac{\partial\varphi}{\partial z_L} = 0. \tag{4.38}$$

Then we obtain the update for z_L

$$z_L \leftarrow \frac{\eta\hat{z}_L + \beta W_L a_{L-1} - \nabla f_{\hat{z}_L}}{\eta + \beta}. \tag{4.39}$$

Now, the key question is how to choose the starting point $\hat{z}_L$ for each iteration of the update. The Algorithm 7 shows how to update z_L and choose $\hat{z}_L$.

In our project, we adopt the cross-entropy function with softmax as the loss function. The gradient of f at point $\hat{z_L}$ is

$$\nabla f_{\hat{z_L}} = S(z_L) - y + \lambda \tag{4.40}$$

where $S(\cdot)$ denotes the softmax function.

Algorithm 7 FISTA Algorithm for updating z_L

Initialize: $t_1 = 1, \hat{z}_L^1 = z_L^0, s = 1$
while not converged **do**
  Update: $z_L^s \leftarrow \frac{\eta \hat{z}_L + \beta W_L a_{L-1} - \nabla f_{\hat{z}_L}}{\eta + \beta}$
  Update: $t^{s+1} \leftarrow \frac{1+\sqrt{1+4t^{s^2}}}{2}$
  Update: $\hat{z}_L^{s+1} \leftarrow z_L^s + (\frac{t^s - 1}{t^s})(z_L^s - z_L^{s-1})$
  Update: $s \leftarrow s + 1$
end while

4.6 Update Lagrangian Multiplier

In ADMM, the Lagrange multiplier update is gradient ascent in the dual space. There is only one Lagrange multiplier in the objective function. It can be updated through the function:

$$\lambda \leftarrow \lambda + \beta(z_L - W_L a_{L-1}). \tag{4.41}$$

5. Layerwise Quantization with ADMM Approach

We also propose a method to quantize neural networks layer by layer. There are two main steps to quantize each layer of the neural network:
(1) quantize the parameter with ADMM approach
(2) update the parameters of remaining non-quantized layers

5.1 Quantization with ADMM approach

Suppose a pre-trained neural network with L layers is given, the loss function of the pre-trained neural network is denoted by $R(\cdot)$. In our method, the parameters of quantized layers are used as the input for quantizing the subsequent layer. As shown in Figure 5.1, suppose one has quantized the pre-trained network up to the $(l-1)$-th layer. Then to quantize l-th layer, we consider the $\overline{W}_{[1,2,\dots,l-1]} = \{\overline{W}_1, \overline{W}_2, ..., \overline{W}_{l-1}\}$ as input, where the $\overline{W}$ represents the quantized parameters. The set $W_{[l+1,\dots,L]}$ denotes the remaining non-quantized layers.
The loss function of the partially quantized neural network is given by

$$\underset{W_l}{\text{minimize}}\, R(\overline{W}_{[1,2,\dots,l-1]}, W_l, W_{[l+1,\dots,L]}). \tag{5.1}$$

Then we introduce an auxiliary variable G_l. To quantize the parameter W_l, G_l is constrained in a discrete set $C_l = \{-(2^N-1)\alpha_l, ..., -2\alpha_l, -\alpha_l, 0, \alpha_l, 2\alpha_l, ..., (2^N-1)\alpha_l\}$. In order to quantize l-th layer, We define

$$\begin{aligned} F(W_l, G_l, \lambda) = {} & R(\overline{W}_{[1,2,\dots,l-1]}, W_l, W_{[l+1,\dots,L]}) \\ & + \frac{\rho}{2}\|W_l - G_l\|_2^2 + \langle W_l - G_l, \lambda\rangle, \end{aligned} \tag{5.2}$$

where $\rho > 0$ denotes the penalty factor and $F(\cdot)$ denotes the objective function.

The objective function should be

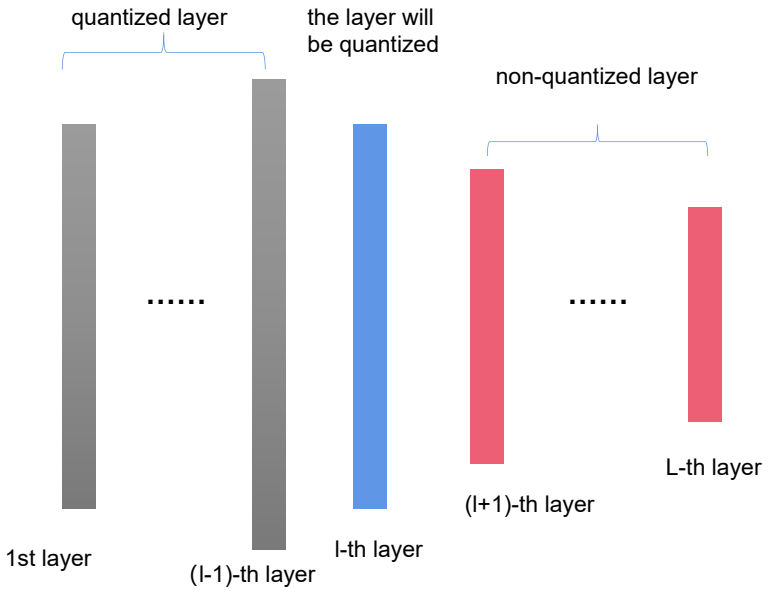

Figure 5.1: A L-layer neural network with $l-1$ quantized layers and l-th will be quantized with ADMM approach.

$$\begin{aligned} &\underset{W_l, G_l}{\text{minimize}}\ F(W_l, G_l, \lambda) \\ &\textit{subject to}\ G_l \in C_l \end{aligned} \tag{5.3}$$

Using ADMM, the Equation 5.3 could be split into 3 sub-problems:

$$W_l \leftarrow \arg\min_{W_l} F(W_l, G_l, \lambda) \tag{5.4}$$

$$G_l \leftarrow \arg\min_{G_l} F(W_l, G_l, \lambda) \tag{5.5}$$

$$\lambda \leftarrow \lambda + \rho(W_l - G_l). \tag{5.6}$$

5.1.1 Update W_l

Equation 5.2 could be reformulated as

$$\begin{aligned} F(W_l, G_l, \lambda) =& R(\overline{W}_{[1,2,\ldots,l-1]}, W_l, W_{[l+1,\ldots,L]}) \\ &+ \frac{\rho}{2}\|W_l - G_l + \frac{\lambda}{\rho}\|_2^2 - \frac{1}{2\rho}\|\lambda\|_2^2. \end{aligned} \tag{5.7}$$

Since λ and ρ are constants when updating W_l, they do not affect the objective value. So the constant term $\frac{1}{2\rho}\|\lambda\|_2^2$ can be omitted. The objective function of the sub-problem for updating W_l could be simplified as

$$\begin{aligned} \underset{W_l}{\text{minimize}} & R(\overline{W}_{[1,2,\ldots,l-1]}, W_l, W_{[l+1,\ldots,L]}) \\ &+ \frac{\rho}{2}\|W_l - G_l + \frac{\lambda}{\rho}\|_2^2. \end{aligned} \tag{5.8}$$

W_l can be optimized simply by a gradient descent algorithm.

5.1.2 Update G_l

The sub-problem of G_l is

$$\begin{aligned} \underset{G_l}{\text{minimize}}\ & \frac{\rho}{2}\|W_l + \frac{\lambda}{\rho} - G_l\|_2^2 \\ \text{subject to}\ & G_l \in C_l, \end{aligned} \tag{5.9}$$

where $C_l = \{-(2^N-1)\alpha_l, ..., -2\alpha_l, -\alpha_l, 0, \alpha_l, 2\alpha_l, ..., (2^N-1)\alpha_l\}$. For convenience, we denote $W_l + \frac{\lambda}{\rho}$ as V_l. The projection of V_l onto C_l can be formulated as

$$\begin{aligned} \underset{G_l}{\text{minimize}}\ & \frac{\rho}{2}\|V_l - G_l\|_2^2 \\ \text{subject to}\ & G_l \in C_l. \end{aligned} \tag{5.10}$$

Separating the scaling factor from the constraints, the objective function can be equivalently formulated as:

$$\begin{aligned} \underset{Q_l,\alpha_l}{\text{minimize}}\ & \frac{\rho}{2}\|V_l - \alpha_l Q_l\|_2^2 \\ \text{subject to}\ & Q_l \in C, \end{aligned} \tag{5.11}$$

where $C = \{-(2^N-1), ..., -2, -1, 0, 1, 2, ..., 2^N-1\}$. Q_l is a matrix whose all elements are integers. Q_l has same dimension as W_l. $\alpha_l > 0$ is scaling factor. We iteratively optimize α_l and Q_l. Firstly, we fix the Q_l and update α_l. We take a two-dimensional W_l matrix as an example. We suppose that the size of V_l is c and d. v_l^{ij} represents the element of the matrix V_l in the i-th row and j-th column. q_l^{ij} represents the element of the matrix Q_l in the i-th row and j-th column. We can rewrite Equation 5.11 as

$$\begin{aligned} \underset{Q_l,\alpha_l}{\text{minimize}}\ & \sum_{i=1}^{c}\sum_{j=1}^{d}((v_l^{ij})^2 + \alpha_l^2(q_l^{ij})^2 - 2\alpha_l(v_l^{ij})(q_l^{ij})) \\ \text{subject to}\ & Q_l \in \{-(2^N-1), ..., -2, -, 0, , 2\alpha, ...(2^N-1)\}. \end{aligned} \tag{5.12}$$

When Q_l is fixed, Equation 5.11 is a convex function. And α_l is continuous. To optimize α_l, the optimality condition must be satisfied by

$$\sum_{i=1}^{c}\sum_{j=1}^{d}(2\alpha_l(q_l^{ij})^2 - 2v_l^{ij}q_l^{ij}) = 0. \tag{5.13}$$

Then we obtain the update of α_l

$$\alpha_l \leftarrow \frac{\sum_{i=1}^{c}\sum_{j=1}^{d} v_l^{ij}q_l^{ij}}{\sum_{i=1}^{c}\sum_{j=1}^{d}(q_l^{ij})^2}. \tag{5.14}$$

Secondly, we fix α_l. All entries of Q_l must be integers, the objective function of the sub-problem for updating Q_l is not differentiable. Therefore, the optimality condition can not be used. The optimal Q_l can be obtained by projection of $\frac{V_l}{\alpha_l}$

$$Q_l = \Pi_{\{0,\pm1,\pm2,...,\pm2^N-1\}}(\frac{V_l}{\alpha_l}), \tag{5.15}$$

where Π represents the projection operator. The projection onto a discrete set corresponds to finding the point within the set that is nearest to the given point.

Algorithm 8 shows the pseudocode of updating G_l. We adopt the early stop to stop the loop.

Algorithm 8 The Algorithm to update Q_l and α_l

Ensure: Q_l, α_l
Require: W_l, λ, ρ
  Initialize Q_l and α_l, maximum_patience , best_loss $= \infty$, patience $= 0$
  $V_l = W_l + \frac{\lambda}{\rho}$
  while the maximum iteration is not reached **do**
    $\alpha_l \leftarrow \frac{\sum_{i=1}^{c}\sum_{j=1}^{d} v_l^{ij} q_l^{ij}}{\sum_{i=1}^{c}\sum_{j=1}^{d} (q_l^{ij})^2}$
    $Q_l = \Pi_{\{0,\pm1,\pm2,\ldots,\pm 2^N-1\}} \frac{V_l}{\alpha_l}$
    if $\|V_l - \alpha_l Q_l\|_2^2 <$ best_loss **then**
      best_loss $= \|V_l - \alpha_l Q_l\|_2^2$
      $patience = 0$
    else
      $patience+ = 1$
    end if
    if $patience \geq$ maximium_patience **then**
      break
    end if
  end while
  $G_l = \alpha_l Q_l$
  Output G_l

5.1.3 Update λ

After updating W_l and G_l, the λ is updated using following rule:

$$\lambda \leftarrow \lambda + \rho(W_l - G_l). \tag{5.16}$$

5.2 Remaining non-quantized layers update

As shown in Figure 5.2, after the l-th layer is quantized, we obtain the network where the previous l layers are quantized while the remaining ones are not. Next, we need to retrain the non-quantized layers of the neural network. Therefore, the parameters of the remaining non-quantized layers can be adjusted to accommodate the changes in the quantized layers.
The loss function is still the function $R(\cdot)$. The input is quantized parameters $\overline{W}_{[1,2,\ldots,l]}$. The objective function is given by

$$\underset{W_{[l+1,\ldots,L]}}{\text{minimize}}\, R(\overline{W}_{[1,2,\ldots,l]}, W_{[l+1,\ldots,L]}). \tag{5.17}$$

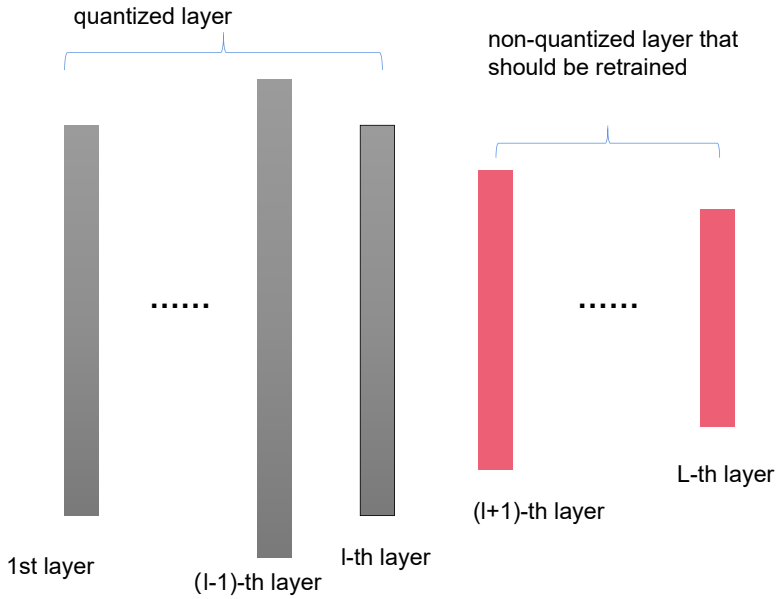

Figure 5.2: A L-layer neural network with l quantized layers and the remaining non-quantized layers will be retrained.

6. Evaluation

6.1 Experiment Setup

In this section, we present experimental results that compare the performance of two ADMM approaches to other approaches.

6.1.1 Dataset

In our experiment, we use 5 datasets to evaluate the performance of different approaches. Below, we describe these five datasets.

The first dataset is Seeds. The dataset Seeds holds data on area, perimeter, compaction, seed length, seed width, asymmetry coefficient, seed ventral groove length, and category data for different varieties of wheat seeds. The dataset has a total of 210 records, 7 features, and one label, and the labels are divided into 3 categories.

The second dataset is Pendigits. The dataset Pendigits is a multiclass classification dataset with 16 integer attributes and 10 classes. The digit database is created by collecting 250 samples from 44 writers. The samples written by 30 writers are used for training, cross-validation, and writer-dependent testing, and the digits written by the other 14 are used for writer-independent testing. In this dataset, all classes have equal frequencies.

The third dataset is MNIST. MNIST is a dataset of images of handwritten digits that counts a total of 250 images of handwritten digits from 250 different people. It comprises approximately 70,000 greyscale images of handwritten digits, each measuring 28×28 pixels. The images represent the digits 0 to 9. The dataset is divided into 60,000 training samples and 10,000 testing samples.

The fourth dataset is CIFAR10. The dataset CIFAR10 is a widely used benchmark dataset in the field of computer vision and machine learning. It is composed of a collection of 60,000 color images belonging to 10 different classes. Each image in the dataset CIFAR10 is a 32×32-pixel RGB image, which means it has three color channels (red, green, and blue). The dataset is divided into 50,000 training images and 10,000 testing images. The training set is further subdivided into five equal parts, known as “training batches”, each containing 10,000 images.

The fifth dataset is CIFAR100. The dataset CIFAR100 is an extension of the dataset CIFAR10 and is designed to be even more challenging. It contains 60,000 color images divided into 100 fine-grained classes, with 600 images per class. Each image in CIFAR100 is a 32×32-pixel RGB image, similar to CIFAR10. The dataset is split into 50,000 training images and 10,000 testing images. Like CIFAR10, the training set is further divided into five training batches, each containing 10,000 images.

Table 6.1 shows the number of features (input neurons in neural networks) and classes (output neurons) of different datasets.

dataset	the number of features and classes
Seeds	7 features and 3 classes
Pendigits	16 features and 10 classes
MNIST	28×28 features and 10 classes
CIFAR10	32×32×3 features and 10 classes
CIFAR100	32×32×3 features and 10 classes

Table 6.1: Table of datasets

6.1.2 Experiment Settings

In this experiment, five benchmark datasets were used for performance evaluation: Seeds, Pendigits, MNIST, CIFAR10, and CIFAR100. We design two pipelines for our experiments: Multilayer Perceptron (MLP) and convolutional neural network (CNN). Datasets Seeds, Pendigits, and MNIST are used for MLP pipeline, while CIFAR10 and CIFAR100 are used for CNN pipeline.

6.1.2.1 MLP pipeline

For different datasets, we design the models with different topologies. For example, for Seeds, the model has 3 layers. The input layer has 7 neurons. Each hidden layer has 5 neurons. The output layer has 3 neurons. Table 6.2 shows the topologies of models in MLP pipeline.

dataset	topology
Seeds	[7,5,5,3]
Pendigits	[16,12,12,10]
MNIST	[784,200,200,10]

Table 6.2: Table of datasets and their topology of models for MLP

In the MLP pipeline, we compare our two methods:
(1) training quantized neural network with ADMM approach (QNN-ADMM)
(2) layerwise quantization for the neural network using ADMM approach (LQ)
with the methods "Post training quantization" (PTQ) and "training quantized neural network with Straight through estimator" (QNN-STE). We compare their accuracies in the cases of 8 bits, 4 bits, 2 bits, and 1 bit, respectively.

PyTorch int8 neural network is the real-world state-of-art of model quantization. PyTorch is a popular deep learning framework. It supports the training and deployment

of neural networks using int8 data types (8-bit integers). PyTorch offers three methods: Dynamic Quantization, Static Quantization, and Quantization aware training. As Dynamic Quantization only support the MLP. So, We compare our methods with Dynamic quantization (DQ) and Quantization aware training (QAT) in MLP pipeline. Static Quantization supports the CNN. Therefore, the Static Quantization and Quantization aware training are chosen as the comparison methods. Note that we are only comparing the 8-bit case here.

6.1.2.2 CNN pipeline

In the CNN pipeline, we choose ResNet20 as the model. The experimental setup is similar to the MLP pipeline. But there is one more comparison method: Deep Neural Network Quantization via Layer-wise Optimization Using Limited Training Data (LDNQ) [10]. LDNQ also uses the ADMM approach to quantize the neural network layer by layer. However, the ground truth is not the label in the dataset but the pre-trained model. As the QNN-ADMM is only available for MLPs, we do not compare it with other methods in the CNN pipeline.

6.2 Experiment Result

6.2.1 MLP pipeline

6.2.1.1 Dataset Seeds

First, we obtain the results on small dataset: Seeds. Table 6.3 shows the accuracies of different methods, when the model is respectively quantized to 8 bits, 4 bits, 2 bits, and 1 bit.

We can observe that QNN-ADMM performs worse than other methods in the case of 8 bits. And QNN-ADMM has a large standard deviation in accuracy at 8 bits. But in the case of 2 bits, QNN-ADMM does not have a significant loss of precision. In the case of 1 bit, QNN-ADMM performs best among all methods. LQ performs well in the case of 8 bits, 4 bits and 2 bits. At 1 bit, LQ is second only to ADMM.

We then show the box plot respectively at 8 bits, 4 bits, 2 bits, and 1 bit. There are five most important values in the box plot. They are median, maximum, minimum, upper quartile and lower quartile. The black horizontal line in the graph represents the median. The two light blue horizontal lines represent the maximum and minimum values. The top and bottom edges of the coloured blocks represent the upper and lower quartiles. For all box plots in MLP pipeline, the red block denotes Post training Quantization. The yellow block denotes training quantized neural network with Straight through estimator. The blue block denotes training quantized neural network with ADMM. The dark green block denotes the layerwise quantization with ADMM. The pink block denotes Dynamic quantization. The orange block denotes Quantization aware training.

full precision	93.25% ± 2.84%					
method / nbits	PTQ	QNN-STE	QNN-ADMM	LQ	DQ	QAT
8 bits	93.25% ±2.84%	93.25% ±2.84%	82.56% ±9.32%	93.72% ±2.34%	93.72% ±2.95%	94.65% ±2.95%
4 bits	86.74% ±4.03%	89.77% ±1.86%	91.16% ±3.72%%	92.56% ±3.09%	-	-
2 bits	63.26% ±22.49%	87.67% ±4.17%	88.61% ±3.95%	89.07% ±4.99%	-	-
1 bit	33.52% ±13.36%	38.14% ±18.70%	66.51% ±18.49%	55.81% ±22.09%	-	-

Table 6.3: Table of accuracies among PTQ, QNN-STE, QNN-ADMM, LQ, DQ, QAT, on dataset Seeds in the case of 8 bits, 4 bits, 2 bits and 1 bit. PTQ is the abbreviation of Post training quantization. QNN-STE is the abbreviation of training quantized neural network with Straight through estimator. QNN-ADMM is the abbreviation of training quantized neural network with ADMM. LQ is the abbreviation of layerwise quantization using ADMM. DQ is the abbreviation of Dynamic Quantization. QAT is the abbreviation of Quantization aware training.

Figure 6.1 shows the accuracies of the quantized neural network of different methods at 8 bits. We can observe that the result of QNN-ADMM is obviously worse than other methods.

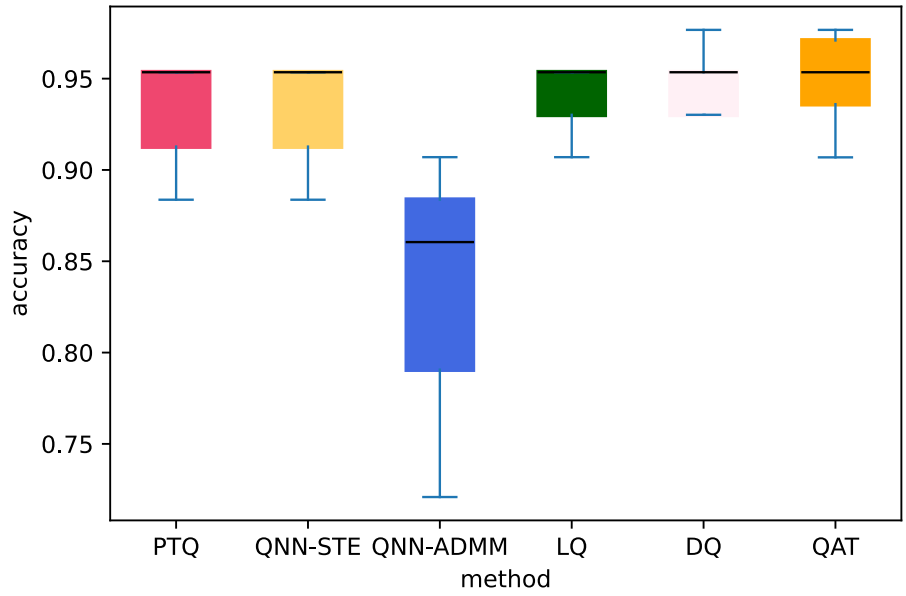

Figure 6.1: The performance of different methods at 8 bits on dataset Seeds.

Figure 6.2 shows the accuracies of the quantized neural network of different methods at 4 bits. LQ performs the best among all methods.

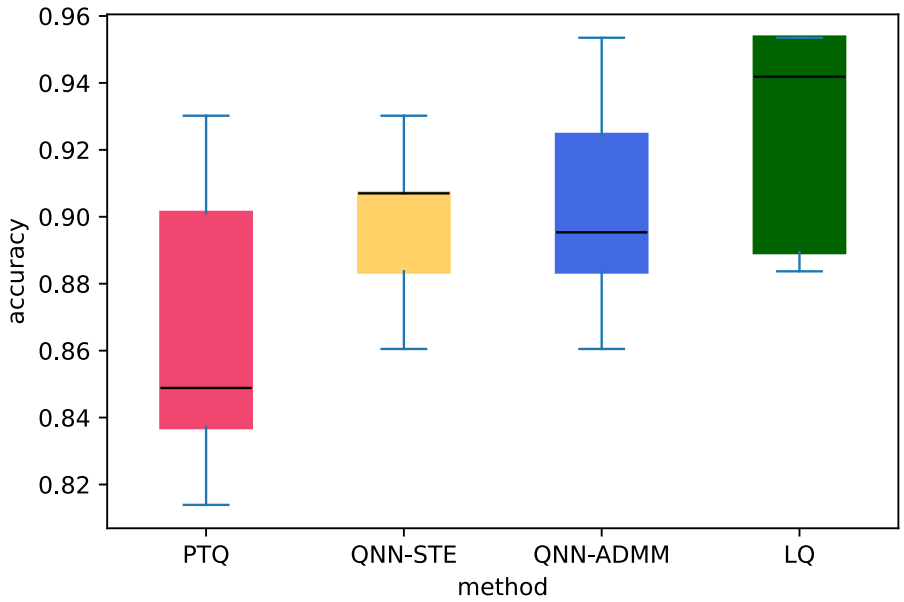

Figure 6.2: The performance of different methods at 4 bits on dataset Seeds.

Figure 6.3 shows the accuracies of the quantized neural network of different methods at 2 bits. Except for PTQ, all other three methods perform well.

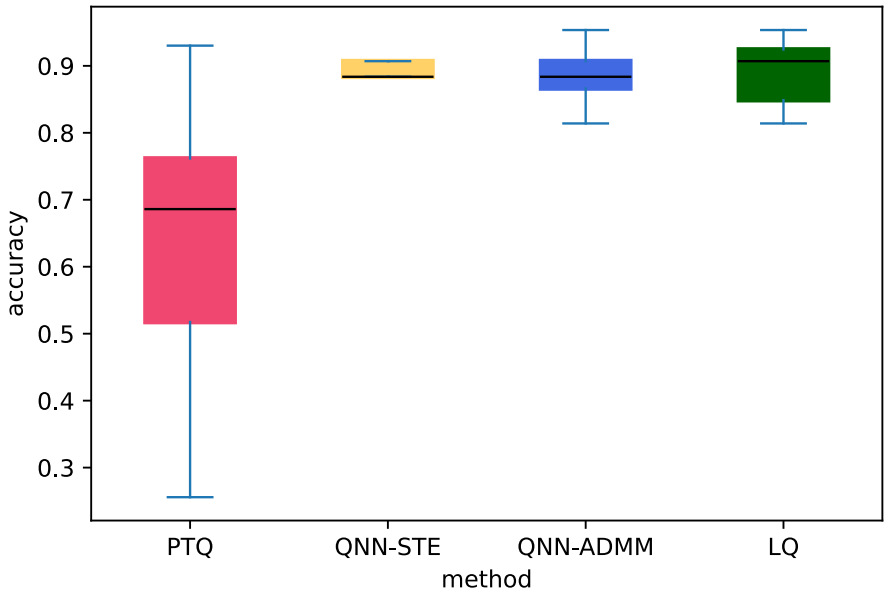

Figure 6.3: The performance of different methods at 2 bits on dataset Seeds.

Figure 6.4 shows the accuracies of the quantized neural network of different methods at 1 bit. QNN-ADMM and LQ perform much better than the other two methods.

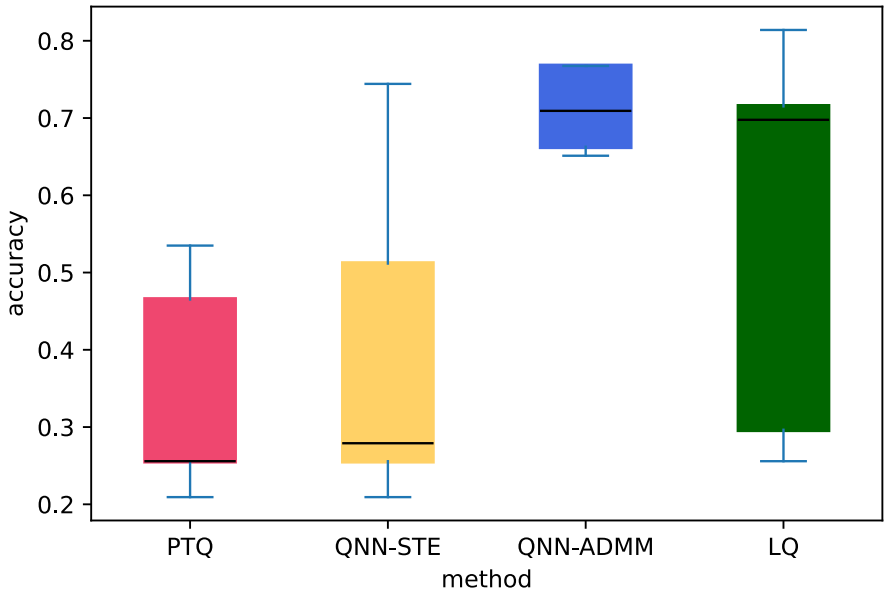

Figure 6.4: The performance of different methods at 1 bit on dataset Seeds.

6.2.1.2 Dataset Pendigits

Second, we obtain the results on datasets Pendigits. Table 6.4 shows the accuracies of different methods, when the model is respectively quantized to 8 bits, 4 bits, 2 bits, and 1 bit. We can observe that QNN-ADMM has a significant loss of accuracy at 8 bits and 4 bits, but performs best at 2 bits and 1 bit. The accuracy of LQ is second to QNN-ADMM at 2 bits and 1 bit. And At 8 bits, the average accuracy of LQ is just a little lower than QAT, but it has a smaller standard deviation. This indicates that LQ performs more stalely.

Figure 6.5 shows the accuracies of the quantized neural network of different methods at 8 bits on dataset Pendigits. Almost all the methods have accuracies around 98% except QNN-ADMM. QNN-ADMM performs the worst and has a wide range of variations in accuracy.

full precision accuracy	98.32% ± 0.25%					
nbits \ method	PTQ	QNN-STE	QNN-ADMM	LQ	DQ	QAT
8 bits	97.99% ±0.36%	97.89% ±0.43%	85.01% ±1.88%	98.15% ±0.2%	98.07% ±0.23%	98.17% ±2.92%
4 bits	67.04% ±14.21%	92.76% ±1.68%	81.93% ±4.33%%	91.54% ±4.29%	-	-
2 bits	25.17% ±8.60%	45.80% ±14.65%	53.32% ±11.62%	53.08% ±8.79%	-	-
1 bit	9.89% ±0.25%	16.52% ±4.32%	35.73% ±9.21%	31.12% ±10.12%	-	-

Table 6.4: Table of the accuracies on dataset Pendigits in the case of 8 bits, 4 bits, 2 bits, 1 bit. PTQ is the abbreviation of Post training quantization. QNN-STE is the abbreviation of training quantized neural network with Straight through estimator. QNN-ADMM is the abbreviation of training quantized neural network with ADMM. LQ is the abbreviation of layerwise quantization using ADMM. DQ is the abbreviation of Dynamic Quantization. QAT is the abbreviation of Quantization aware training.

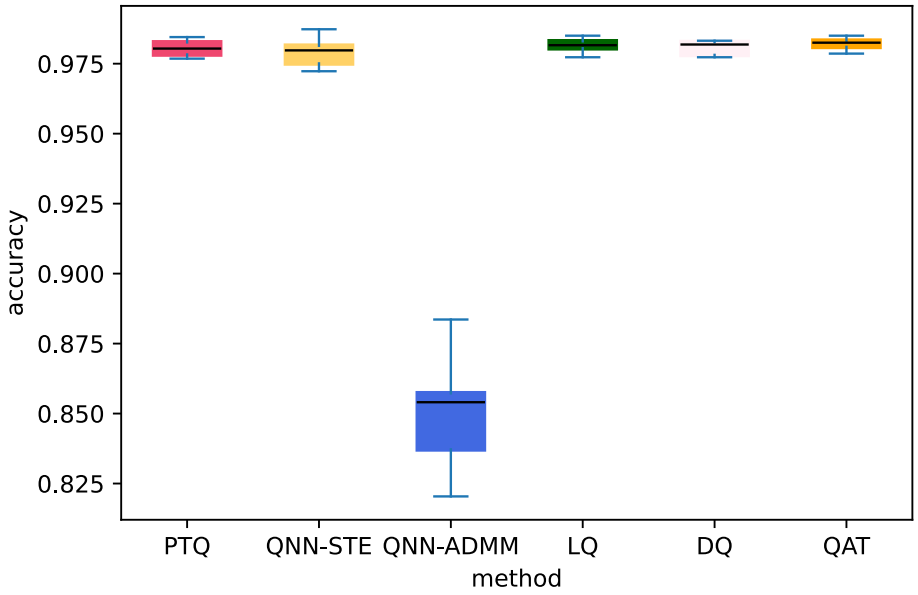

Figure 6.5: The performance of different methods at 8 bits on dataset Pendigits.

Figure 6.6 shows the accuracies of the quantized neural network of different methods at 4 bits on dataset Pendigits. The accuracy of QNN-STE has the highest median and a small range of variation. LQ also has high accuracy, but has a larger range of variation than QNN-STE.

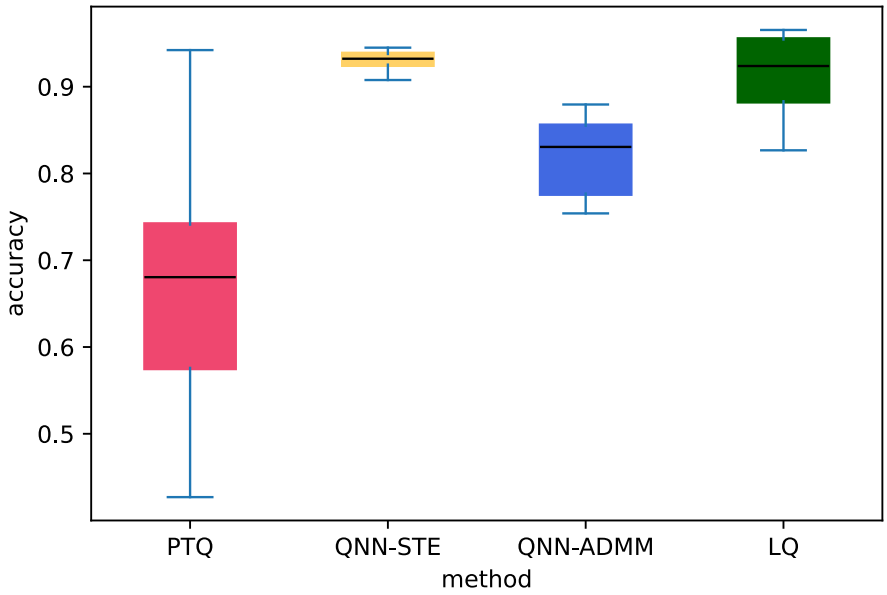

Figure 6.6: The performance of different methods at 4 bits based on dataset Pendigits.

Figure 6.7 shows the accuracies of the quantized neural networks among different methods at 2 bits on dataset Pendigits. QNN-ADMM and LQ perform better than PTQ and QNN-STE. Even though the median accuracy of LQ is lower than that of QNN-ADMM, LQ is more stable.

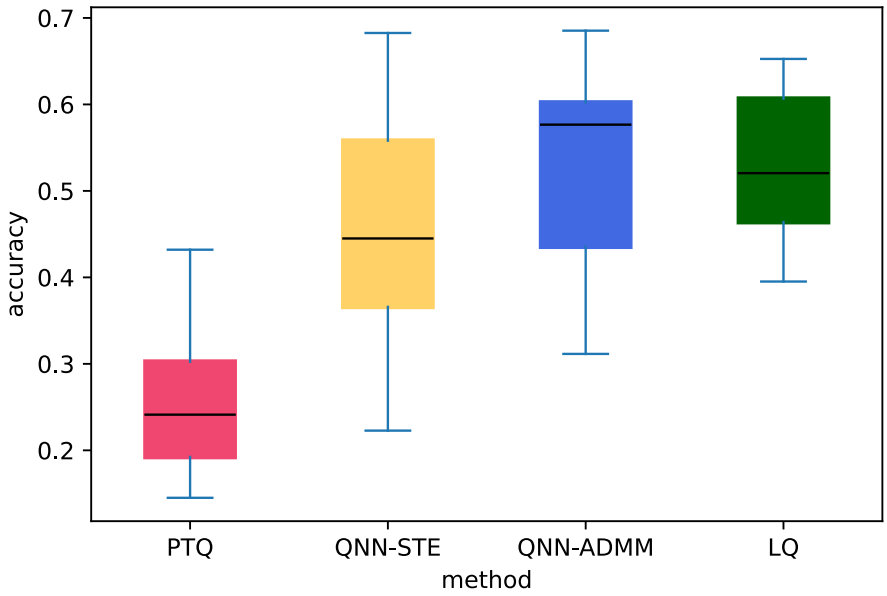

Figure 6.7: The performance of different methods at 2 bits on dataset Pendigits.

Figure 6.8 shows the accuracies of the quantized neural network of different methods at 1 bit on dataset Pendigits. QNN-ADMM and LQ perform much better than the

other two methods. Even though their accuracies have a wide range of variability, their worst accuracy is still better than the other methods.

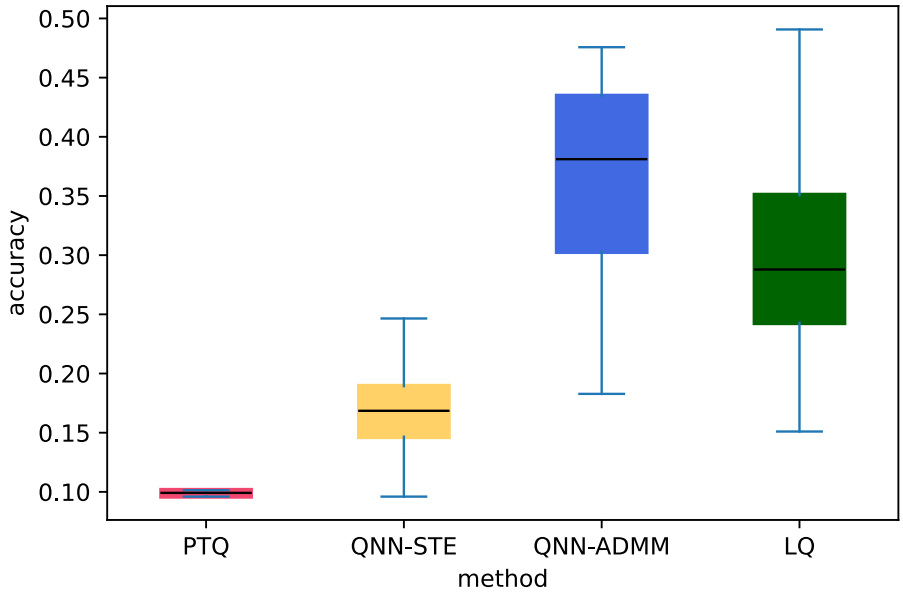

Figure 6.8: The performance of different methods that neural network at 1 bit on dataset Pendigits.

6.2.1.3 Dataset MNIST

The images of dataset MNIST have 28×28 pixels. We flatten these images into column vector which can be used as input for MLP. The results on dataset MNIST are shown in Table 6.5. LQ performs best at 8 bits, 4 bits, 2 bits. At 1 bit, LQ has high mean accuracy but also has a large standard deviation. It means it is not stable at 1 bit. The results of QNN-ADMM are similar to the other two datasets. It is obvious that its accuracies are lower than other methods at 8 bits. However, it exhibits excellent performance at 4 bits, 2 bits and 1 bit.

full precision accuracy	95.33% ± 0.27%					
method / nbits	PTQ	QNN-STE	QNN-ADMM	LQ	DQ	QAT
8 bits	94.31% ±1.32%	95.27% ±0.93%	86.28% ±0.74%	97.26% ±0.09%	97.22% ±0.08%	95.55% ±0.65%
4 bits	39.37% ±21.21%	85.99% ±1.3%	86.08% ±0.75%	97.09% ±0.16%	-	-
2 bits	10.53% ±1.11%	16.19% ±5.98%	81.26% ±2.06%	82.61% ±23.69%	-	-
1 bit	9.74% ±0.89%	10.65% ±1.58%	67.12% ±5.11%	23.07% ±23.62%	-	-

Table 6.5: Table of the accuracy of dataset MNIST in the case of 8 bits, 4 bits, 2 bits, 1 bit. PTQ is the abbreviation of Post training quantization. QNN-STE is the abbreviation of training quantized neural network with Straight through estimator. QNN-ADMM is the abbreviation of training quantized neural network with ADMM. LQ is the abbreviation of layerwise quantization using ADMM. DQ is the abbreviation of Dynamic Quantization. QAT is the abbreviation of Quantization aware training.

Figure 6.9 shows the accuracies of the quantized neural network of different methods at 8 bits in dataset MNIST. We can clearly observe that the accuracy of QNN-ADMM is far lower than other methods. Even the maximum accuracy of QNN-ADMM is still lower than other methods. And LQ performs best and consistently.

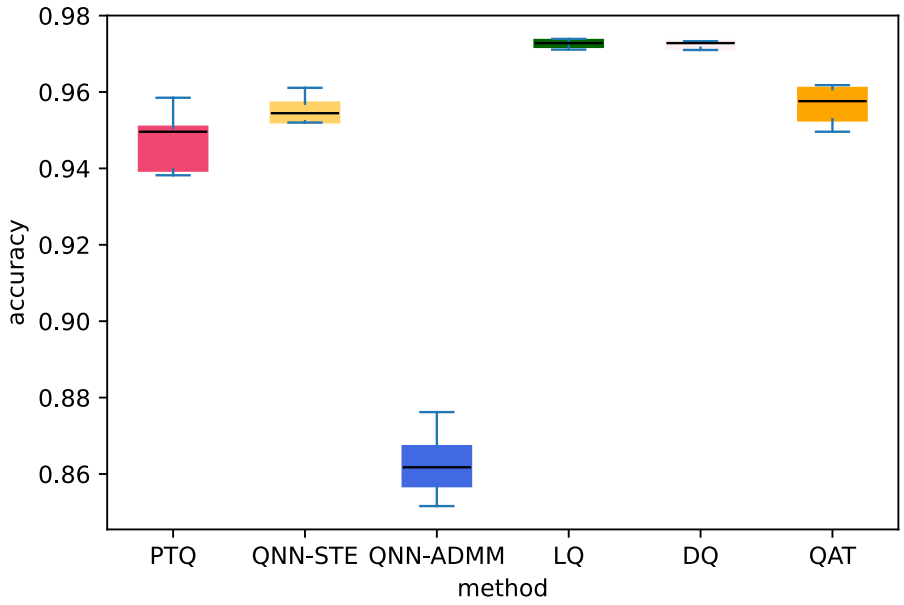

Figure 6.9: The performance of different methods at 8 bits on dataset MNIST.

Figure 6.10 shows the accuracies of the quantized neural network of different methods at 4 bits. LQ performs far better than other methods. The QNN-STE has higher medians than QNN-ADMM, but QNN-ADMM shows more stability.

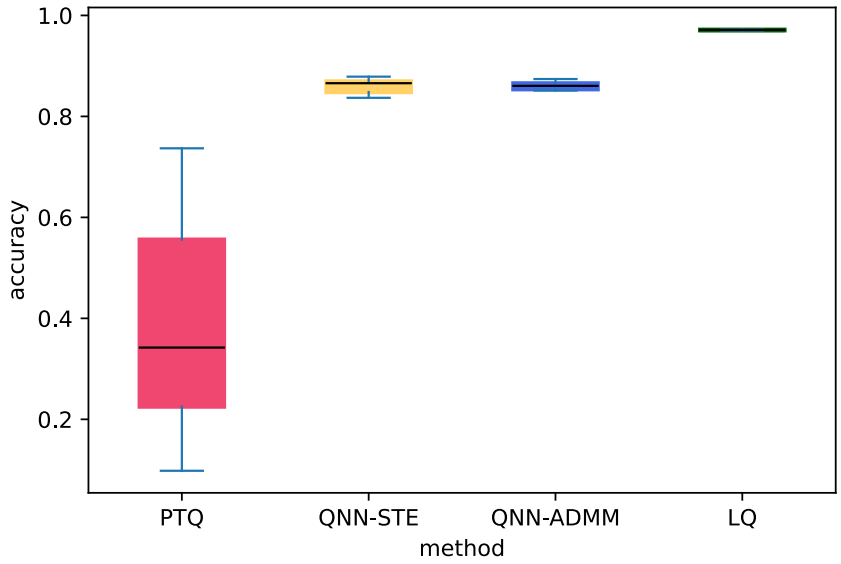

Figure 6.10: The performance of different methods at 4 bits on dataset MNIST.

Figure 6.11 shows the accuracies of the quantized neural network of different methods at 2 bits on dataset MNIST. The accuracy rank is LQ, QNN-ADMM, QNN-STE, PTQ.

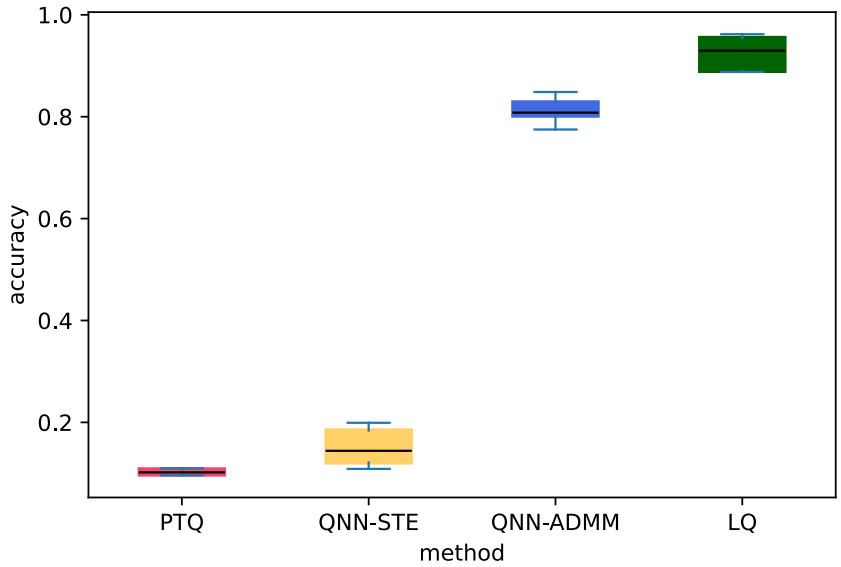

Figure 6.11: The performance of different methods at 2 bits on dataset MNIST.

Figure 6.12 shows the accuracies of the quantized neural network of different methods at 1 bit on dataset MNIST. The accuracy rank is QNN-ADMM, LQ, QNN-STE, PTQ.

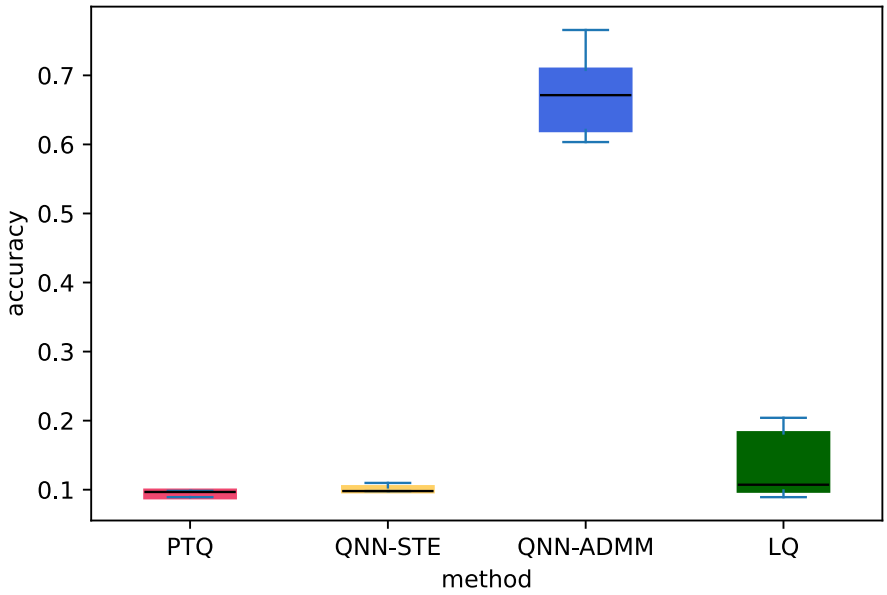

Figure 6.12: The performance of different methods that neural network at 1 bit based on dataset MNIST.

6.2.2 CNN pipeline

6.2.2.1 Dataset CIFAR10

Table 6.6 shows the results on dataset CIFAR10. We can observe that the LQ has the highest accuracy at 4 bits and second-highest accuracy at 8 bits and 2 bits. In particular, the accuracy of LQ improves by around 4% compared to the full-precision model in the 8-bit and 4-bit cases. Due to batch normalization, these methods still show good performance at 1 bit. But at 1 bit, the LQ does not perform as well as QNN-STE and LDNQ.

full precision accuracy	84.23% ± 1.01%					
nbits \ method	PTQ	QNN-STE	LDNQ	LQ	SQ	QAT
8 bits	84.27% ±0.99%	84.68% ±0.97%	86.10% ±0.37%	88.11% ±0.45%	84.40% ±0.73%	88.61% ±0.31%
4 bits	83.24% ±1.21%	84.03% ±1.18%	86.01% ±0.41%	88.05% ±0.38%	-	-
2 bits	44.19% ±4.75%	81.64% ±1.23%	85.75% ±0.36%	84.37% ±1.28%	-	-
1 bit	9.91% ±0.24%	80.56% ±1.09%	80.41% ±0.43%	74.20% ±1.77%	-	-

Table 6.6: Table of the accuracies on dataset CIFAR10 in the case of 8 bits, 4 bits, 2 bits, 1 bit. PTQ is the abbreviation of Post training quantization. QNN-STE is the abbreviation of training quantized neural network with Straight through estimator. LDNQ is the abbreviation of Deep Neural Network Quantization via Layer-wise Optimization Using Limited Training Data. LQ is the abbreviation of layerwise quantization using ADMM. SQ is the abbreviation of Static Quantization. QAT is the abbreviation of Quantization aware training.

Now, we show the results through box plots. For all box plots in CNN pipeline, the red block denotes Post training Quantization. The yellow block denotes training quantized neural network with Straight through estimator. The light green block denotes Deep Neural Network Quantization via Layer-wise Optimization Using Limited Training Data. The dark green block denotes the layerwise quantization with ADMM. The orange block denotes Static Quantization. The light red block denotes Quantization aware training.

Figure 6.13 shows the accuracies of the quantized neural network of different methods at 8 bits on dataset CIFAR10. We can clearly observe that the accuracy of LQ rank second at 8 bits but still much better than PTQ, QNN-STE, LDNQ.

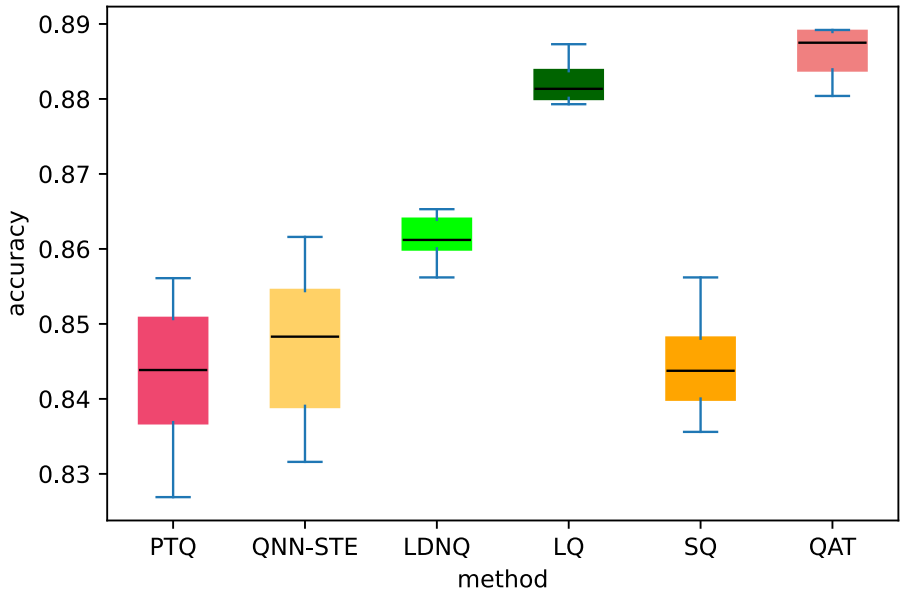

Figure 6.13: The performance of different methods at 8 bits on dataset CIFAR10.

Figure 6.14 shows the accuracies of the quantized neural network of different methods at 4 bits on dataset CIFAR10. The rank of accuracy of different methods is: LQ, LDNQ, QNN-STE, PTQ.

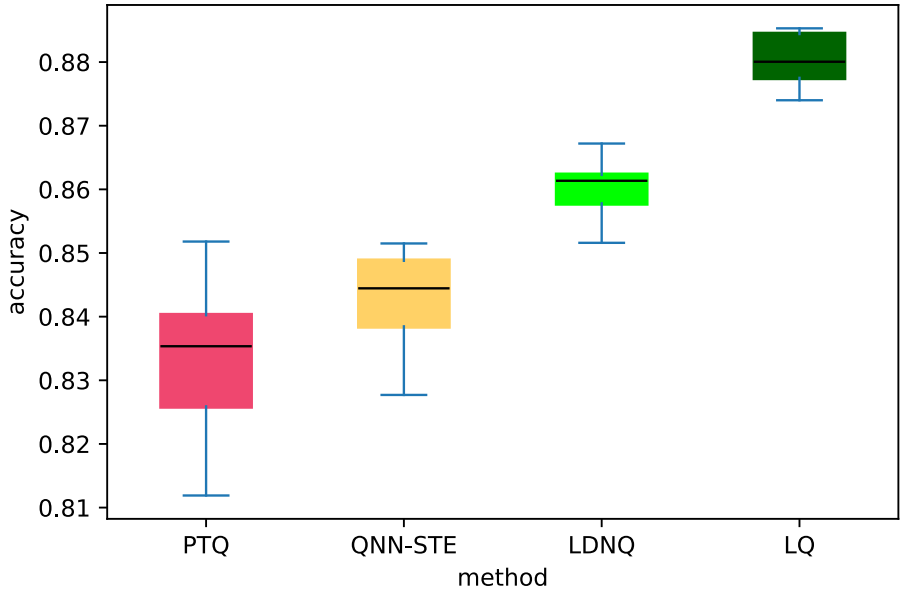

Figure 6.14: The performance of different methods at 4 bits on dataset CIFAR10.

Figure 6.15 shows the accuracies of the quantized neural network of different methods at 2 bits on dataset CIFAR10. We can see the accuracy of LQ is a little lower than LDNQ.

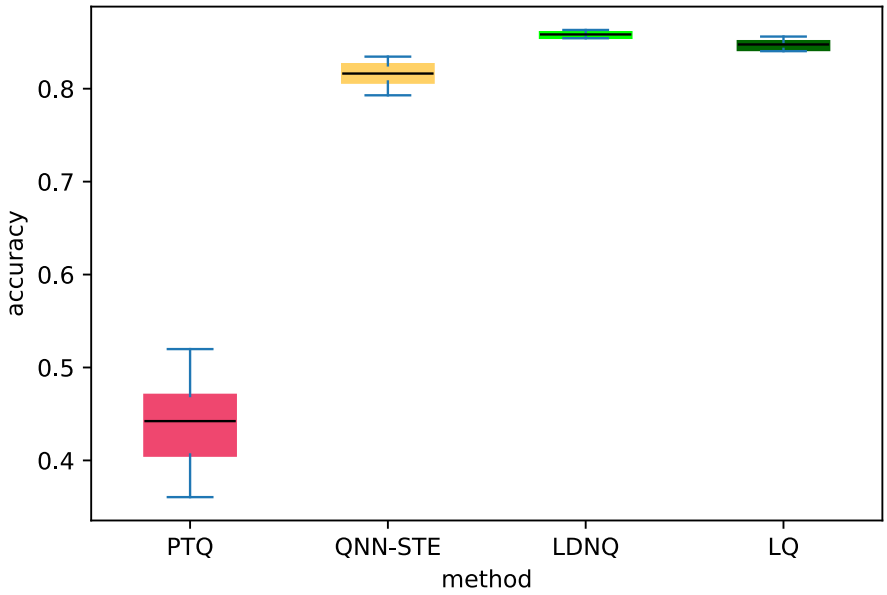

Figure 6.15: The performance of different methods at 2 bits on dataset CIFAR10.

Figure 6.14 shows the accuracies of the quantized neural network of different methods at 1 bit on dataset CIFAR10. LQ does not perform as well as QNN-STE and LDNQ.

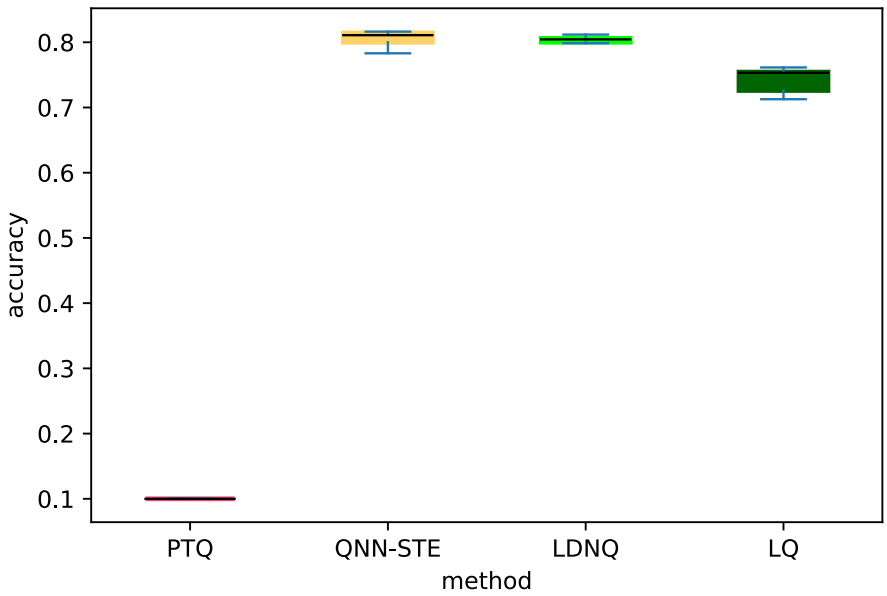

Figure 6.16: The performance of different methods at 1 bit on dataset CIFAR10.

6.2.2.2 Dataset CIFAR100

Table 6.7 shows the results of dataset CIFAR100. The regular is similar to CIFAR10. We can see that LQ has large accuracy gains at 8 bits and 4 bits compared to the full precision model. And LQ is second best at 4 bits and does not perform well at 1 bit.

full precision accuracy	52.28% ± 1.61%					
nbits \ method	PTQ	QNN-STE	LDNQ	LQ	SQ	QAT
8 bits	52.30% ±1.29%	52.10% ±0.76%	52.75% ±0.85%	59.57% ±0.78%	54.11% ±1.18%	59.22% ±0.86%
4 bits	49.67% ±1.01%	50.82% ±9.55%	52.70% ±0.81%	57.55% ±0.38%	-	-
2 bits	28.17% ±2.60%	41.60% ±1.29%	51.38% ±1.03%	46.94% ±1.79%	-	-
1 bit	1.09% ±0.27%	35.80% ±2.66%	32.88% ±1.61%	27.25% ±2.82%	-	-

Table 6.7: Table of the accuracies on dataset CIFAR100 in the case of 8 bits, 4 bits, 2 bits, 1 bit. QNN-STE is the abbreviation of training quantized neural network with Straight through estimator. LDNQ is the abbreviation of Deep Neural Network Quantization via Layer-wise Optimization Using Limited Training Data. LQ is the abbreviation of layerwise quantization using ADMM. SQ is the abbreviation of Static Quantization. QAT is the abbreviation of Quantization aware training.

Figure 6.17 shows the accuracies of the quantized neural network of different methods at 8 bits on dataset CIFAR100. LQ and QAT are better than others.

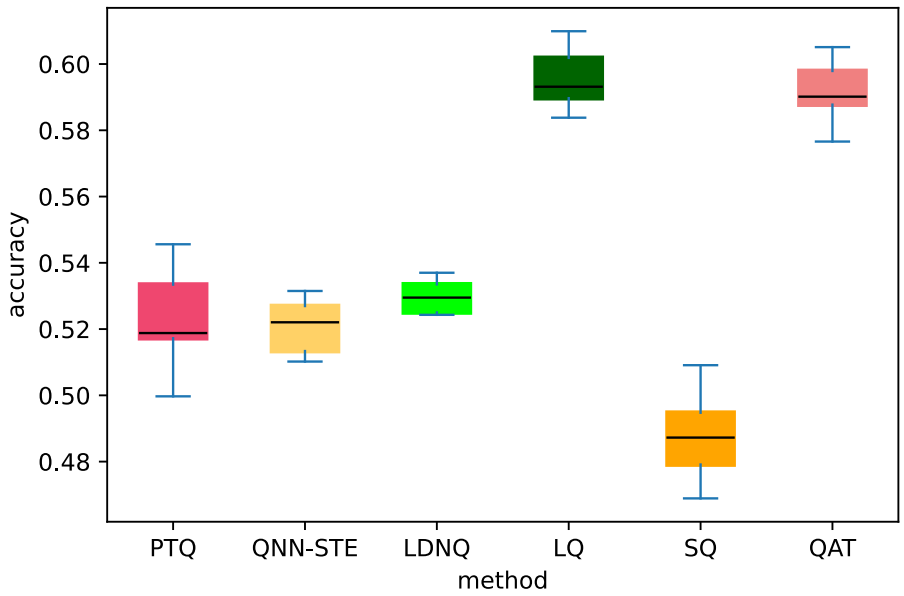

Figure 6.17: The performance of different methods at 8 bits on dataset CIFAR100.

Figure 6.18 shows the accuracies of the quantized neural network of different methods at 4 bits on dataset CIFAR100. The rank of the method is: LQ, LDNQ, QNN-STE, PTQ.

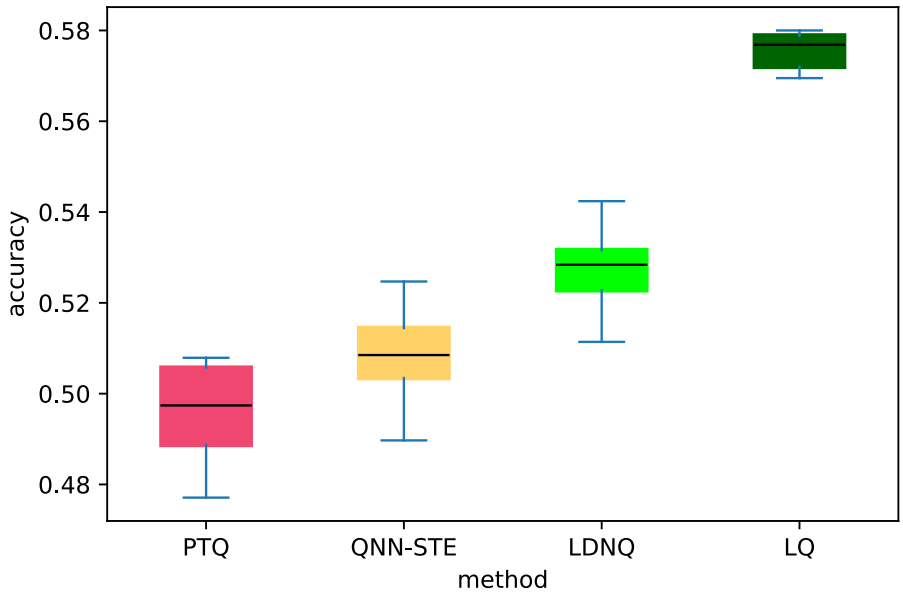

Figure 6.18: The performance of different methods at 4 bits on dataset CIFAR100.

Figure 6.19 shows the accuracies of the quantized neural network of different methods at 2 bits on dataset CIFAR100. LDNQ has the highest accuracy. And LQ has second-highest accuracy.

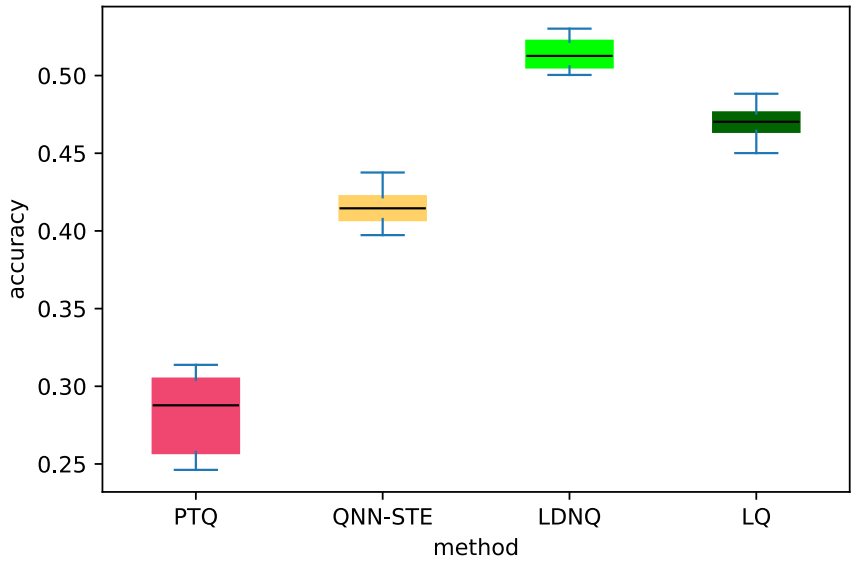

Figure 6.19: The performance of different methods at 2 bits based on dataset CIFAR100.

Figure 6.20 shows the accuracies of the quantized neural network of different methods at 1 bit on dataset CIFAR100. We can observe that QNN-STE performs best among all methods at 1 bit.

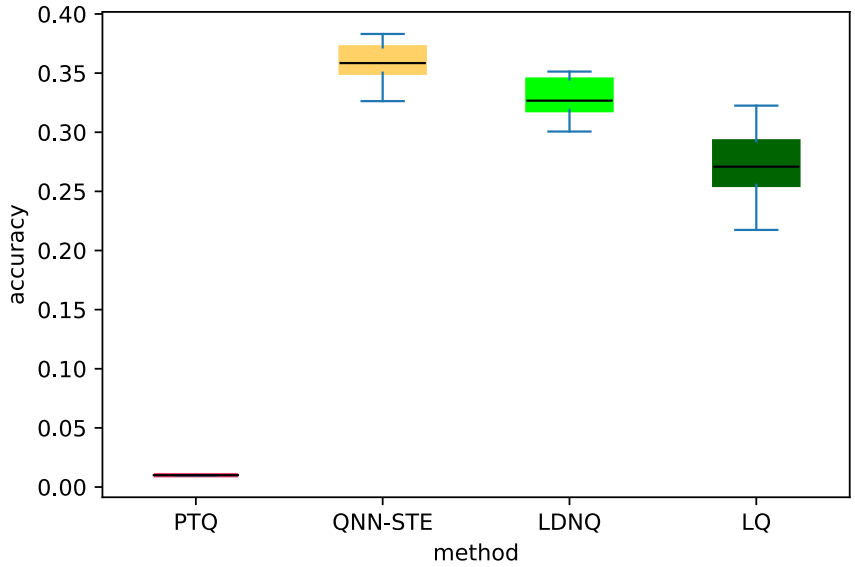

Figure 6.20: The performance of different methods at 1 bit on dataset CIFAR100.

6.3 Discussion

We can find a regular about QNN-ADMM: QNN-ADMM always performs poorly at 8 bits and 4 bits. But it always performs well at 2 bits and 1 bit. QNN-ADMM only adds a Lagrange multiplier to the sub-problem of the neural network's output z_L. There are no Lagrange multipliers on the sub-problems of other parameters. The constraints of these sub-problems are not strictly fulfilled. So the obtained minimum of the sub-problems without adding Lagrange multipliers are not the exact solution. This means that there are errors in each layer of the neural network. When these errors are accumulated, the accuracy becomes lower. However, it will become unstable if Lagrange multipliers are added to all sub-problems. QNN-STE performs well at 8 bits and 4 bits, as it approximates the gradient well. However, the difference between the approximated backward gradient using QNN-STE and the true backward gradient is much larger at 2 bits and 1 bit, so QNN-STE does not perform well at 2 bits and 1 bit. Although in the QNN-ADMM approach, there are errors in each layer, which leads to a loss of accuracy. However, it does not require gradient in the optimization process, which makes it show good performance. In MLP pipelines, we found LQ performs well on 3 datasets and at four kinds of bits. In CNN pipeline, the performance of LQ ranks top 2 except the 1-bit case. We found that LQ is more suitable for MLP. We also observe, whether in the CNN pipeline or the MLP pipeline, the accuracy of the quantized model sometimes increases compared to the full precision model at 8 bits, instead of decreasing. We guess that LQ can help the model to escape from the local minimum and exhibit improved performance.

7. Conclusion and Future Work

In this thesis, we focus on quantization of neural network to reduce the memory footprint and computational complexity of deep learning models while maintaining their performance.

We propose to quantize the neural network using ADMM approach. In Chapter 3 and Chapter 4 we present a method to train quantized neural networks without gradient. Borrowing the idea from ADMM, the training process of a neural network can be decomposed into the update process of each parameter within the neural network. By optimizing each parameter under its relevant constraints, the quantized neural network can be trained without gradients. In Chapter 5, we propose another method to quantize a pre-trained neural network layerwisely with ADMM approach. After each layer is quantized, the remaining non-quantized layers must be updated to compensate the loss of accuracy that the quantization results in.

In Chapter 6, we compare the mean value and standard variation of accuracy among different methods at different bits on 5 datasets. we found, that the model has higher accuracy at 2 bits and 1 bit when the quantized network is trained using ADMM approach. The method, layerwise quantization with ADMM approach, has higher accuracy compared to other comparison methods. And it is more suitable to Multilayer Perceptron than convolutional neural networks.

Bibliography

[1] https://mlnotebook.github.io/post/CNN1/.

[2] Sungsoo Ahn et al. "Variational Information Distillation for Knowledge Transfer". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 9163–9171.

[3] Yali Amit, Pedro Felzenszwalb, and Ross Girshick. "Object Detection". In: *Computer Vision: A Reference Guide* (2020), pp. 1–9.

[4] Yu Bai, Yu-Xiang Wang, and Edo Liberty. "Proxquant: Quantized Neural Networks via Proximal Operators". In: *arXiv preprint arXiv:1810.00861* (2018).

[5] Ron Banner, Yury Nahshan, and Daniel Soudry. "Post Training 4-Bit Quantization of Convolutional Networks for Rapid-deployment". In: *Advances in Neural Information Processing Systems* 32 (2019).

[6] Amir Beck and Marc Teboulle. "A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems". In: *SIAM journal on imaging sciences* 2.1 (2009), pp. 183–202.

[7] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. "Estimating or Propagating Gradients through Stochastic Neurons for Conditional Computation". In: *arXiv preprint arXiv:1308.3432* (2013).

[8] Stephen Boyd et al. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends® in Machine learning* 3.1 (2011), pp. 1–122.

[9] Rishidev Chaudhuri and Ila Fiete. "Computational Principles of Memory". In: *Nature neuroscience* 19.3 (2016), pp. 394–403.

[10] Shangyu Chen, Wenya Wang, and Sinno Jialin Pan. "Deep Neural Network Quantization via Layer-wise Optimization Using Limited Training Data". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 3329–3336.

[11] Dan Claudiu Ciresan et al. "Flexible, High Performance Convolutional Neural Networks for Image Classification". In: *Twenty-second international joint conference on artificial intelligence*. Citeseer. 2011.

[12] Ronan Collobert and Jason Weston. "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning". In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 160–167.

[13] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training Deep Neural Networks with Binary Weights during Propagations". In: *Advances in neural information processing systems* 28 (2015).

[14] Amir Gholami et al. "A Survey of Quantization Methods for Efficient Neural Network Inference". In: *arXiv preprint arXiv:2103.13630* (2021).

[15] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[16] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the Knowledge in a Neural Network". In: *arXiv preprint arXiv:1503.02531* (2015).

[17] Tianjian Huang et al. "Alternating Direction Method of Multipliers for Quantization". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2021, pp. 208–216.

[18] David H Hubel and Torsten N Wiesel. "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex". In: *The Journal of physiology* 160.1 (1962), p. 106.

[19] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-arithmetic-only Inference". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[20] Mel Win Khaw, Luminita Stevens, and Michael Woodford. "Discrete Adjustment to a Changing Environment: Experimental Evidence". In: *Journal of Monetary Economics* 91 (2017), pp. 88–103.

[21] Raghuraman Krishnamoorthi. "Quantizing Deep Convolutional Networks for Efficient Inference: A Whitepaper". In: *arXiv preprint arXiv:1806.08342* (2018).

[22] Yann LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural computation* 1.4 (1989), pp. 541–551.

[23] Dengsheng Lu and Qihao Weng. "A Survey of Image Classification Methods and Techniques for Improving Classification Performance". In: *International journal of Remote sensing* 28.5 (2007), pp. 823–870.

[24] Warren S McCulloch and Walter Pitts. "A Logical Calculus of the Ideas Immanent in Nervous Activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[25] Shervin Minaee et al. "Image Segmentation Using Deep Learning: A Survey". In: *IEEE transactions on pattern analysis and machine intelligence* 44.7 (2021), pp. 3523–3542.

[26] Leonardo Noriega. "Multilayer Perceptron Tutorial". In: *School of Computing. Staffordshire University* 4.5 (2005), p. 444.

[27] Antonio Polino, Razvan Pascanu, and Dan Alistarh. "Model Compression via Distillation and Quantization". In: *arXiv preprint arXiv:1802.05668* (2018).

[28] Russell Reed. "Pruning Algorithms-a Survey". In: *IEEE transactions on Neural Networks* 4.5 (1993), pp. 740–747.

[29] Gavin Taylor et al. "Training Neural Networks Without Gradients: A Scalable ADMM Approach". In: *International conference on machine learning*. PMLR. 2016, pp. 2722–2731.

[30] Junxiang Wang et al. “ADMM for Efficient Deep Learning with Global Convergence”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 111–119.

[31] Hao Wu et al. “Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation”. In: *arXiv preprint arXiv:2004.09602* (2020).

[32] Penghang Yin et al. “Understanding Straight-through Estimator in Training Activation Quantized Neural Nets”. In: *arXiv preprint arXiv:1903.05662* (2019).