CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Hybrid Compression: Exploiting Model and Data Compression for Deep Neural Network Workloads

Master's thesis in Computer science and engineering

Yunyao Xie

Naicheng Li

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG

Gothenburg, Sweden 2022

Master's thesis 2022

Hybrid Compression: Exploiting Model and Data Compression for Deep Neural Network Workloads

Yunyao Xie
Naicheng Li

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Hybrid Compression: Exploiting Data and Model Compression for Deep Neural Network Workloads

Yunyao Xie
Naicheng Li

© Yunyao Xie, Naicheng Li, 2022.

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Examiner: Miquel Pericas, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2022

Hybrid Compression: Exploiting Data and Model Compression for Deep Neural Network Workloads

Yunyao Xie
Naicheng Li
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Nowadays, various kinds of deep neural networks (DNNs) show human-level capabilities in their domain. But the networks usually have millions or billions of parameters and significant computation cost. To bring the artificial intelligence into people's daily lives, deploying DNNs efficiently on various hardware (e.g., resource-limited edge devices) has become a popular topic. In our thesis work, we explore the model compression and data compression and then combine them to build hybrid compression to reduce the computation cost, memory cost of DNNs and speed up the model inference speed.
The hybrid compression gives us compression ratios of 5.15 and 5.57, speedup of 2.66 and 3.93, and accuracy losses of 2.38% and 2.64%, if we start from pruning 30% of floating point operations (FLOPs) of MobileNetV1 and ResNet50 respectively. Starting from models which are pruned 50% FLOPs, hybrid compression can achieve compression ratio of 6.29 and 7.53, speedup of 2.74 and 4.21 and with accuracy drop of 7.71% and 9.27% for MobileNetV1 and ResNet50. To verify the effectiveness of hybrid compression, we evaluate the inferece speed of compressed model on two edge devices, Nvidia Jetson Nano and Nvidia Jetson Xavier NX. We find that the gains of hybrid compression are hardware related and NX shows more impressive strengths than Nano. At last, we give users recommendations about how to apply the hybrid compression from different aspects.

Keywords: Deep Neural Networks, CNNs, Model Compression, Data Compression, Model Inference.

Acknowledgements

We are pleased and grateful to participate in this interesting and practical thesis project. We want to thank our supervisor Prof. Pedro Petersen Moura Trancoso, who always gives us creative ideas and clear suggestions, and helps us along the way to overcome various difficulties. We also thank our examiner Assoc. Prof. Miquel Pericas, who gives us warm advice on the half and final presentation. We thank all of people who have helped us to finish this great master thesis work.

Yunyao Xie, Naicheng Li, Gothenburg, June 2022

Contents

List of Figures **xi**

List of Tables **xv**

1 Introduction **1**
1.1 Overview of DNNs 1
1.2 Model Compression 3
1.2.1 Pruning 3
1.2.2 Quantization 4
1.3 Data Compression 5
1.4 Contribution 6

2 Theory **9**
2.1 Deep Neural Networks 9
2.1.1 Structure of Neural Network 9
2.1.2 Structure of Convolutional neural networks 11
2.2 Model Compression 14
2.2.1 Pruning 14
2.2.1.1 Filter-Wise Pruning 15
2.2.1.2 Select by L1-norm 16
2.2.1.3 Select by Geometric Median 16
2.2.2 Quantization 17
2.2.2.1 Fundamental of Quantization 17
2.2.2.2 Quantization Parameters 18
2.3 Data Compression 19
2.3.1 Lossless Compression 19
2.3.2 Lossy Compression 22

3 Methods **25**
3.1 Model Compression 25
3.1.1 AI framework 25
3.1.2 Pruning 26
3.1.3 Quantization 29
3.1.3.1 Quantization Aware Training 29
3.1.3.2 Post Training Quantization 31
3.1.4 Model Compression Pipeline 33
3.2 Data Compression 34

3.2.1 Lossless compression of integer weights 36
3.2.2 Lossy compression with floating point weights 37
3.3 Hybrid Compression 39

4 Results 41
4.1 Experimental Setup 41
4.1.1 Experimental Models 41
4.1.2 Platform 41
4.1.3 Dataset 42
4.1.4 Training Settings 42
4.1.5 Metrics 42
4.2 Model Compression 43
4.2.1 Pruning 43
4.2.2 Quantization 46
4.2.3 Model Compression Pipeline 47
4.3 Data Compression 50
4.3.1 INT8 Part Parameter Compression 51
4.3.2 FP32 Part Parameter Compression 53
4.4 Hybrid Compression 55
4.5 Edge Device 57
4.5.1 NVIDIA Jetson Nano 57
4.5.1.1 Hardware Introduction 57
4.5.1.2 Results 57
4.5.2 Nvidia Jetson Xavier NX 59
4.5.2.1 Hardware Introduction 59
4.5.2.2 Results 60
4.5.3 Nano Vs NX 61

5 Analysis 63
5.1 Time Overhead 63
5.1.1 Model Retraining 63
5.1.2 Sensitivity Analysis 63
5.1.3 Others 63
5.2 Network Transmission 64
5.3 Energy 65
5.4 Deploy On Edge 66
5.5 Discussion 67

6 Conclusion 69
6.1 Conclusion and Recommendation 69
6.2 Future Work 70

Bibliography 71

List of Figures

1.1 AI applications in our daily life. 2
1.2 Different granularities of unstructured pruning and structured pruning. K represents the kernel size of CNNs filter, and C represents the number of channels of CNNs filter. 3
1.3 Lossless compression diagram 6
1.4 Lossy compression diagram 6

2.1 The structure of a neural network with single neuron. 9
2.2 The structure of a MLP with three hidden layers. 10
2.3 The operation process of cross-correlation. 11
2.4 The convolution operation after zero padding. 12
2.5 The convolution operation with stride of 2. 13
2.6 A CNNs with multiple input channels and multiple output channels. . 13
2.7 Comparison of network before and after filter-wise pruning. F_i represents the ith feature map in the network, while L_i represents the ith convolutional layer. 15
2.8 Value mapping of affine quantization and scale quantization. 17
2.9 A simple example of integer weights 20
2.10 Frequency table of integer weights 20
2.11 Hoffman Tree 21
2.12 Calculation of string size after Hoffman encoding compression 21
2.13 ZFP algorithm Step 1, divides the input data into basic blocks. . . . 22
2.14 ZFP algorithm Step 5, example diagram for 2D input with 4 bit planes. 23

3.1 Pruning workflow using PaddlePaddle. 26
3.2 Comparison of uniform and non-uniform ratio method. The orange rectangle in the convolution layer represents the filter to be pruned, where the yellow rectangle represents the kernels to be pruned. . . . 27
3.3 An example of sensitivity analysis of pruning on ResNet50 with first 3 convolutional layers (Conv). The value of the horizontal coordinate is the different pruning ratios, and the value of the vertical coordinate is the accuracy loss ratio. The base accuracy of ResNet50 is 75.42%, if we prune 20% of filters in Conv1, the accuracy loss of the whole model is $75.42\% * 12\% = 9.1\%$, then the accuracy of this model changes to $75.42\% - 9.1\% = 66.32\%$. 28

3.4 An example of sensitivity analysis of pruning on MobileNetV1 with first 3 convolutional layers (Conv). The value of the horizontal coordinate is the different pruning ratios, and the value of the vertical coordinate is the accuracy loss ratio. 28
3.5 Workflow of the Quant Aware Training (QAT). 30
3.6 Network structure before and after inserting the fake quantization operations. Quant and De-Quant represent quantization and dequantization operations respectively. Conv denotes a convolutional layer. 31
3.7 Workflow of the Post Training Quantization (PTQ). 32
3.8 Network structure of PTQ model. Quant and De-Quant represent quantization and de-quantization operations respectively. Conv denotes a convolutional layer. 33
3.9 Workflow of model compression pipeline. This pipeline is constructed by the combination of pruning and quantization. 34
3.10 The overall storage structure of the model. 34
3.11 The specific data structure of weight storage. 35
3.12 Data Compression Workflow 36
3.13 Workflow of analyzing the relationship between loss setting parameters θ and final model accuracy. 38
3.14 Lossy compression process for model floating point weights. 38
3.15 Workflow of Hybrid Compression. 39

4.1 The compression ratio comparison from model compression pipeline. The base size of MobileNetV1 and ResNet50 is 17.0 MB and 102.4 MB respectively. The compressed model size can be obtained by dividing base size by compression ratio. The P30 and P50 represents pruning with 30% and 50% pruning ratio respectively. 48
4.2 The Acc1 Drop comparison from model compression pipeline. The base Acc1 of MobileNetV1 and ResNet50 is 70.07% and 75.42% respectively. The Acc1 of compressed model can be obtained by minus Acc1 Drop from base Acc1. 49
4.3 The speedup comparison from model compression pipeline. The base inference speed of MobileNetV1 and ResNet50 is 0.6553 ms and 2.2217 ms respectively (batch size = 1). The accelrated speed can be computed by dividing base speed by speedup. 49
4.4 The ratio of float weights and integer weights in different models, and the corresponding compression ratio after quantization. 51
4.5 The distribution of integers(INT8) in the model parameters after quantization 52
4.6 The accuracy change of the model after zfp lossy compression. 54
4.7 The Acc1 drop comparison of hybrid compression. P30+QAT+D represents that we apply the data compression on the P30+QAT model. The base Acc1 of MobileNetV1 and ResNet50 is 70.07% and 75.42% respectively. 56

4.8 The compression ration comparison of hybrid compression. The base size of MobileNetV1 and ResNet50 is 17.0 MB and 102.4 MB respectively. 56
4.9 The speedup comparison of model compression pipeline on Jetson NANO. BS1 and BS2 represent the cases when the batch size is set to 1 and 2 respectively. 59
4.10 The speedup comparison from model compression pipeline on Jetson Xavier NX. The base inference speed of MobileNetV1 and ResNet50 is 4.1660 ms and 17.1248 ms respectively (batch size = 1). The accelerated speed can be computed by dividing base speed by speedup. 60
4.11 FPS comparison between Nano and NX on MobileNetV1. The value above the orange column is the FPS ratio obtained by dividing NX by Nano. The FPS values can be found in the data table in the figure. It should be noted that the FPS value of Nano under the quantization column comes from FP16 precision inference. 61
4.12 FPS comparison between Nano and NX on ResNet50. 62

5.1 Changes in network transmission time before and after compression by the ResNet50 model, taking into account the local network speed. 64
5.2 Scatter plot of power consumption vs. arithmetic power for different devices. 65

List of Tables

1.1 Comparison of the energy and die area of different computation operations and energy cost of memory read in a 45 nm process. The data are referenced from [23]. 5

4.1 Model information of MobileNetV1 and ResNet50 41
4.2 Acc1 comparision of different pruning strategy. Method and Strategy denote different pruning algorithms and strategy respectively. The Pruned Ratio is the percentage of FLOPs pruned. 44
4.3 Comparison of accuracy of pruned MobileNetV1 and ResNet50 with baseline. In this table, the pruning strategy we used is base on sensitivity analysis. So the 30 and 50 denotes that we plan to prune 30% and 50% of FLOPs of the model. 44
4.4 Comparison of model size of pruned MobileNetV1 and ResNet50 with baseline. The unit of model size is megabytes (MB). 44
4.5 Comparison of performance of pruned MobileNetV1 and ResNet50 with baseline. Performance is measured using PaddleInference, the batch size is set to 1, the unit is milliseconds. 45
4.6 The fine-tune resluts of MobileNetV1 and ResNet50 using FPGM algorithm. The A16 in Method column represents the number of filters is aligned to 16. The Perf is measured when batch size = 1. . . 45
4.7 Comparison of model size of quantized MobileNetV1 and ResNet50 with baseline.The unit of model size is megabytes(MB). 46
4.8 Comparison of accuracy of quantized MobileNetV1 and ResNet50. Acc1 represents the Top-1 prediction accuracy of the model on the validation set 46
4.9 Comparison of inference speed of quantized MobileNetV1 and ResNet50 with baseline. Performance is measured using PaddleInference, the batch size is set to 1, the speedup is the ratio of baseline and pruned performance. 47
4.10 Comparison of model size of in model compression pipeline. The unit of model size is megabytes (MB). P represents the pruning and P+PTQ represents the PTQ after pruning. 47
4.11 Comparison of accuracy in model compression pipeline. 47
4.12 Comparison of inference speed in model compression pipeline. Performance is measured using PaddleInference, the batch size is set to 1. 48

4.13 Model size(MB) after compression by different compression algorithms 50
4.14 Additional Tests: The original size(MB) of the 11 models and the size(MB) after INT8 quantization. 50
4.15 The size(MB) of the integer part of the model changes after Huffman coding 52
4.16 The size(MB) change of AlexNet's FP32 part of data after lossless compression 53
4.17 The size(MB) change of AlexNet's FP32 part of data after lossy compression 53
4.18 Model FP32 part size(MB), compression ratio of FP32 part and accuracy after lossy compression according to sensitivity analysis result 54
4.19 Acc1 and model size comparision after hybrid compression. The DC in Method column denotes Data Compression. The letter B denotes the Base. 55
4.20 NVIDIA Jetson Nano Configuration 57
4.21 Comparison of inference speed on NVIDIA Jetson Nano platform. Performance is measured using PaddleInference, the batch size is set to 1. The values in the table are in milliseconds. The P represents pruning, 30/50 is the pruned ratio. 58
4.22 Comparison of inference speed on NVIDIA Jetson Nano platform. The batch size is set to 2. 58
4.23 NVIDIA Jetson Xavier NX Configuration 59
4.24 Comparison of inference speed on NVIDIA Jetson xavier NX. Performance is measured using PaddleInference and the batch size is set to 1. The unit is milliseconds. 60

5.1 Average energy consumption (kw/h) to be consumed by ResNet50 during model preparation, Dataset is ImageNet 66

1 Introduction

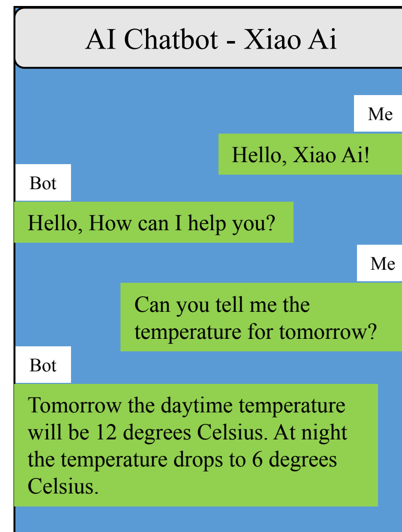
Deep learning has made remarkable progress in recent years, enabling breakthroughs in computer vision, natural language processing, recommendation and decision making, and other areas. Deep neural networks (DNNs) play a key role in the success of these dazzling artificial intelligence (AI) applications. However, the success of most AI tasks is due to the numerous parameters and complex structure of DNNs model [9]. The huge demand for computing power and memory resources needs to be met when training or deploying the model, whether in tech giants' data centers or an embedded device at the edge.
Above situation is like red AI , where researchers pursue stronger results by using massive computing power and ignoring cost [52]. Unlike red AI, our master thesis work is towards to green AI, which promotes approaches that have performance and efficiency trade-offs. The key research problem of our master thesis is how to reduce the memory footprint and computation cost of DNNs without losing significant accuracy. The compression technique is one proven way to solve this problem.
In our work, we intend to explore the effects of model compression and data compression on DNNs model. In model compression, different levels and methods of pruning and quantization techniques will be applied to reduce the memory and computation cost of DNNs model. Lossless and lossy data compression are used in data compression to minimize the model's storage size, which can improve bandwidth transfer, although lossy compression introduces some accuracy loss. However, the loss of accuracy caused by lossy compression should not be ignored. Furthermore, we will combine model compression with data compression to form hybrid compression in order to achieve the ultimate compression results.
Although hybrid compression reduces the computation cost and storage size of the model and can improve the inference speed and network transmission speed, the time overhead and accuracy loss of it can not be ignored. At the end of our thesis, we will analyze above trade-offs from accuracy, compression ratio and speedup aspects and then provide reasonable conclusions.

1.1 Overview of DNNs

With the rise of artificial intelligence (AI), deep neural networks (DNNs) have reached a rapid and extensive development in computer vision, natural language processing, speech recognition, autonomous driving, and many other fields. Endowed with a deep structure, DNNs have the ability to learn high-level features from data, which makes it distinguished from the traditional methods. Nowadays,

DNNs have become the mainstay of AI tasks in both academia and industry [9].

As early as 1998, LeCun proposed the LeNet5 [30] neural network for handwritten number recognition. However, due to the limitation of computing hardware and insufficient computing power at that time, it was challenging to train the DNNs, so the development of DNNs fell into stagnation. After that, this milestone breakthrough boosted a new wave of research on DNNs in our world, and various kinds of powerful DNNs models were proposed by academia and industry. For example, VGG [55] and ResNet [21] for image classification tasks, BERT [10] and GPT-3 [7] for natural language processing tasks, DLRM [41] and Deep Interest Network [65] for recommendation system tasks. As shown in Figure 1.1, the application of AI also provides a lot of convenience to people's daily life in various aspects, and contributes to social and economic prosperity.



Computer Vision

Natural Language Process

Figure 1.1: AI applications in our daily life.

Although these models show human-level capabilities in their domain, they usually have millions to billions of parameters and significant computation cost. For inferring DNNs models, Graphics Processing Units (GPUs) with high processing parallelism and memory bandwidth has been the primary platform for AI task in data centers [9]. However, the era of the Internet Of Things (IOT) has come, and edge computing is booming. AI should not only exist in the data center. Applications of edge computing urgently need the powerful processing capabilities of DNNs to handle various complicated scenarios [61]. Deploying deep learning models on edge devices (e.g., smart camera, robot, mobile phone, etc.), which have limitations on computing power, memory resources and energy, increasingly attracts attention from all walks of life. Therefore, applying different optimization techniques to reduce the computation and memory cost of DNNs models and to gain acceleration becomes an urgent and promising topic.

1.2 Model Compression

There has been a wave of software and hardware optimization techniques to pursue the trade-offs between performance, memory cost, and application accuracy of AI models in the past few years. Researchers have been designing Domain-Specific-Accelerators (DSAs), optimizing memory hierarchies, and performing in-memory computation to obtain high performance on the hardware [9]. On the software side, the design of AI compilers, computation graph optimization, and model compression have become the mainstream approaches [39]. The approaches for model compression are quite divergent, such as pruning, quantization, matrix low-rank factorization, network architecture search, etc. Pruning and quantization will be briefly introduced in the following sections.

1.2.1 Pruning

Network pruning is an efficient model compression method that attempts to reduce the computation operation of a DNNs model while reducing the number of memory accesses and accelerating the inference speed [32]. The main strategy of pruning is to remove the redundant parameters or neurons which do not significantly contribute to the accuracy of prediction results from DNNs. How to accurately measure the importance of the existing neurons, how to plan the structure to be removed, and how to recover the performance loss caused by pruning become the key issues to be considered in the pruning algorithm.
Regarding whether the structure to be pruned is symmetrical, pruning can be categorized into unstructured pruning and structured pruning [32]. Unstructured pruning can occur element-wise, where individual weights can be pruned. Structured pruning can be performed filter-wise, shape-wise, channel-wise, etc. The granularity of these two types of pruning methods for Convolutional Neural Networks (CNNs) is shown in Figure 1.2

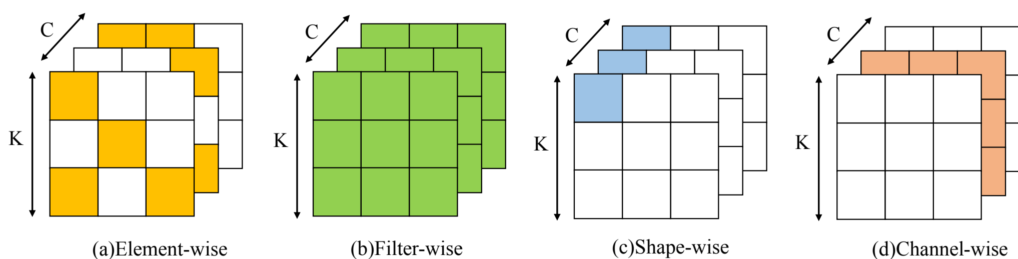


Figure 1.2: Different granularities of unstructured pruning and structured pruning. K represents the kernel size of CNNs filter, and C represents the number of channels of CNNs filter.

Unstructured pruning estimates the importance of each weight individually and removes it separately, which has the finest granularity. This approach usually results in smaller model size and higher compression ratios than structured pruning. However, by zeroing the weights, unstructured pruning does not change the network structure, but gives us a network with sparse weight matrices having many zeros.

High sparsity from unstructured pruning results in a large index memory storage and irregular memory accesses, which may lead to sparse weight matrices that can not be efficiently computed on some general hardware platforms [39]. Unstructured pruning can only reduce the number of parameters in DNNs models, and special hardware and software support is required to obtain speedup [19]. Researchers in [14] states that even with the NVIDIA cuSPARSE library, a meaningful speedup on GPU is only achieved when sparsity exceeds 98%.

Structured pruning has a coarser granularity, as it may remove an entire layer of the DNNs model or some of the channels in the CNNs kernel directly. By changing the network structure, this pruning method can reduce the number of parameters and the number of computation operations in the DNNs model, which can lead to an increase in inference speed, less hardware energy consumption and less storage and memory usage. Moreover, inferring a structured pruning model does not incur additional software and hardware overhead and can be performed with common deep learning frameworks and hardware. However, structured pruning is more aggressive and may cause degradation of model performance (e.g., prediction accuracy). Therefore, it is a challenge to achieve a high compression ratio without too much loss of accuracy in structured pruning.

It is important to be aware that pruning will change the structure of the model, and the pruned model needs to be retrained, also called fine-tuned, to restore the prediction accuracy. Specifically, unstructured pruning can only be fine-tuned to restore accuracy [32], which means the unstructured pruning should implement on a pre-trained model. But structured pruning can be fine-tuned or trained from scratch, [36] gives a result that contradicts the common perception that a pruned network trained from scratch may perform better than a fine-tuned model.

1.2.2 Quantization

Quantization is a process in which a large range of values is represented by a much smaller set of values [51]. In model compression, quantization attempts to reduce the bit width of the parameters in DNNs model, which can bring benefits such as low-bit computation operations and shrinked memory cost.

As shown in Table 1.1, the benefits of quantization are illustrated more clearly and obviously. Compared to 32-bit floating point, lower bit representations consume less energy for addition, multiplication operations and data movement. Furthermore, the shrinked model size from quantization can reduce the amount of on-chip memory, which reduces the area cost of the chip. It allows for more parallel computation, data reuse, and increases throughput [57]. This is also why quantization is prevalent and demanding during DNNs inference, especially for resource-limited hardware.

Table 1.1: Comparison of the energy and die area of different computation operations and energy cost of memory read in a 45 nm process. The data are referenced from [23].

Operation	Energy (pJ)	Area (μm^2)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
8b Mult	0.2	282
32b Mult	3.1	3495

After implementing quantization on the DNNs model, it is very necessary to tune the model parameters to reduce the impact of quantization on model output. According to whether neural network training is involved, quantization can be categorized into Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ). QAT takes into account the characteristics of quantization in the training process. The common method is to simulate the accuracy loss caused by quantization by introducing fake quantization nodes, and then execute the forward and backward computation on floating point, but quantify the model parameters after each gradient update. QAT usually achieves higher accuracy but introduces additional overheads due to the need for training and using the entire data set. If an application using a quantized model cannot tolerate low accuracy, then the extra overhead of QAT is worth it. PTQ does not require retraining the network to quantize the model and adjust the parameters. PTQ can complete the quantization procedure using a limited number of unlabeled data [15]. This is friendly for rapid hardware deployment, but PTQ causes more accuracy loss in the model than QAT, so mitigating the accuracy degradation of PTQ is becoming a popular topic.

1.3 Data Compression

Unlike approaches such as pruning and quantization, data compression starts with the distribution of data to reduce the redundancy of model weights, thus achieving further compression of deep learning models.

Among data compression as well as loss theory, this area of research work was mainly laid down by the American scholar Claude Elwood Shannon, who published fundamental papers [54] in this area in the late 1940s and early 1950s. Data compression techniques are now widely used in various fields.

There are basically two processes of data compression for deep learning models, the compression of the original data according to some compression algorithm and the decompression of the original model based on the compressed values. According to the final result, we can classify the data compression algorithms into lossless compression algorithms and lossy compression algorithms.

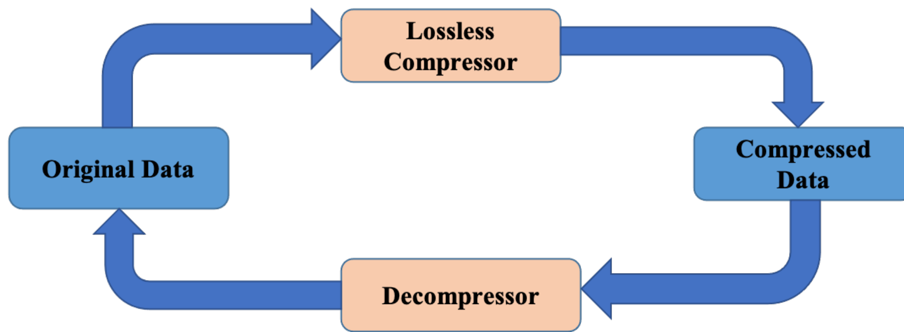


Figure 1.3: Lossless compression diagram

As in Figure 1.3, lossless compression allows the exact restoration of the original data from the compressed data, which is also known as reversible compression. If classified by time [17], the traditional lossless compression algorithms are Run Length Encoding (RLE) [2], Huffman Encoding [25], LZ-77 Encoding [66], etc. Modern lossless compression algorithms include Deflate [26], Lempel Ziv Markov Chain Algorithm (LZMA) [48], Bzip2 [53], etc.

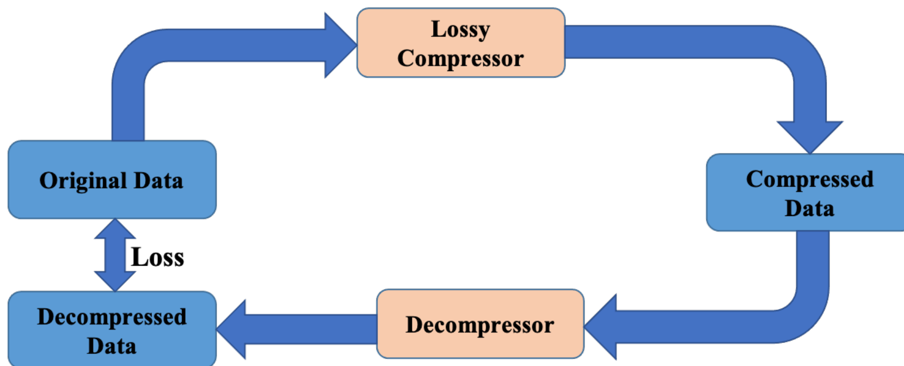


Figure 1.4: Lossy compression diagram

While lossy compression can only restore an approximation of the original data from the compressed data, which is irreversible compression. However, at the same time, lossy compression often results in a higher compression ratio than lossless compression [27].

1.4 Contribution

We explore the model compression and data compression for neural networks models, and construct the hybrid compression workflow in our thesis work.
We build the model compression pipeline using pruning and quantization methods. The structured pruning without additional hardware and software support can combine very well with quantization. So we can quantize a pruned model to achieve higher compression ratio and faster inference speed. The compressed model after this pipeline can easily deployed on the common used hardware like Nvidia GPU. Furthermore, the quantized INT8 weights and un-quantized FP32 data are

stored in the model file together, which is well suited for our subsequent work (data compression).
By analyzing the type and distribution of weights in the pruned and quantized model, we propose a set of data compression methods to further compress the model size. This data compression method can be used to determine the importance of each layer of weights in the model by performing sensitivity analysis on the model, so as to better compress the model while reducing the accuracy degradation. We call the combination of model compression and data compression methods hybrid compression. In hybrid compression, data compression is only responsible for reducing the size of the model file storage and does not affect the runtime file size. When deployed to a specific device, only decompression is required for the files.
We evaluate our hybrid compression on the real edge devices. We find that not all hardware can gain inference speed through model compression, which reminds users to consider what hardware is appropriate when deploying AI models for practicial applications.

2

Theory

This chapter introduces the theoretical foundations of deep neural networks, model compression and data compression.

2.1 Deep Neural Networks

As the name implies, *deep* in DNNs means that the neural network has many layers between input and output. There are many types of DNNs, such as multilayer perceptron (MLP), convolutional neural networks (CNNs), and recurrent neural networks (RNNs). These various DNNs usually have different structures and are applied to different task scenarios. In our thesis work, we focus on CNNs applied to the field of computer vision. In the following sections, we briefly introduce the basic structure of neural networks and CNNs.

2.1.1 Structure of Neural Network

A neuron is the basic unit of a neural network, which receives signals from input neurons and then performs a specific nonlinear transformation, outputting the result to the next neurons [9].

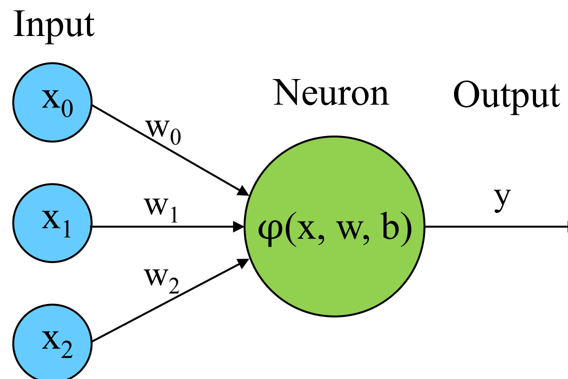


Figure 2.1: The structure of a neural network with single neuron.

As shown in Figure 2.1, this is the structure of a single neuron. We refer to the input and output signals of neurons as activations. The connection between neurons with input and output called synapse, the weights are introduced to scale the value of the activations on synapses. Different weights will respond differently to the activations, just as the brain emphasizes different information, and the process of

adjusting weights can be seen as a learning process for neurons. In a neuron, the function used for the nonlinear transformation is called the activation function, and the widely used activation functions are Sigmoid [18], ReLU [62] and tanh [16] etc. The computation on the ith neuron can be shown in the following formula:

$$y = \phi(\sum_j x_j w_{ij} + b_i). \tag{2.1}$$

Where y is the output, ϕ is the activation function, x is the input, w is the weight, and b is the bias to be inserted. Such a structured network with one neuron can be called a single-layer neural network.
Nowadays, in various fields, neuron networks that perform beyond human capabilities are usually DNNs. In the following contents, we will introduce deep neural networks with the simplest model, the multilayer perceptron (MLP). Compared with single-layer neural networks, MLP introduces one or more hidden layers between the input and output layers, which significantly enhances the representation ability of the models. In the MLP shown in the Figure 2.2, the input layer has two units, the output layer ($Layer3$) has one unit. There are two hidden layers in the middle of the network, they are $Layer1$ with 4 neurons and $Layer2$ with 3 neurons. Since the input layer is not involved in the computation, the number of layers of this MLP is 3. The neurons in each hidden layer are connected to every neuron of the input layer, and such a hidden layer is also called a fully connected layer (FC-layer).

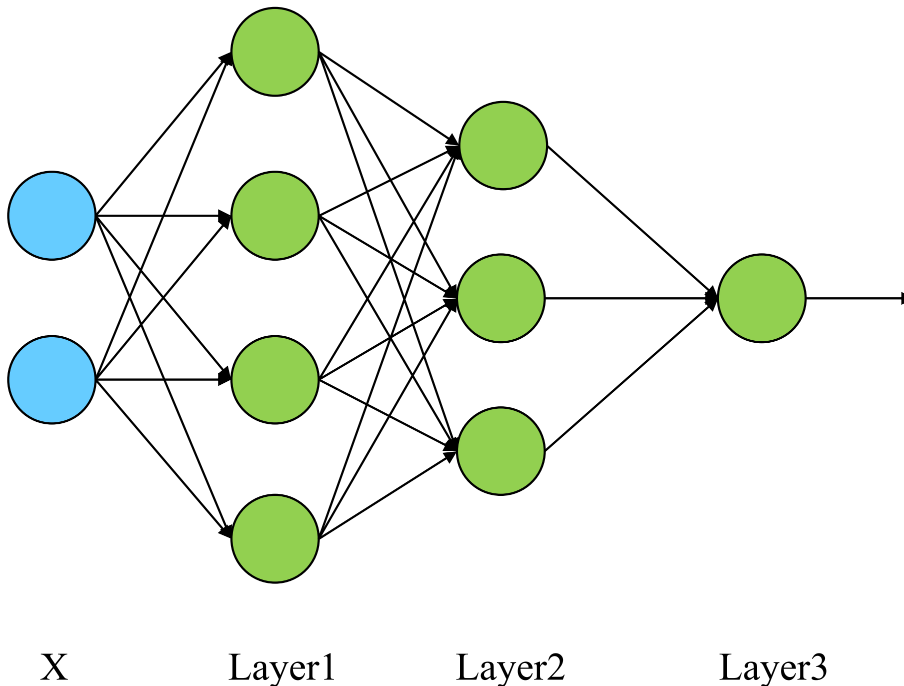


Figure 2.2: The structure of a MLP with three hidden layers.

In the Figure 2.2, given that $W^1 = (w_{12}...w_{14}, w_{21}...w_{24})$ as the weights of the first layer, $X = (x_1, x_2)$ as the input of the first layer, B^1 is the bias, and A^1 is the activation. In $Layer1$, the forward computation is $A^1 = \phi(W^1X + B^1)$, and then

the A^1 will be the input signals for $Layer2$. In general, Let l represent the lth hidden layer, then for each layer we have :

$$A^l = \phi(W^l A^{(l-1)} + B^l). \tag{2.2}$$

Today, the typical numbers of layers used in DNNs range from 5 to more than 1,000, and DNNs are capable of learning high-level features with more complexity and abstraction than shallower neural networks [57]. Depending on the application scenario, there are many variants of DNNs. Different types of layers can be combined to build the DNNs model, and these layers include FC-layers, convolutional layers, etc. In the next section, we will discuss CNNs, which is a type of powerful and widely used DNNs.

2.1.2 Structure of Convolutional neural networks

Convolutional Neural Networks are a class of DNNs, which mainly use convolution layers to extract features from input data. In this section, we describe in detail the convolutional layer and its backbone operations.

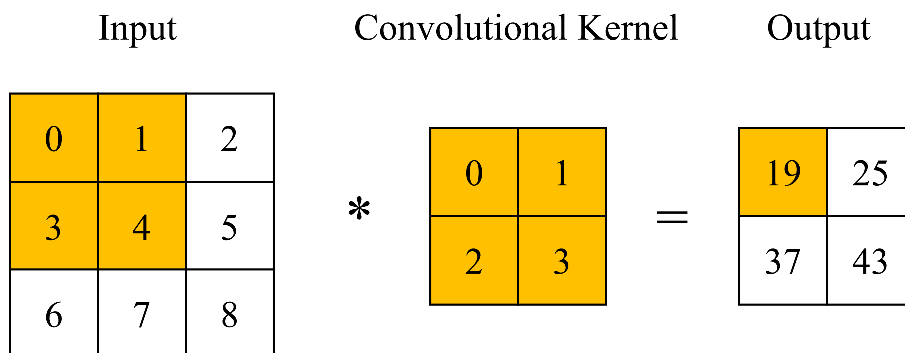


Figure 2.3: The operation process of cross-correlation.

In the convolutional layer, the operation performed is similar to cross-correlation. The cross-correlation computation occurs on the input 2D matrix and the 2D convolution kernel. As shown in the Figure 2.3, the input is a 2D matrix of height and width 3, whose shape can be denoted as (3,3), and the size of the convolution kernel is 2. The orange value of 19 in the output matrix is calculated from the orange region of the input sub-matrix and the convolution kernel, which is $0 * 0 + 1 * 1 + 3 * 2 + 4 * 3 = 19$. Next, the convolution kernel slides over the input matrix, in order, from left to right and from top to bottom. When the convolution kernel slides to a certain position, the multiply-and-accumulates(MACs) will be executed on the (2,2) sub-matrix of input and the (2,2) kernel. Given that Out is the ouput, In is the input and K denotes the convolutional kernel, then cross-correlation operation can be formulated as:

$$Out(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} In(i+m, j+n)K(m,n). \tag{2.3}$$

where M and N are the number of rows and columns of the convolution kernel, respectively.

Convolution is slightly different from cross-correlation operation, but they are very similar. To get the output of the convolution operation, we only need to flip the 2D kernel matrix both horizontally and vertically, and then perform the cross-correlation operation with the input matrix [64]. Many convolution implementations from machine learning libraries are actually cross-correlation, and in our thesis, the convolution operations mentioned later are all cross-correlation operations.
In the convolution operation, the shape of the output is determined by both the input and the convolution kernel, and its shape can be calculated as:

$$(O_h, O_w) = (N_h - K_h + 1, N_w - K_w + 1). \tag{2.4}$$

In the above equation, O, N, and K are the size of the output, input, and convolution kernel, respectively.
Since the convolution operation is performed only once on the boundary of the input matrix, it may result in the loss of information on the boundary. One solution named zero padding is to add extra zero pixels to the boundary of the input matrix, thus increasing the effective size of the input. In general, if we add $2P_h$ rows of padding on height (P_h on top and P_h on bottom) and $2P_w$ columns of padding on width (P_w on the left and P_w on the right), the output shape will be:

$$(O_h, O_w) = (N_h + 2P_h - K_h + 1, N_w + 2P_w - K_w + 1). \tag{2.5}$$

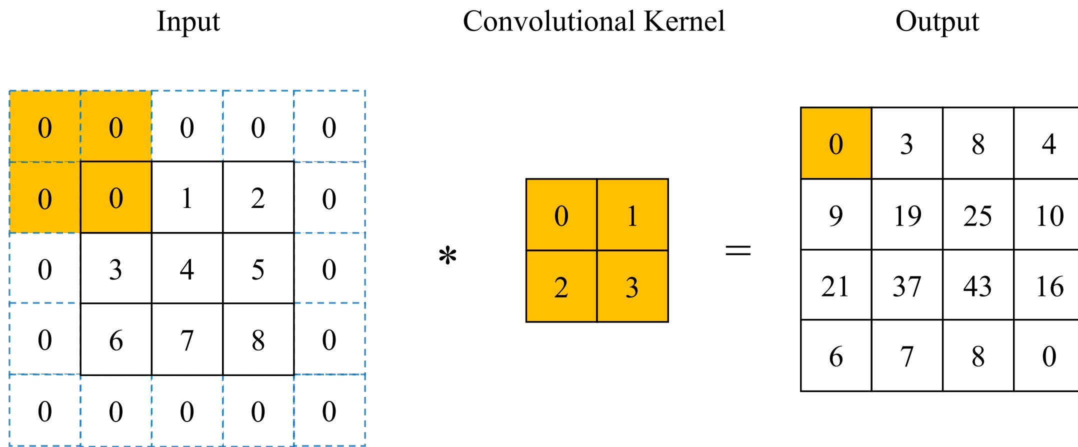


Figure 2.4: The convolution operation after zero padding.

As shown in the Figure 2.4, we set P_h and P_w to 1, the original (3,3) input matrix turns into a (5,5) matrix, and we can observe that the boundaries that we pad are all filled with zeros. Accordingly, after the computation with padding, the size of the output is changed.
In convolution operation, the convolution kernel usually slides one step at a time on the input data. Still, when we want to reduce the output dimension drastically, we can let the kernel skip some elements, which means it can slide more than one step in every operation. The stride refers to the number of rows and columns that the kernel travels through on each slide. We illustrate a convolution operation with a vertical stride of 2 and a horizontal stride of 2 in Figure 2.5. The result of colored

output is computed by the kernel on the sub-matrix of the corresponding region of the input. In general, when the stride is set to S, the output size will be :

$$(O_h, O_w) = (\lfloor \frac{N_h + 2P_h - K_h + S_h}{S_h} \rfloor, \lfloor \frac{N_w + 2P_w - K_w + S_w}{S_w} \rfloor). \tag{2.6}$$

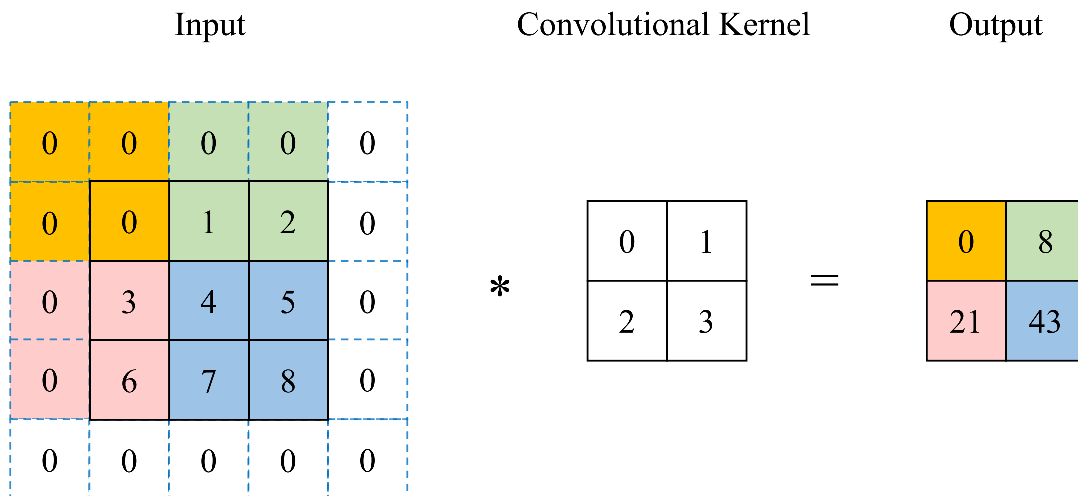


Figure 2.5: The convolution operation with stride of 2.

In the previous content, we describe the basic 2D convolutional computation. However, in real applications, especially in computer vision, the input data has a higher dimensionality and can be a multidimensional array. For example, a color picture can have three RGB color channels, and we call this dimension channel. When the input data is a multidimensional array, we also need to extend the dimensionality of the convolutional kernels to complete the convolution operations.

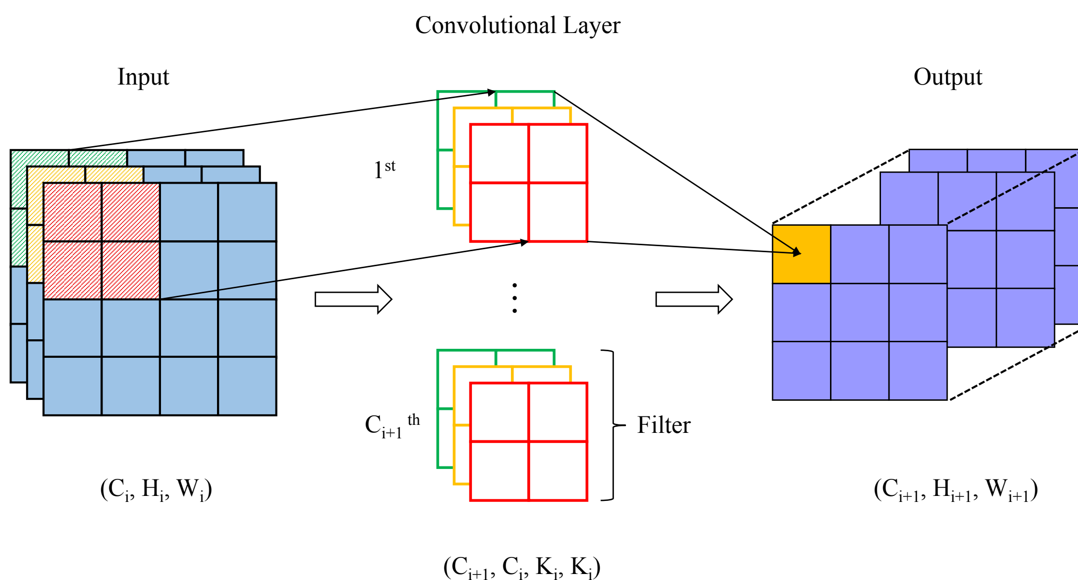


Figure 2.6: A CNNs with multiple input channels and multiple output channels.

As shown in the Figure 2.6, the shape of the input data is (C_i, H_i, W_i), and the number of channels of the convolution kernel is increased to C_i in order to complete the convolution computation. We usually refer to a set of convolution kernels with a C_i number of channels as a filter. Usually a convolutional layer is composed of one or more filters. In the convolutional layer of the figure, there are C_{i+1} filters in total. In the first filter, we can observe that the red 2D kernel will perform the convolution computation in the red region of the input, and accordingly, the yellow and green kernels will do the computation in their corresponding regions. When the first filter finishes the first convolution operation, the result of the orange square in the output is obtained. When the first filter slides over the input completely, the result of the first channel in the output is obtained. Obviously, after the C_{i+1} filter completes its operation on the input, the output will obtain the results of the C_{i+1} channel. It can be noticed that the number of channels of input, C_i, determines the number of channels in the subsequent convolutional layer filters. The number of filters in the convolutional layer, C_{i+1}, determines the number of channels in output. We can obviously observe that they have this kind of relationship. Note that the number of filters in the convolutional layer is typically set by the designer of the network structure.

In the forward computation of CNNs, floating point operations (FLOPs) are introduced to measure the computation complexity of a network. If we treat the MAC as one operation, the FLOPs of ith convolutional layer can be given by

$$FLOPs = C_{i+1} * C_i * {K_i}^2 * H_{i+1} * W_{i+1}. \tag{2.7}$$

Where C_{i+1} is the number of filters, and C_i is the number of channel in every filter. K_i is the size of the convolutional kernel and H_{i+1} and W_{i+1} are the height and width of the output.

2.2 Model Compression

The goal of model compression is to achieve simplification of the original model without significant loss of accuracy, which often leads to a reduction in model storage and a boost in model inference speed. There are various approaches to model compression, and we will focus on pruning and quantization in the following sections.

2.2.1 Pruning

By removing redundant parameters of DNNs, pruning is widely used to reduce the model storage and computation operations. In the Section 1.2.1, we categorize pruning into unstructured and structured pruning and introduce the different pruning granularities. Compared to other types of pruning, filter-wise structured pruning is friendlier to software libraries and hardware, and it also provides excellent pruning effects.

2.2.1.1 Filter-Wise Pruning

In our thesis work, we mainly focus on the filter-wise pruning, and we will describe the process of this method in the following content.

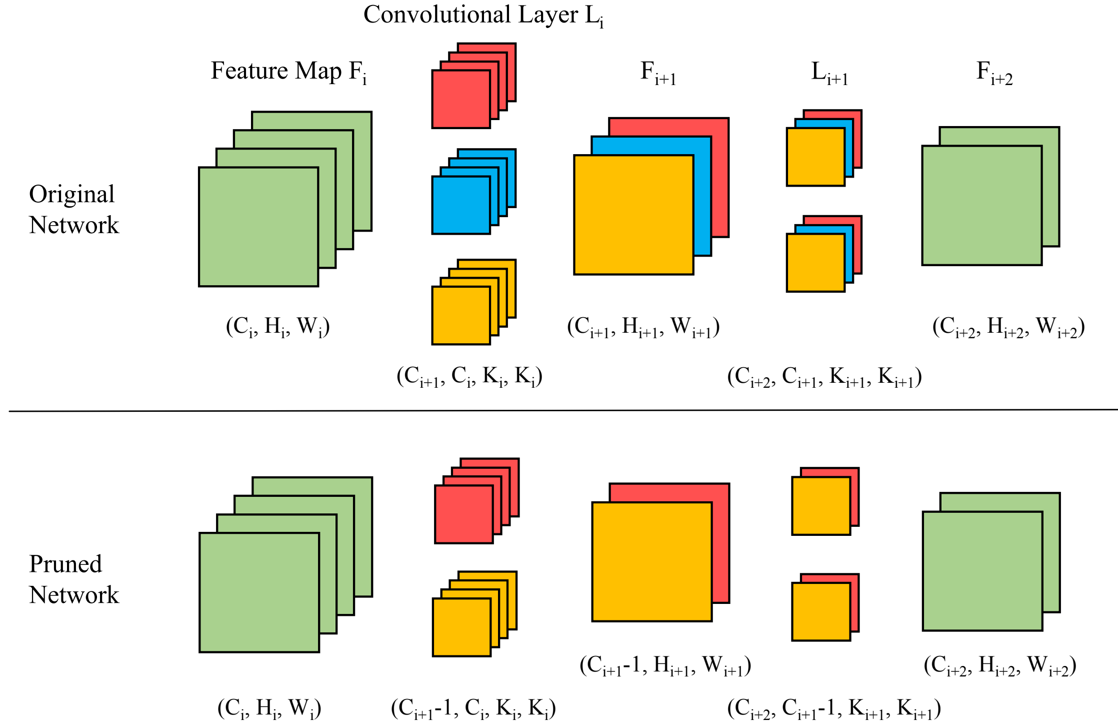


Figure 2.7: Comparison of network before and after filter-wise pruning. F_i represents the ith feature map in the network, while L_i represents the ith convolutional layer.

As shown in Figure 2.7, there are two simple neural networks with two convolutional layers respectively, the upper one is the original network. After we prune the blue filter in L_i, then the entire network structure will become the part below in the figure. We can observe that when the blue filter in L_i is removed, the blue channel will no longer exist in the F_{i+1}. To ensure the correct computation of F_{i+1} and L_{i+1}, the kernels that should have been applied to blue channel in F_{i+1} will also be removed. In brief, pruning the filters in L_i will decrease the number of kernels in L_{i+1}, and we call such a pair of convolutional layers a group.
Generally, We assume that in the original network, the shape of the ith convolutional layer is denoted as (C_{i+1}, C_i, K_i, K_i). C_{i+1} represents the number of filters in this convolutional layer, C_i represents the number of kernels in each filter, and K_i represents the size of each kernel. The input of this convolution layer is the ith feature map, its dimension is (C_i, H_i, W_i), where C_i is the number of channels, H_i and W_i are the height and width of feature map respectively. The $i+1$th feature map with dimensions of $(C_{i+1}, H_{i+1}, W_{i+1})$ is the output of the F_i and L_i. Note that the number of kernels in each filter of the ith convolutional layer is equal to the number of channels in the input feature map, and the number of filters is equal to the number of channels in the output feature map.

The number of computation operation of L_i is $C_{i+1} * C_i * {K_i}^2 * H_{i+1} * W_{i+1}$, if we prune one filter in L_i, then the operation becomes $(C_{i+1} - 1) * C_i * {K_i}^2 * H_{i+1} * W_{i+1}$, so $C_i * {K_i}^2 * H_{i+1} * W_{i+1}$ operations will be reduced. Because the number of filters in L_i is reduced by 1, the channel number of its output feature map is also reduced by 1. This results in a corresponding reduction in kernels from each filter in L_{i+1} that were supposed to finish the computation with the reduced channels. Therefore, the operation of L_{i+1} will shrink from $C_{i+2} * C_{i+1} * {K_{i+1}}^2 * H_{i+2} * W_{i+2}$ to $C_{i+2} * (C_{i+1} - 1) * {K_{i+1}}^2 * H_{i+2} * W_{i+2}$. Pruning p filters of L_i will reduce p/C_{i+1} of the computation cost for both L_i and L_{i+1}.
The filter-wise pruning needs to remove filters that have little contribution to the model's performance. So the selection of the filters to be pruned of each convolutional layer is crucial in filter-wise pruning. Two metrics for selecting the filters to be pruned are described in the following sections.

2.2.1.2 Select by L1-norm

As one of the earliest filter-wise pruning algorithms, [31] measures the relative importance of filters in each layer by L1-norm , which is actually the sum of the absolute values of the weights in the filter:

$$S_j = \|F_{i,j}\|_1 . \tag{2.8}$$

Where $F_{i,j}$ represents the jth filter of the ith convolutional layer. After collecting the S_j, all filters in each layer are sorted by this value. Filters with smaller value of S_j are considered less important and are the first choice to be pruned.

2.2.1.3 Select by Geometric Median

Researchers in [22] consider filters as several points in Euclidean space, and the "centroid" of these points can be obtained by computing geometric median [13]. Filters closer to the centroid are considered to be redundant and can be represented by other filters. Such redundant filters can be pruned without significant impact on the performance of the model. The name of this algorithm is called filter pruning via geometric median (FPGM).
Suppose there are n points $a^{(i)}$ with $i \in (1, n)$, and each $a^{(i)} \in \mathbb{R}^d$. The purpose is to find a point x^* that minimizes the sum of euclidean distances to every a(i):

$$x^* \in \underset{x \in \mathbb{R}^d}{\arg\min} \sum_{i \in [1,n]} \left\| x - a^{(i)} \right\|_2 . \tag{2.9}$$

Inspired by this idea, the filters in each layer are considered as points $a^{(i)}$ in Equation 2.9. The problem is thus transferred to calculating the geometric median F_i^{GM} of all filters:

$$F_i^{GM} \in \underset{x \in \mathbb{R}^{C_i \times K \times K}}{\arg\min} \sum_{j \in [1, C_{i+1}]} \|x - F_{i,j}\|_2 . \tag{2.10}$$

Where C_i is the number of kernels with size of K in every filter. $F_{i,j}$ represents the jth filter of the ith convolutional layer, and there are totally C_{i+1} filters in one layer.

After obtaining the F_i^{GM}, we try to find the filter with the closest euclidean distance to the F_i^{GM}, which is the redundant filter F_{i,j^*}. This process can be formulated as:

$$F_{i,j^*} \in \underset{j \in [1, C_{i+1}]}{\arg\min} \left\| F_{i,j} - F_i^{GM} \right\|_2 . \tag{2.11}$$

Furthermore, we can just select P_i filters that are closest to the F_i^{GM} if we intend to prune P_i filters in the ith layer.

2.2.2 Quantization

Quantization is one of the most popular methods to reduce the memory footprint and computation cost of DNNs model. In the following content, we will discuss the fundamentals of quantization and how to determine the quantization parameters.

2.2.2.1 Fundamental of Quantization

In quantization, we first need to select a range of real numbers to be quantized and then map these real numbers to integers that the quantization bit width can represent. We focus on quantizing 32-bit floating point into 8-bit integers in our work. In detail, we chose the range $[\beta, \alpha]$ from the floating point and map them into the range of $[-2^{b-1}, 2^{b-1} - 1]$, where $b = 8$ is the quantization bit width for integers. Figure 2.8 illustrates two common transformations of quantization for value mapping, which are affine quantization and scale quantization.

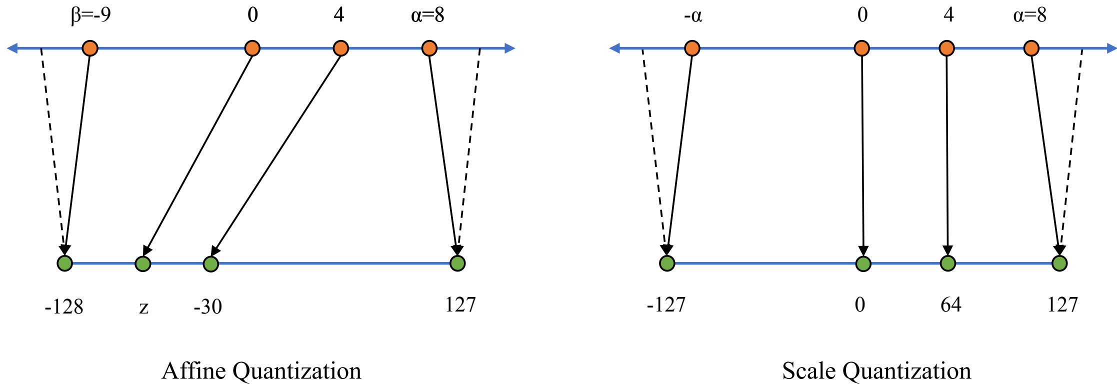


Figure 2.8: Value mapping of affine quantization and scale quantization.

Affine quantization. In affine quantization, the transformation is defined as:

$$f(x) = S \cdot x + z \tag{2.12}$$

Where S is the scale factor and z is the zero-point, and they are formulated as:

$$S = \frac{2^b - 1}{\alpha - \beta} \tag{2.13}$$

$$z = -\lfloor \beta \cdot S \rceil - 2^{b-1} \tag{2.14}$$

After transformation, if the value of the integers is out the range of $[-2^{b-1}, 2^{b-1}-1]$, we have to clip the outliers to the endpoint value. We define the clip process as :

$$Clip(x, l, r) = \begin{cases} l, & x < l \\ x, & l \leq x \leq r \\ r, & x > r \end{cases} \tag{2.15}$$

Then the affine quantization operation can be defined as:

$$x_q = Clip(\lfloor S \cdot x + z \rceil, -2^{b-1}, 2^{b-1} - 1) \tag{2.16}$$

Accordingly, the de-quantization that converts an integer to a floating-point number can be expressed as:

$$x_{dq} = \frac{x_q - z}{S} \tag{2.17}$$

Scale quantization. The scale quantization is also called symmetric quantization, where the real value range of input and quantized value range are around zero symmetrically. The zero-point is not needed anymore in the transformation function, which represented as:

$$f(x) = S \cdot x \tag{2.18}$$

The scale factor for scale quantization is changed to:

$$S = \frac{2^{b-1} - 1}{\alpha} \tag{2.19}$$

And the scale quantization operation is given as:

$$x_q = Clip(\lfloor S \cdot x \rceil, -2^{b-1} + 1, 2^{b-1} - 1) \tag{2.20}$$

Due to the presence of zero-point, affine quantization introduces computational overhead and may eliminate the performance benefits from integer computation, while scale quantization is a proven method that is sufficient for 8-bit integer quantization on DNNs [63]. In our work, the quantization method applied is scale quantization.

2.2.2.2 Quantization Parameters

We can clearly observe that in the formula 2.19, after $b = 8$ is determined, the remaining problem is transformed into how to choose the quantization parameter α. In the following, we introduce some popular methods of choosing this quantization parameter.

Maximum absolute value. This method directly picks the maximum of the absolute value of the input data x, that is, we make $\alpha = max(abs(x))$.
Moving average of max absolute value. The quantization parameter α is computed by moving average of maximum absolute values of the input data:

$$\alpha_t = k \cdot \alpha_{t-1} + (1 - k) \cdot max(abs(x)). \tag{2.21}$$

Where α_t is the quantization parameter of current batch of input data x, and α_{t-1} is for the last batch. k is a coefficient value for moving average, and the initial value of α_t is the $max(abs(x))$ from the first batch of data.

Percentile. In this method, α is set to a percentile (e.g. 99%, 99.9%) of the distribution of the input absolute values. For instance, 99% percentile would discard 1% of the larger values of the whole input data, then we choose the value that lies exactly in the 99% percentile as α.

The maximum absolute value method is widely used in weight quantization [28], and the moving average of max absolute value and percentile methods are usually used in activation quantization [63].

2.3 Data Compression

In this section we will use various algorithms to compress data to the minimum and minimize accuracy degradation, in order to improve transmission efficiency and saving storage space.

If we classify the data compression according to the distortion level of the information after compression, then data compression can be classified into lossless compression and lossy compression. In lossless compression, the data is not altered or lost during both compression and decompression, but the compression ratio is generally low. In contrast, lossy compression uses some higher finite distortion data compression algorithms to substantially reduce redundant information, and its compression rate is much higher than lossless compression, but the decompressed data is necessarily lost compared with the source data.

Since integers are generated in the quantization process of model compression, we have learned from previous sections that the quantization process is lossy, and integer weights can be seen as initial weights that have been lossy compressed, so we take lossless compression for integer weights in the subsequent processing. For the floating-point weights we use a lossy compression method with a higher compression rate.

2.3.1 Lossless Compression

Huffman encoding, the main purpose of which is to maximize the saving of storage space for characters (encoding) based on the frequency of use. Usually, some characters appear more frequently than others in a text; Huffman encoding takes advantage of this by re-encoding all the characters that appear in the text, so that the more frequent characters take up less space to achieve compression. Each different integer can be seen as a different character and can be compressed using Hoffman encoding.

Suppose the following integer weights are sent over the network:

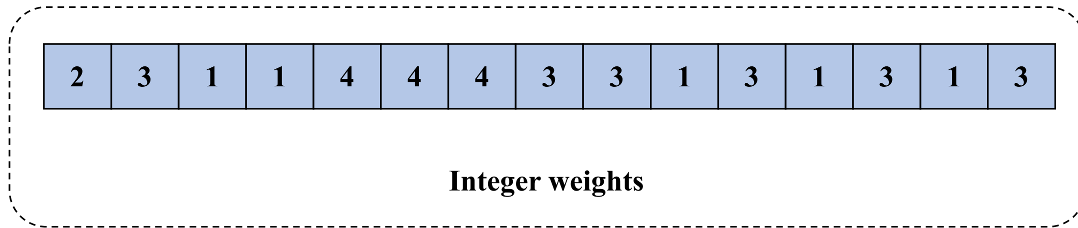


Figure 2.9: A simple example of integer weights

If these integers are stored in INT8 format, then each integer takes up 8 bits, and the string above has 15 integers, so it takes up $15 \times 8 = 120$ bits in total.

When we use Hoffman encoding, we first need to count the frequency of integers, and then arrange them in ascending order, which is shown as follows:

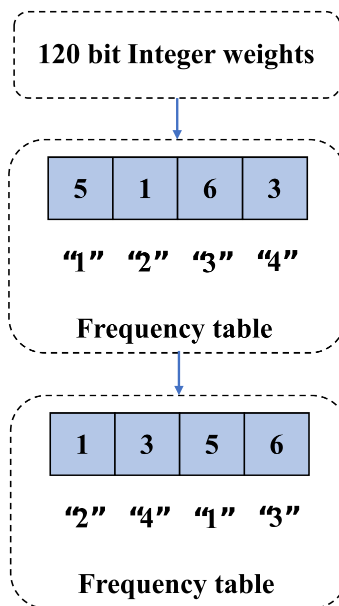


Figure 2.10: Frequency table of integer weights

These characters are then used as leaf nodes to start building a tree, with the following pseudo-code:

Algorithm 1: Hoffman tree building algorithm

Data: The set of n integers C.

Result: Hoffman Tree T

1 $Q \leftarrow C$;
2 $i \leftarrow 0$;
3 **for** i<n **do**
4   new tree node z;
5     $x \leftarrow EXTRACT_MIN(Q)$;
6     $y \leftarrow EXTRACT_MIN(Q)$;
7     $z.left \leftarrow x$;
8     $z.right \leftarrow y$;
9     $z.frequency \leftarrow x.frequency + y.frequency$;
10     $INSERT(Q, z)$;
11     $i \leftarrow i + 1$
12 **end**
13 $T \leftarrow EXTRACT_MIN(Q)$

After the above process, we are able to obtain a Hoffman tree. The tree corresponding to the previous integer weights is as follows:

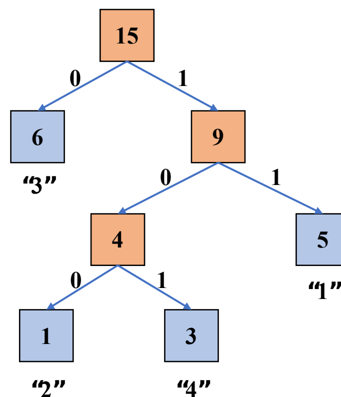


Figure 2.11: Hoffman Tree

When we have a Hoffman tree, we can know the encoding of each integer by simply walking along the tree from the root node.

120 bit Integer weights

Integer	New code	Number of Bits
"1"	11	2x5=10
"2"	100	3x1=3
"3"	0	1x6=6
"4"	101	3x3=9

Sum: 28 bits

Figure 2.12: Calculation of string size after Hoffman encoding compression

On the contrary, if we want to decompress, we only need to construct the Hoffman tree according to the pre-stored frequency table, and then recover the original integer weights according to the Hoffman tree.

2.3.2 Lossy Compression

The ZFP compression algorithm was first proposed by Peter Lindstrom in 2014 [34], and an improved implementation was presented in 2019 [11]. The ZFP algorithm is a lossy compression algorithm that supports 2D, 3D and 4D arrays compression, and can be artificially set to the lose. The following steps are a general description of the operation of the ZFP algorithm, and a detailed description of the steps and loss calculations are described in Peter Lindstrom's 2019 paper [11].

Step 1: Suppose the input is an n-dimensional array, then first define a base block of dimension n and each dimension of size 4. Then if the input array cannot be divided exactly into multiple base block sizes, the bounds of the filled array will be filled in until it can be divided exactly. Example diagram is as follows.

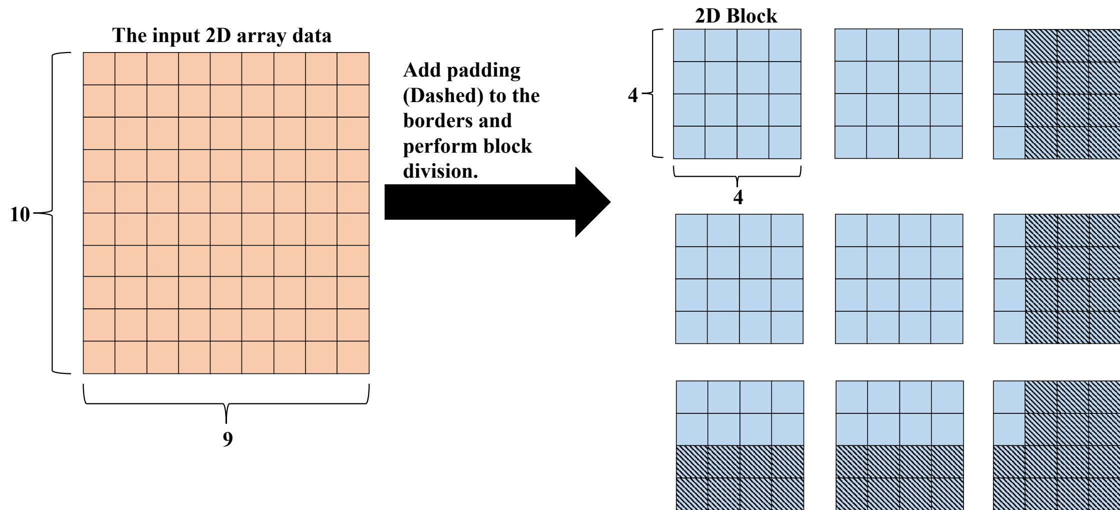


Figure 2.13: ZFP algorithm Step 1, divides the input data into basic blocks.

Step 2: At this time the data is still stored as multiple individual floating-point numbers, but we can convert the floating-point numbers in each base block to Block-Floating-Point form, followed by truncation to 32-bit integers, according to the method proposed by Abhijit Mitra in 2007 [40].

Step 3: This step then processes the output from the previous step. A linear transformation L is defined in the ZFP algorithm, which is a near-orthogonal transform, similar to the discrete cosine transform. Also this transformation operator L can be applied to each dimension separately, so for n-dimensional data, the total transformation equation is as follows. Where $\bigotimes$ is the Kronecker product.

$$L_n = \underbrace{L \bigotimes L \bigotimes \ldots \bigotimes L}_{n} \tag{2.22}$$

Step 4: Sort the coefficients generated by the previous transformations. Each bit plane consisting of 4^n bits is re-encoded and compressed losslessly.

Step 5: The encoder continuously encodes and discards subsequent bit planes once the specified precision is achieved. In the example shown in Figure 2.14, the encoder keeps encoding the input data, and once it detects that the set precision requirements are met, it directly discards the subsequent bit planes (dashed part) and truncates the bit stream.

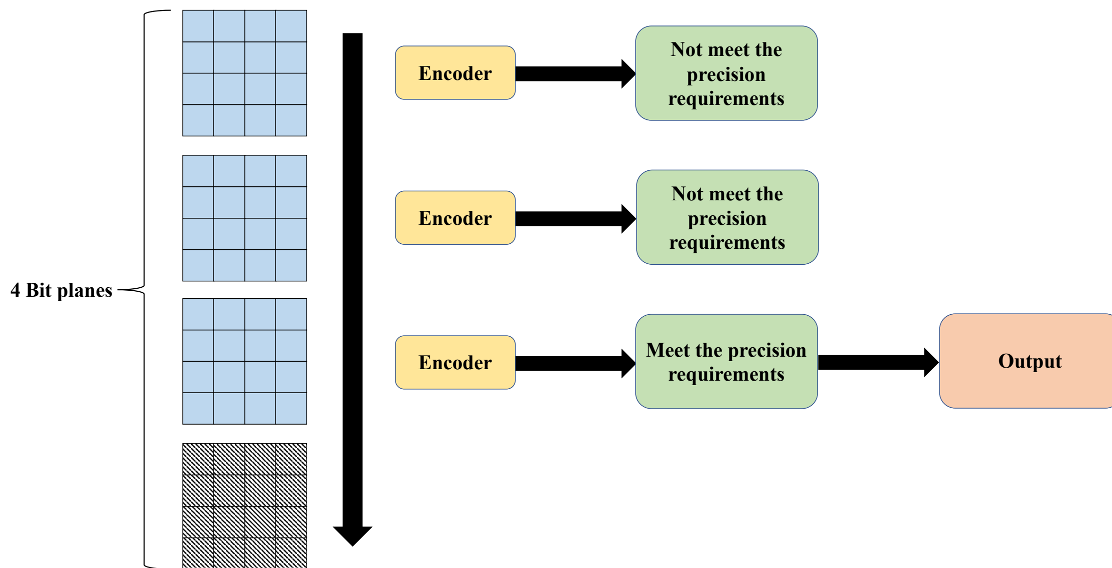


Figure 2.14: ZFP algorithm Step 5, example diagram for 2D input with 4 bit planes.

From the above steps we can see that the loss in the ZFP lossy compression algorithm is mainly related to the number of bit planes kept in the last step. Assuming that the number of bit planes retained in step 5 is β and the maximum exponent in the Block-Floating-Point form [40] is e_{max}, for some precision k, to guarantee b bits of accuracy, then the following equation [11] is available.

$$\beta \geq log_2 \left(\frac{\frac{16}{3}(1+c)}{\frac{(\frac{4}{15})^n \times 2^{-b-e_{max}} - 2^{1-k}}{1+2^{1-k}} - c} \right) \tag{2.23}$$

Inequality 2.23 illustrates that when the precision requirement is set, the number of bit planes retained is expected, which also means that approximately what data will be left is computable.

3

Methods

In this chapter, we will discuss the workflow and implementation details of model compression and data compression, and how we combine them together to build our hybrid compression solution.

3.1 Model Compression

3.1.1 AI framework

The flourishing of artificial intelligence in academia and industry is inseparable from AI software frameworks, such as the well-known TensorFlow [1] and PyTorch [46]. For model compression, these two popular frameworks have different supports.

TensorFlow Lite [60] is needed to deploy the TensorFlow model after model compression, and the supported hardware platforms include Android and iOS devices, embedded Linux, and microcontrollers. Most of these platforms are based on ARM architecture processors and Google Tensor Processing Unit (TPU). PyTorch does not provide quantized operator implementations on CUDA, the model quantized by PyTorch can only be deployed on the x86 and ARM CPU platform at present [47]. If we want to deploy the compressed model on Nvidia GPUs, first PyTorch-Quantization toolkit [43] is needed to compress and convert the model to ONNX [44] format, and then send it to Nvidia TensorRT [42] for deployment, which is a tedious and inconvenient process.

Considering the integration of model compression and data compression and the hardware resources we have, we chose PaddlePaddle [38]. Like Google's TensorFlow and Facebook's PyTorch, the Baidu open source PaddlePaddle provides the tools, services and resources for rapid and large-scale implementation of deep learning for all levels of software developers. PaddleSlim [6] is PaddlePaddle's open source library for model compression, supporting various methods such as pruning, quantization, knowledge distillation, network architecture search, etc. By utilizing PaddleSlim, the quantized parameter value (e.g. INT8) can be read directly after the model quantization, which can be perfectly combined with our data compression work. In addition, the compressed model does not require any transforms and can be deployed straight on Nvidia GPUs. The above advantages are the reasons why we chose PaddlePaddle as the AI framework.

3.1.2 Pruning

In this section, we will discuss the process of network pruning, and the pruning strategy. In PaddlePaddle, the workflow of pruning is very clear, as shown in figure 3.1. We implement the network pruning based on the pre-trained model. Pruning will change the network structure, the pruned network has trouble maintaining the previous performance and usually suffers from a loss of prediction accuracy, so fine-tuning is necessary. In the fine-tuning stage, we retrain the model to recover the accuracy, and finally save the model with the highest accuracy to complete the pruning process. It is important to note that the choice of using a pre-trained model as a start is not mandatory, and the researchers in [36] point out that although retraining based on a pretrained model may converge faster, the accuracy of a pruned model trained from scratch can also achieve a comparable performance.

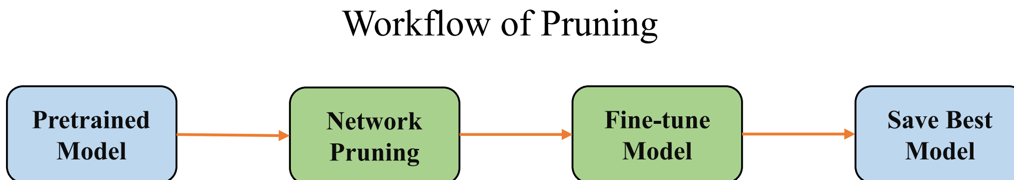


Figure 3.1: Pruning workflow using PaddlePaddle.

In PaddlePaddle, two pruning strategies are available to accomplish filter-wise pruning. They are custom ratio pruning and pruning based on sensitivity analysis.

Within the custom ratio pruning strategy, uniform ratio methods and non-uniform ratio methods are provided. As shown in Figure 3.2, in the uniform ratio approach, the pruning ratio of each group is the same, we just need to determine a value P, then for each group there will be $P\%$ of the filters removed from the first convolutional layer and $P\%$ of the kernels removed from the second convolutional layer. It should be noted that the convolutinal layer which is in the middle of figure belongs to both Group M and Group N, and will have $P\%$ of filters and kernels pruned together. In contrast, in the non-uniform ratio approach, different pruning rates can be set to different groups manually. For example, we set $Q\%$ for Group M and $P\%$ for Group N. With this strategy, the pruning process can be executed quickly and without additional overhead. However, there is no guide to advise us what the optimal pruning ratio is for each layer, and the pruning ratio we set may appear to be unreasonable. This dilemma probably has many drawbacks, such as difficult fine-tuning or excessive loss of network accuracy.

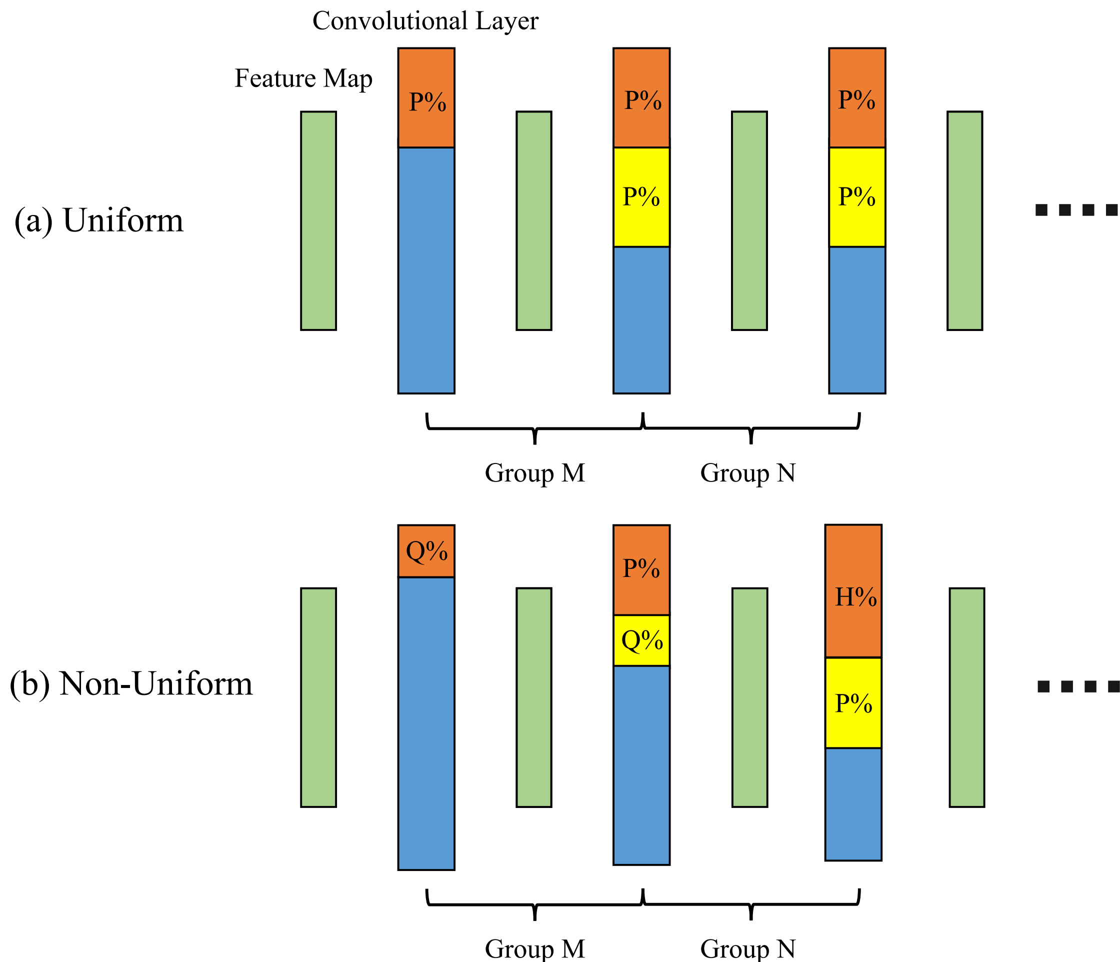


Figure 3.2: Comparison of uniform and non-uniform ratio method. The orange rectangle in the convolution layer represents the filter to be pruned, where the yellow rectangle represents the kernels to be pruned.

Compared to the custom ratio pruning strategy, the sensitivity analysis strategy gives us a guideline. PaddleSlim defines sensitivity in such a way that we prune each layer independently at different ratios and evaluate the accuracy loss, if some layers have a large accuracy loss when only a small number of filters are pruned, then they have a higher sensitivity. Among the multiple convolutional layers of a network, we make the sensitivity represent the importance of the convolutional layer, the more sensitive the layer is, the more important it is, and the more important layers will be pruned with relatively few filters. Under the guideline of sensitivity analysis, we can avoid setting the pruning rate arbitrarily, which may help us get higher accuracy after pruning.

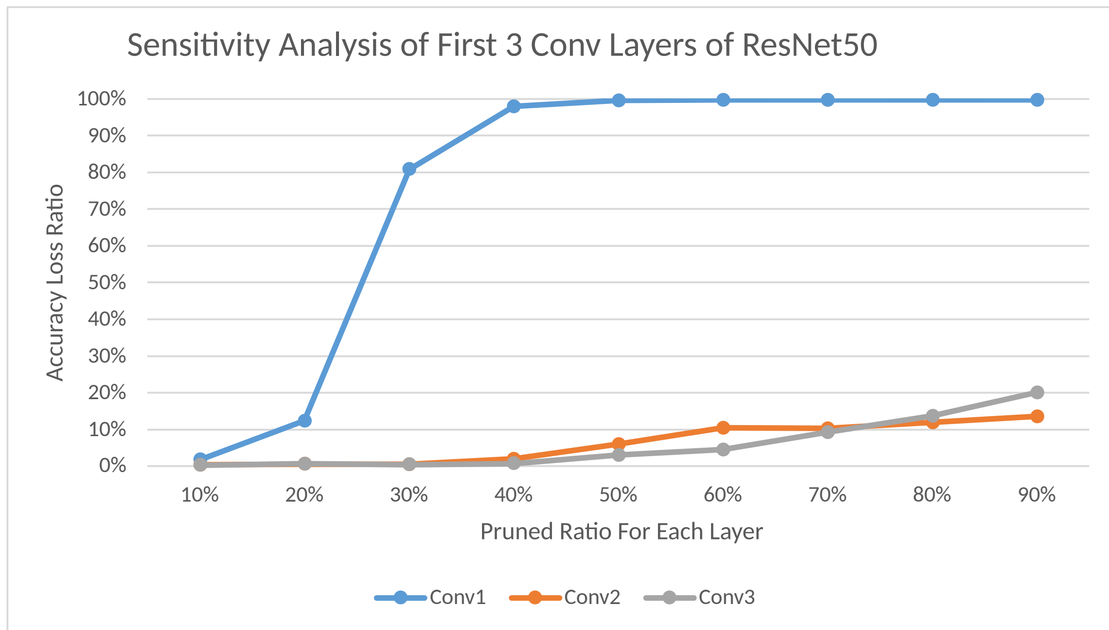


Figure 3.3: An example of sensitivity analysis of pruning on ResNet50 with first 3 convolutional layers (Conv). The value of the horizontal coordinate is the different pruning ratios, and the value of the vertical coordinate is the accuracy loss ratio. The base accuracy of ResNet50 is 75.42%, if we prune 20% of filters in Conv1, the accuracy loss of the whole model is $75.42\% * 12\% = 9.1\%$, then the accuracy of this model changes to $75.42\% - 9.1\% = 66.32\%$.

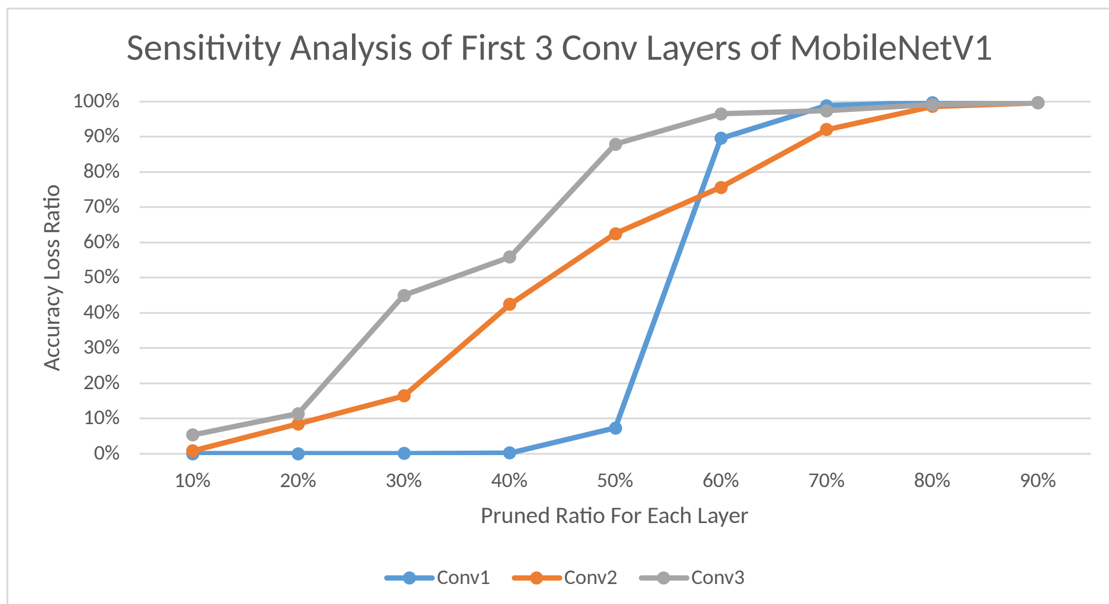


Figure 3.4: An example of sensitivity analysis of pruning on MobileNetV1 with first 3 convolutional layers (Conv). The value of the horizontal coordinate is the different pruning ratios, and the value of the vertical coordinate is the accuracy loss ratio.

Before performing the sensitivity analysis, the importance of filters in each convolutional layer are sorted by the L1-norm metric (mentioned in Section 2.2.1.2), and unimportant filters are the primary choice to be pruned. In the process of sensitivity analysis, we first prune 10% of the filters of first layer, and record the accuracy loss in the validation set. Then, we reset the model back to the initial state with original accuracy, and we remove 20% of the filters from this layer then record accuracy loss. And so on, we prune from 30% to 90%, the entire sensitivity analysis results of first layer can be obtained. In the same way, the sensitivity analysis result of all layers can be obtained. However, to complete the sensitivity analysis of one model, a number of accuracy evaluations will be executed on the validation set. Suppose we perform sensitivity analysis for a network with N convolutional layers using C different pruning ratios. Then $C * N$ times of accuracy evaluations will be executed. If we perform a sensitivity analysis on a model with a large number of convolutional layers, the overhead will be huge. Fortunately, we only need to complete the sensitivity analysis once for each model, and we then can use the results of it as many times as we want.
As shown in Figure 3.3, this is the result of a sensitivity analysis for the first three convolutional layers of ResNet50. We can observe from this figure that pruning a small part of $Conv1$ brings a dramatic loss of accuracy, and we can tell that $Conv1$ is more sensitive than $Conv2$ and $Conv3$, and such a convolutional layer has significant importance, so we should prune its filter less or not at all. In contrast, $Conv2$ and $Conv3$ appear to be insensitive, which means that they are not much important, and we can remove more filters within such insensitive convolutional layers. We can also observe that the $Conv1$ layer in Figure 3.4
PaddleSlim provides an automatic pruning method based on sensitivity analysis. When we set a parameter P%, it will prune different ratios of filters for different layers based on the results of sensitivity analysis, which will cause P% of FLOPs to be pruned with minimum loss of accuracy. In the later results chapter, we use this approach.

3.1.3 Quantization

In our thesis work, we apply two popular methods for model quantization. One is Quantization Aware Training (QAT), which requires model training, and the other is Post Training Quantization (PTQ), which requires only a small amount of time for calibration.

3.1.3.1 Quantization Aware Training

QAT is a method for simulating quantization effects in model training. The workflow of QAT is illustrated in Figure 3.5. Following suggestions from [28], we start with a pre-trained model with 32-bit floating point weights to perform quantization, which can bring higher accuracy and require fewer training epochs. To simulate quantization, PaddleSlim inserts pairs of fake quantization operations before fine-tuning. After the fake quantization operations are inserted into the DNNs model, we begin to fine-tune the model, which requires the entire training data set. When

the fine-tuning is finished, we will convert the model with the highest accuracy into an inference format, by removing some of the fake quantization operations.

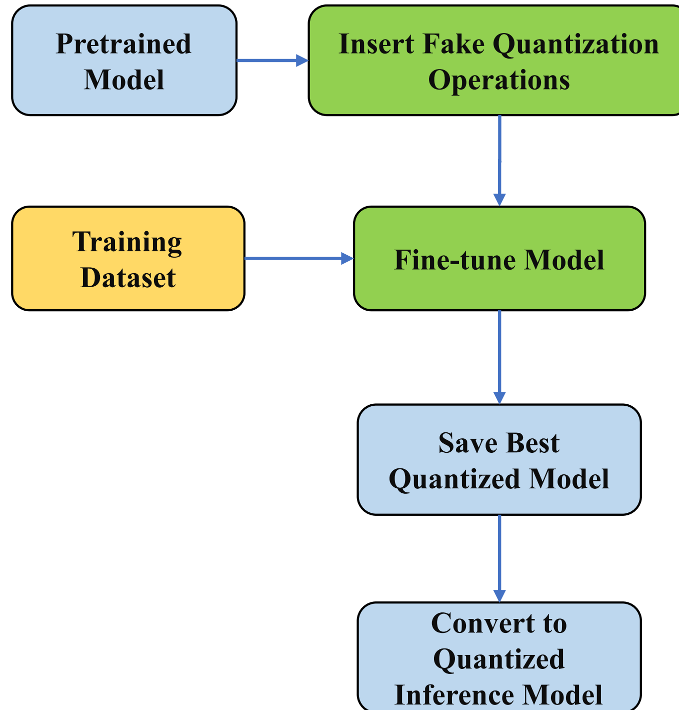


Figure 3.5: Workflow of the Quant Aware Training (QAT).

To illustrate more clearly, we describe QAT from the perspective of the model structure in the following content. As shown in Figure 3.6, they are a small part of DNNs model. Compared to the original model structure on the left, the structure on the right has two pairs of green fake quantization operations inserted. The quantization parameters used for the calculation (e.g. scale and zero point) are stored in these operations. The weights and activations in FP32 format are processed into INT8 format after the *Quant* operation, and they will be converted back to FP32 format after the *DeQuant* operation. During the quantization/de-quantization process, the errors caused by quantization are introduced, and the FP32 data output by the *DeQuant* operation is no longer the original data. For the rest of the operations in the network, the FP32 data is still used for training. It is worth noting that during the back-propagation process in training, the learned weight parameters will be updated to the initial FP32 format data. At the same time, the quantization parameters will also be adjusted. After training such a network, the weights and quantization parameters are tuned to the appropriate values for the task at hand, minimizing the accuracy degradation of the quantized model.

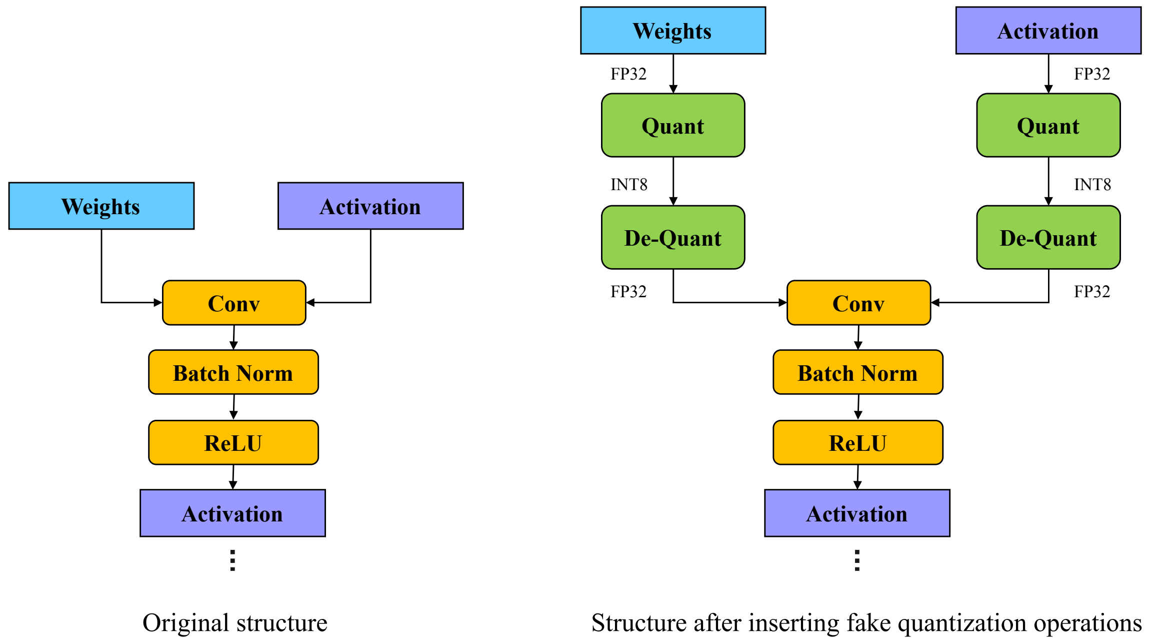


Figure 3.6: Network structure before and after inserting the fake quantization operations. Quant and De-Quant represent quantization and de-quantization operations respectively. Conv denotes a convolutional layer.

3.1.3.2 Post Training Quantization

Compared to QAT, PTQ does not require training to quantify the model. As shown in the Figure 3.7, this is the workflow of PTQ. Before quantization, we only need to use a small part of the validation dataset, known as the calibration set, to complete the calibration. Calibration is applied to calculate the quantization parameters for the activations. In the process of calibration, we perform network inference on the calibration set using a pre-trained model, simultaneously sampling the data distribution in the activations. In our work, based on the sampled activation data, the quantization parameters for activation will be determined with percentile method. Meanwhile, the quantitative parameters for the weights are obtained using the maximum absolute value method. After that, the corresponding quantization operations are inserted into the network and we can acquire a network for inference.

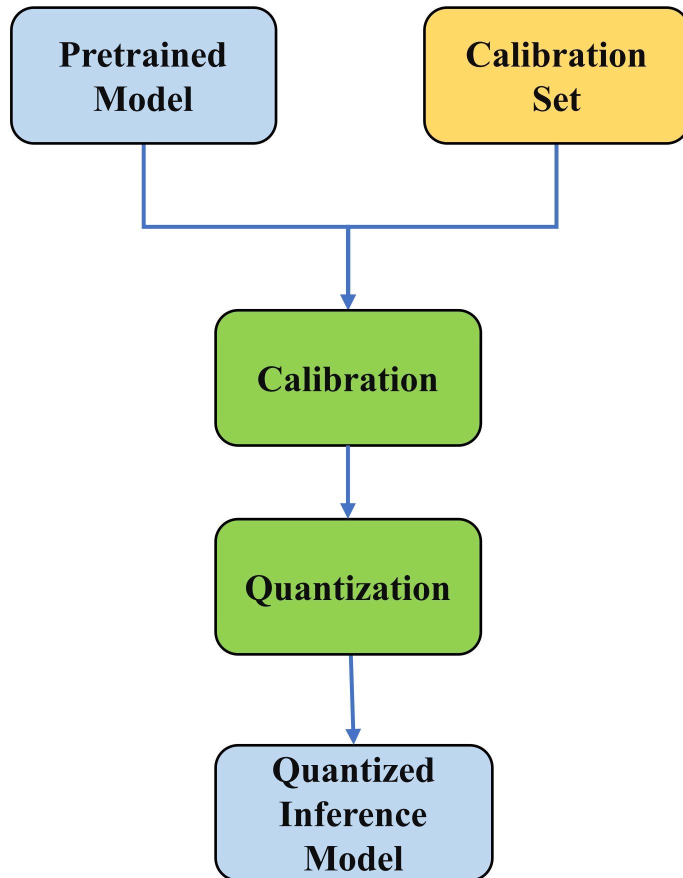


Figure 3.7: Workflow of the Post Training Quantization (PTQ).

The PTQ model structure is slightly different from QAT, as shown in the figure 3.8. In this figure we can observe that the quantized weight are stored in INT8 format, and INT8 computation will be carried out directly in the Conv layer. Then, the $De - Quant$ operation is still needed in order to complete the subsequent calculations. With the PTQ method, no training process is involved to finish the quantization process, but the prediction accuracy of the quantized model is not as good as QAT. It is worth noting that we will utilize PaddleInference, which is compiled with TensorRT, to measure the model's inference speed. TensorRT will help to remove $Quant$ and $De - Quant$ operations, so these quantization operators will not be present in the actual inference computation.

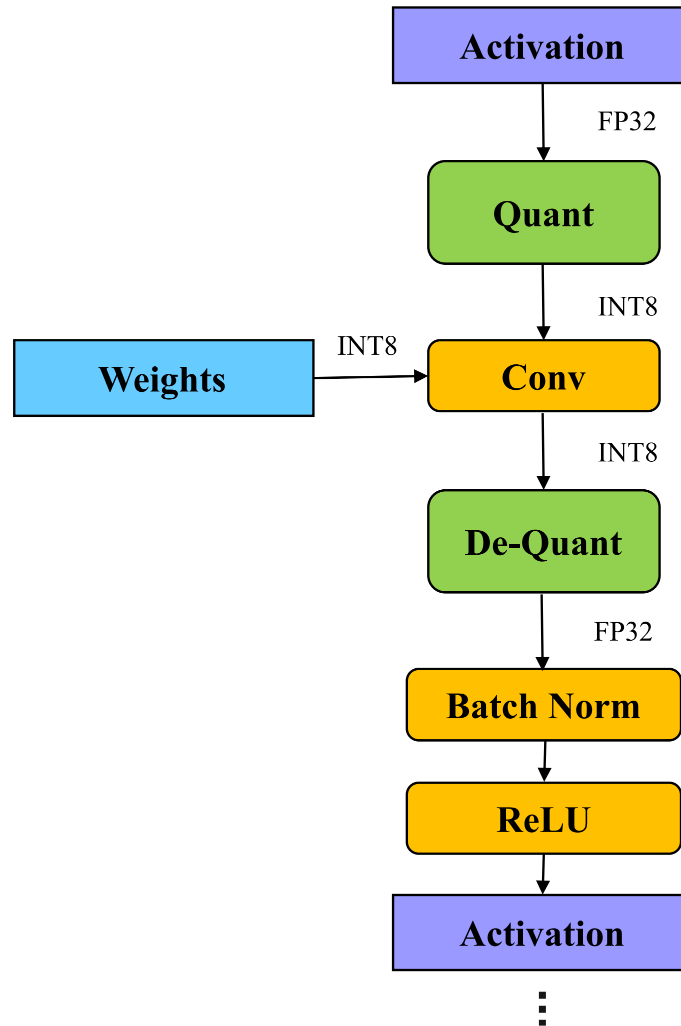
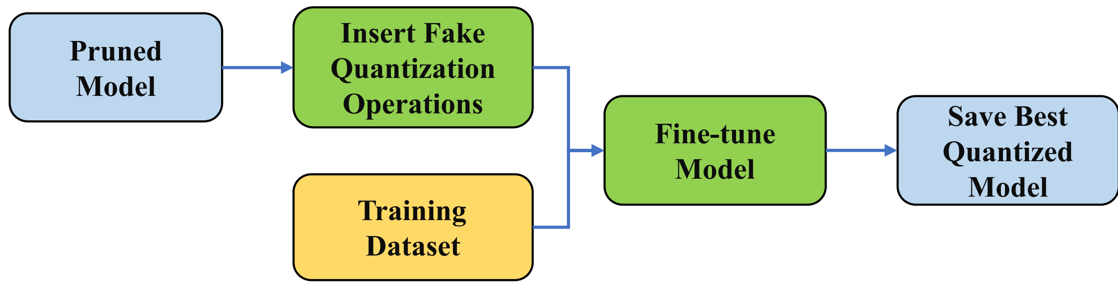


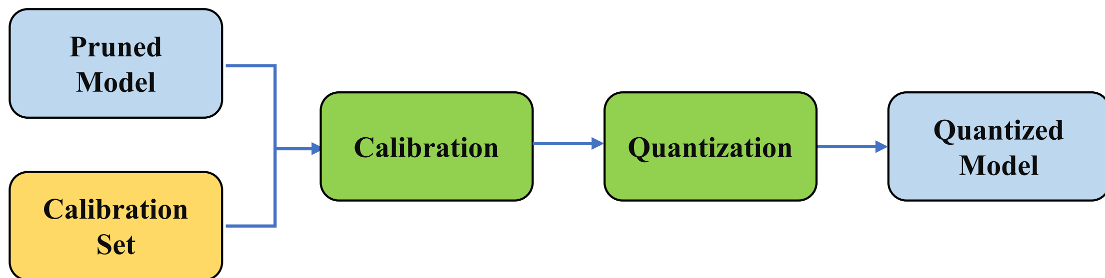
Figure 3.8: Network structure of PTQ model. Quant and De-Quant represent quantization and de-quantization operations respectively. Conv denotes a convolutional layer.

3.1.4 Model Compression Pipeline

In order to obtain the ultimate compression ratio, we combine pruning and quantization to construct a model compression pipeline. With the generalization of the structured pruned model, it is convenient to implement this joint-way compression in PaddlePaddle. As shown in the Figure 3.9, we only need to replace the pre-trained model with the pruned model and then perform quantization to complete the combination of pruning and quantization.



Pruning and QAT



Pruning and PTQ

Figure 3.9: Workflow of model compression pipeline. This pipeline is constructed by the combination of pruning and quantization.

3.2 Data Compression

First, we analyze the overall storage structure of the model. The specific situation is as follows:

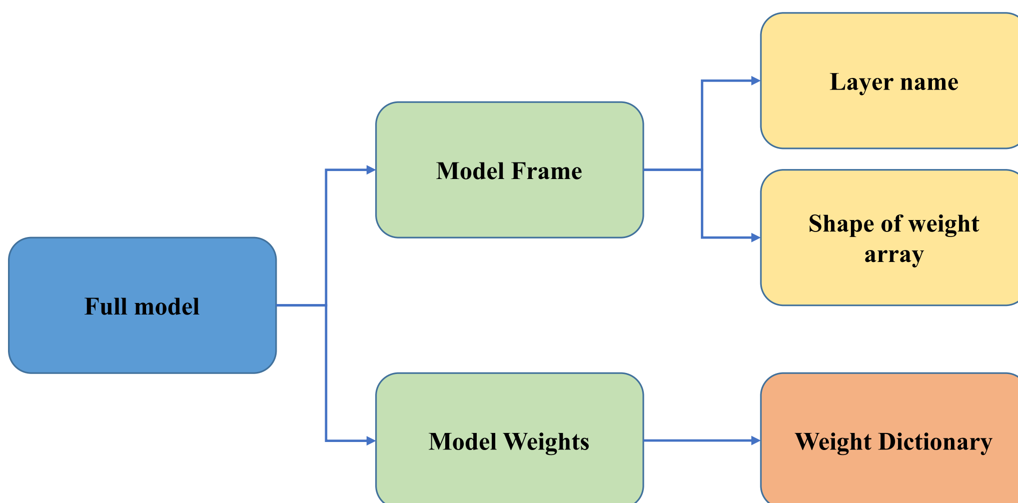


Figure 3.10: The overall storage structure of the model.

The file size of Model Frame accounts for a small proportion of the total file size, and is non-data content, so we mainly deal with Model Weights subsequently.
Then we analyze the data structure of weight storage after quantization.

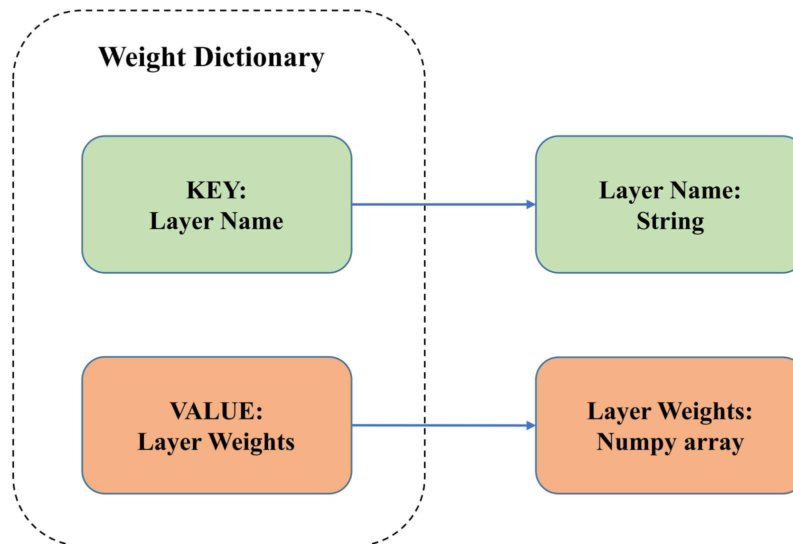


Figure 3.11: The specific data structure of weight storage.

From Figure 3.11, we can know that the model weights are stored in the form of a dictionary in PaddlePaddle.
And in order to connect the previous quantitative operations, we also need to analyze the types of weight data in the model. The specific pseudocode is as follows:

Algorithm 2: Analysis of weight data types in the model.

Data: Model weight dictionary D
Result: Weight category result in model R

```
1 foreach key k of D do
2     temp ← D[k];
3     if temp mod 1 = 0 then
4         R[Integer] ← R[Integer] + Size(temp)
5     else
6         R[Float] ← R[Float] + Size(temp)
7     end
8 end
```

After analysis we know that there are two parts in the quantized model, the integer weight and the floating point weight. Due to the difference in compression processing between integer and floating point numbers, we have to split the model weights into two parts. And because integers are discrete distribution and also due to the limitation of computer storage, it is difficult to keep the loss within the range we want if we use lossy compression algorithm for integers. So for integers we use lossless compression. For floating point numbers, there are no such limitations, so we prefer to use a lossy compression algorithm which can bring higher compression ratio. The main workflow is as follows:

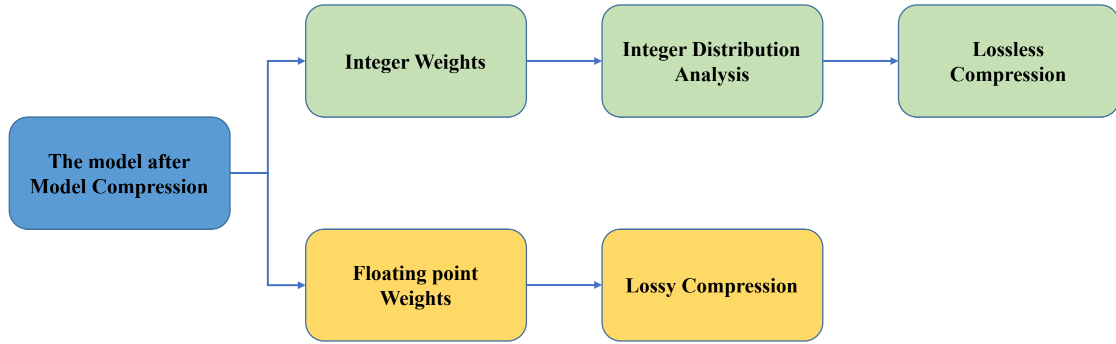


Figure 3.12: Data Compression Workflow

Of course, there are other elements in the model, such as activation functions like ReLU, and they are stored in the Model Frame for storage, which corresponds specifically to the operators in the library. The storage space occupies a very small percentage, so it is not processed in the subsequent processing.

3.2.1 Lossless compression of integer weights

First, we analyze the distribution of integer weights. The specific pseudocode is as follows:

Algorithm 3: Integer weight distribution analysis

Data: Integer weight dictionary D

Result: Integer weight distribution array A in the model

```
1 foreach key k of D do
2     temp ← D[k];
3     foreach integer i of temp do
4         A[i] ← A[i] + 1;
5     end
6 end
```

After analyzing the integer distribution, we found that the integer distribution in the model is not uniform. So we can re-encode the integers in the matrix so that the more frequent integers occupy less space for compression. Here, we choose the Huffman coding compression method, and the list generated by the previous integer frequency analysis can be used to generate the Huffman tree.

As shown in Figure 3.10, in the actual storage, the model frame is stored separately from the weight file, which contains the shape information of each layer weight, so we can flatten all the integer weights and process them uniformly. And when we want to decompress and restore, we only need to use the shape information of each layer of weights in the model frame file to restore the original shape.

Therefore the pseudo code of the Huffman compression part is as follows:

Algorithm 4: Compress integer weights using the Huffman coding algorithm

```
Data: Integer weight dictionary D, Integer weight distribution list A
Result: Compressed binary code C
1  foreach key k of D do
2  |  temp ← D[k];
3  |  foreach integer i of temp do
4  |  |  String S ← S + "i";
5  |  end
6  end
7  C ← Huffman_encode(S, A);
   /* Add padding to make C minimal storage.  The length of C is
      denoted by l.                                              */
8  l ← Length(C);
9  if l mod 8 ≠ 0 then
10 |  C ← C + "0" × (8 − l mod 8);
11 end
12 C ←Serialize C to int8 binary code;
```

Finally we just need to write C and integer weight distribution list A to the file to get the compressed file. When we want to decompress the encoded file, we only need to use the shape information of the weight in the model frame to reshape after Huffman decoding.

3.2.2 Lossy compression with floating point weights

When we are dealing with floating point numbers, lossy compression can be employed to obtain greater compression ratios. However, since the weights are lossy compressed, the decompressed data must be different from the previous weight data, which will lead to changes in accuracy, so we need to analyze the relationship between loss parameter settings in lossy compression and the accuracy of the final model.

At the same time, in order to avoid interference between different layers, we do not use the same integer weights to combine all weight matrices, but process the weight data at the layer granularity.

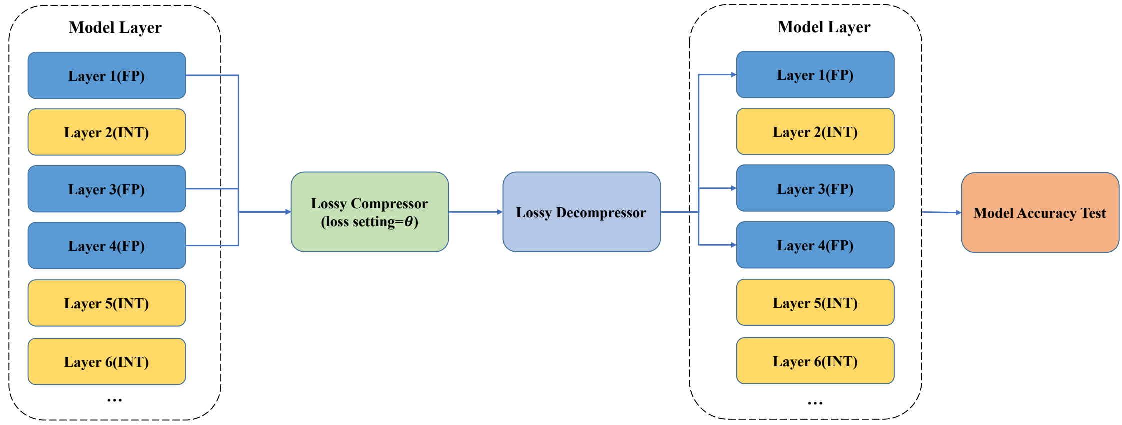


Figure 3.13: Workflow of analyzing the relationship between loss setting parameters θ and final model accuracy.

After the above tests, we realized that there are certain "sensitive" layers in the model, for which even a small loss parameter setting can bring about a huge change in the model accuracy. Therefore, on this basis, we adopted a similar approach to sensitivity pruning, first performing sensitivity analysis on the model, and then compressing the model according to the results of the sensitivity analysis.

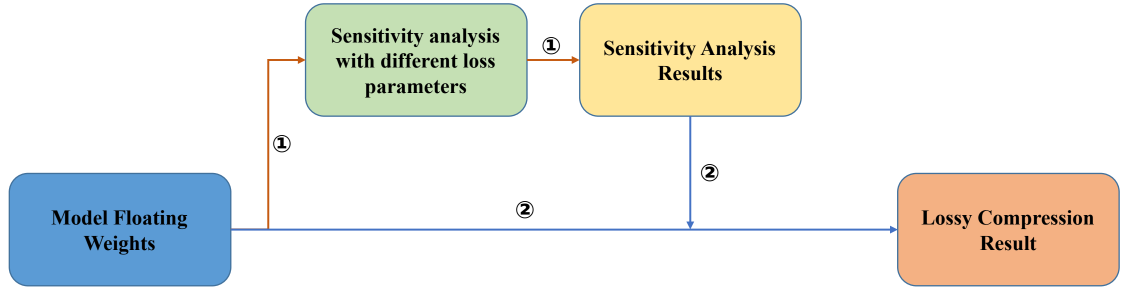


Figure 3.14: Lossy compression process for model floating point weights.

As shown in Figure 3.14, in order to perform lossy compression, we first perform a sensitivity analysis. The pseudocode of the sensitivity analysis algorithm is as follows:

Algorithm 5: Sensitivity analysis algorithm for lossy compression.

Data: Floating point weight dictionary D, List of loss parameters to be analyzed L

Result: Results of Sensitivity Analysis A

```
1 foreach key k of D do
2     temp ← D[k];
3     foreach Loss parameters θ of L do
4         D[k] ← Lossy_Compressor(D[k], θ);
5         D[k] ← Lossy_Decompressor(D[k]);
6         A[k][θ] ← Accuracy of model;
7     end
8     D[k] ← temp;
9 end
```

When we have the sensitivity analysis results, the final lossy compression can be performed. In terms of the compression strategy, we adopt a greedy idea to find the maximum loss setting that matches the maximum allowable accuracy reduction value set by the user. The specific compression strategy adopted is as follows:

Algorithm 6: Lossy compression with floating point weights

Data: Floating point weight dictionary D, The maximum allowable accuracy reduction value set by the user T, Results of Sensitivity Analysis A, Baseline accuracy of the model $Base$

Result: Lossy compression results C

```
1 foreach key k of D do
2     foreach Loss parameters θ of A[k] do
          /* The loss parameters traversed from large to small.
             */
3         if Base − A[k][θ] ≤ T then
4             C[k] ← Lossy_Compressor(D[k], θ)
5         else
6             C[k] ← Lossy_Compressor(D[k], 0)
7         end
8     end
9 end
```

3.3 Hybrid Compression

We create hybrid compression by combining model compression and data compression. The specific workflow is as follows:

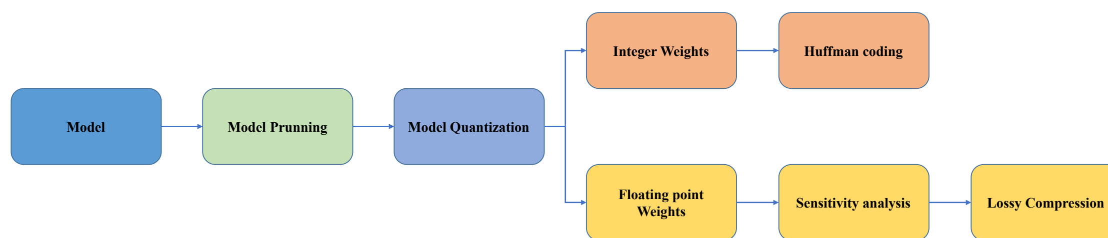


Figure 3.15: Workflow of Hybrid Compression.

After model compression, the values of model weights are divided into floating point and integers. We then apply the different data compression methods to compress these values. The model compressed by hybrid compression can obtain the highest compression ratio.

4

Results

In this chapter, we describe the experiment results of model compression, data compression and hybrid compression.

4.1 Experimental Setup

4.1.1 Experimental Models

In our thesis work, we mainly focus on DNNs for image classification task. We choose MobileNetV1 [24] and ResNet50 [21] for model compression. Some other representative classification models are also included in the data compression. In the final hybrid compression stage, MobileNetV1 and ResNet50 are selected for experimentation. Pre-trained models for these networks are provided by PaddleClas [5].
MobileNetV1 is built on a streamlined architecture using the depthwise separable convolutions for efficient inference on resource limited hardware. Depth-separable convolution separates a standard convolution into depthwise convolution and pointwise convolution, which help to reduce considerable parameters. ResNet50 is constructed from numerous residual blocks. In the residual block, the output feature map can be added with the input by the shortcut path, which can solve the degradation problem in DNNs.

Table 4.1: Model information of MobileNetV1 and ResNet50

Model	MobileNetV1	ResNet50
Conv Layers	27	53
Total Parameters	4253864	25610152
FLOPs	0.58 GFLOPs	4.11 GFLOPs

Table 4.1 shows the number of convolutional layers, total parameters and FLOPs of MovileNetV1 and ResNet50. The data are provided by PaddlePaddle.

4.1.2 Platform

A Nvidia Tesla V100 GPU server from Baidu AI Studio [3] provides the computing power for the DNNs training work. The inference performance is measured on a RTX 3080Ti on our personal computer. For the software, we choose PaddlePaddle

v2.2.2 as the deep learning framework. Model compression and model inference are empowered by two toolkits, PaddleSlim [6] and PaddleInference [4] respectively.

4.1.3 Dataset

The dataset we choose is the validation set of ILSVRC-2012 [50], which contains 50000 (50K) images of 1000 classes. We split this dataset into a training set containing 40000 (40K) images and a validation set containing 10000 (10K) images.

4.1.4 Training Settings

We select the same training hyperparameters for all models. The learning rate is set to 0.001 for pruning and 0.01 for quantization with cosine decay [37] scheduler. We retrain 90 and 30 epochs for the pruned model and the quantized model, respectively.

4.1.5 Metrics

In our experiments, we evaluate our solution in terms of model prediction accuracy, model storage size and model inference speed by below metrics.

- **Acc1.** There are 1000 different categories in ImageNet dataset. In the output of the model prediction, there are 1000 probabilities of ranking from high to low. When the output with the highest probability is exactly the expected answer, we call it the correct result. We define $Acc1$ as:

$$Acc1 = \frac{Number_of_correct_results}{Number_of_total_prediction_data} \tag{4.1}$$

- **Acc1 Drop.** Compression can have an impact on the prediction accuracy of the model, for example, by suffering a loss of accuracy. We use the $Acc1$ *Drop* to denote the decrease of accuracy, which is the value of the base accuracy minus the accuracy after compression.
- **Size.** We use $Size$ to record the storage size of the model on the hard drive.
- **Compression Ratio(CR).** The compressed size of a model divided by the original size of the model is defined as the compression ratio (CR). The mathematical expression of CR is:

$$CR = \frac{Original_size}{Compressed_size} \tag{4.2}$$

- **Perf.** We use $Perf$ to denote the inference performance of the model. Here, performance refers to the time consumed by the model to infer a batch size of data.
- **FPS.** Frame per second (FPS) is the inverse of inference time. When we compare the performance of edge devices, we use this metrics to show their difference.
- **Speedup.** Model compression increases the inference speed of the model. The *Speedup* can be obtained as follow:

$$Speedup = \frac{Original_perf}{Compressed_perf} \tag{4.3}$$

4.2 Model Compression

We implement pruning and quantization respectively, then combine these model compression techniques together into a compression pipeline and experiment on MobileNetV1 and ResNet50.

4.2.1 Pruning

We start from the ILSVRC-2012 pre-trained models provided by PaddlePaddle. To get the baseline of accuracy, we firstly evaluate these models on our 10K validation set. Han [20] reports that up to 3× original training time is needed for retraining the pruned models. In our experiments, we try different combinations of hyperparameters to retrain the pruned models, which has taken us a lot of time, but we do not get the same results as the official releases [45]. Because our focus is not on the retraining, for every model, we set the same hyperparameters. We retrain the pruned models for 90 epochs on our 40K training set and save those models that get the best top-1 accuracy on the 10K validation set.

In our experiments, MobileNetV1 and ResNet50 are pruned using FPGM and L1-Norm algorithms with different pruning ratio. The effect of pruning are demonstrated from three aspects: model accuracy, model size and inference performance in Table 4.3, Table 4.4 and Table 4.5. It should be noted that the model accuracy after pruning provided by PaddlePaddle is much better than ours. Due to the limitation of retraining, it is difficult for us to retrain the model to higher accuracy, and the accuracy of the model after pruning thus can not reflect the actual performance of the algorithms.

We first compare two pruning strategies described in the Section 3.1.2, uniform custom ratio pruning and pruning based on the sensitivity analysis. In the Strategy column of Table 4.2, the Uniform-30 means that we prune 30% of the number of filters in each layer, while Sensitive-50 denotes that we plan to prune 50% of FLOPs of the whole model using sensitivity analysis. The Pruned Ratio column shows the total FLOPs which are pruned in our experiment of a model. For instance, when we use Uniform-30 L1-Norm method on MobileNetV1, it will prune 30% of the filters in each layer, which results in a reduction of 47.78% of the total number of FLOPs. From Table 4.2 , we can observe that for different models and different methods, Pruned Ratio given by Sensitive-50 is slightly higher than Uniform-30, while the Acc1 is also higher than Uniform-30. The above demonstrates that by using the pruning strategy based on sensitivity analysis, we can obtain higher accuracy while pruning more parameters. Therefore, the effect of sensitivity analysis pruning is better than the uniform method. Therefore, we chose the sensitivity analysis strategy for all the pruning experiments.

Table 4.2: Acc1 comparision of different pruning strategy. Method and Strategy denote different pruning algorithms and strategy respectively. The Pruned Ratio is the percentage of FLOPs pruned.

Model	Method	Strategy	Pruned Ratio	Acc1
MobileNetV1	L1-Norm	Uniform-30	47.78%	63.17%
MobileNetV1	L1-Norm	Sensitive-50	49.90%	63.19%
MobileNetV1	FPGM	Uniform-30	47.78%	63.70%
MobileNetV1	FPGM	Sensitive-50	49.90%	63.72%
ResNet50	L1-Norm	Uniform-30	48.93%	66.78%
ResNet50	L1-Norm	Sensitive-50	50.01%	67.03%
ResNet50	FPGM	Uniform-30	48.93%	66.88%
ResNet50	FPGM	Sensitive-50	50.01%	67.78%

Table 4.3: Comparison of accuracy of pruned MobileNetV1 and ResNet50 with baseline. In this table, the pruning strategy we used is base on sensitivity analysis. So the 30 and 50 denotes that we plan to prune 30% and 50% of FLOPs of the model.

Model	Method	Base Acc1	Pruned Acc1	Acc1 Drop ↓
MobileNetV1	L1Norm-30	70.07%	67.58%	2.49%
MobileNetV1	FPGM-30	70.07%	68.03%	2.04%
MobileNetV1	L1Norm-50	70.07%	63.19%	6.88%
MobileNetV1	FPGM-50	70.07%	63.72%	6.38%
ResNet50	L1Norm-30	75.42%	71.29%	4.13%
ResNet50	FPGM-30	75.42%	72.02%	3.40%
ResNet50	L1Norm-50	75.42%	66.70%	8.72%
ResNet50	FPGM-50	75.42%	67.78%	7.64%

Table 4.3 shows the model accuracy under different pruning algorithms. As shown in the Method column, L1-Norm and FPGM methods are evaluated with two pruned ratios, 30% and 50% with sensitivity analysis respectively. We observe that under the same pruned ratio, FPGM algorithm shows less Acc1 Drop, a phenomenon that is present in both MobileNetV1 and ResNet50. So the FPGM is the better one, and we mainly focus on this algorithm in subsequent experiments.

Table 4.4: Comparison of model size of pruned MobileNetV1 and ResNet50 with baseline. The unit of model size is megabytes (MB).

Model	Method	Base Size	Pruned Size	CR
MobileNetV1	FPGM-30	17.0	13.4	1.27
MobileNetV1	FPGM-50	17.0	11.0	1.55
ResNet50	FPGM-30	102.4	84.0	1.22
ResNet50	FPGM-50	102.4	62.1	1.65

As shown in Table 4.4, in each row, the size of the pruned model is obviously reduced. It should be reminded that we only prune the filters of each convolutional layer, and the value of CR is not linearly proportional to the pruned ratio.

Table 4.5: Comparison of performance of pruned MobileNetV1 and ResNet50 with baseline. Performance is measured using PaddleInference, the batch size is set to 1, the unit is milliseconds.

Model	Method	Base Perf	Pruned Perf	Speedup
MobileNetV1	FPGM-30	0.6553	0.5854	1.12
MobileNetV1	FPGM-50	0.6553	0.5251	1.25
ResNet50	FPGM-30	2.2217	2.0834	1.07
ResNet50	FPGM-50	2.2217	2.0027	1.11

The result of performance is the time consumed to infer one batch size of data. The pruned model has fewer filters in convolutional layers and less computation workload. We can observe that the speedup of inference performance in Table 4.5. During model inference, PaddleInference will provide computation optimization for matrix multiplication. After pruning the model, the number of filters may become odd, which may lose some chance to hit the optimization. For example, if a special optimization O is applied for $A = [1024, 1024]$ and $B = [1024, 256]$, but when A is clipped to $[679, 1024]$, it is possible that the optimization O will not be implemented. In order to get better performance, We fine tune the pruning algorithms and make the number of filters in the pruned layer be a multiple of 16.

Table 4.6: The fine-tune resluts of MobileNetV1 and ResNet50 using FPGM algorithm. The A16 in Method column represents the number of filters is aligned to 16. The Perf is measured when batch size = 1.

Model	Method	Acc1	Size	Perf
MobileNetV1	FPGM-30	68.03%	13.4	0.5854
MobileNetV1	FPGM-30-A16	68.44%	13.5	0.5856
MobileNetV1	FPGM-50	63.72%	11.0	0.5251
MobileNetV1	FPGM-50-A16	63.36%	11.0	0.4894
ResNet50	FPGM-30	72.02%	84.0	2.0834
ResNet50	FPGM-30-A16	72.95%	84.6	2.0574
ResNet50	FPGM-50	67.78%	62.1	2.0027
ResNet50	FPGM-50-A16	68.22%	61.9	1.9026

The fine tuned results are show in Table 4.6. We can observe that the A16 models has slight higher accuracy than model without fine tune, except for MobileNetV1-FPGM-50-A16. The model size after fine tune has increased a bit, but the size of ResNet50-FPGM-50-A16 has decreased. The inference performance has been improved after fine-tuning except MobileNetV1-FPGM-30-A16. So the fine-tuned pruned models usually show higher accuracy and faster inference speed while maintaining the negligible changed model size. When we combine pruning and quantiza-

tion in the later section, all the pruned models evaluated are the A16 models, and the number of filters in these models is a multiple of 16.

4.2.2 Quantization

The quantization method we implement is quant-aware-training(QAT) and post-training-quantization(PTQ). The QAT also needs to train the models, but we can get a good accuracy after a few epochs of fine tune. We set the learning rate to 0.01 and use the step decay scheduler to implement QAT for 30 epochs on every model. Like pruning, we still show the results of quantization from three aspects. It should be noted that the weights of the original model are all FP32 format. After the quantization, the weights of convolutional layers come to INT8, the other data (such as weights in batch norm and FC-layer) are still FP32, which is the same as the quantized model architecture mentioned in Section 3.1.3.2.

Table 4.7: Comparison of model size of quantized MobileNetV1 and ResNet50 with baseline.The unit of model size is megabytes(MB).

Model	Method	Base Size	Quantized Size	CR
MobileNetV1	PTQ	17.0	7.5	2.27
ResNet50	PTQ	102.4	32.2	3.18

Significant reduction in model size can be observed in in Table 4.7, and the model sizes of QAT and PTQ are the same, so we only show the PTQ results. Because different models contain different proportions of quantized CNN layers, the CR is not the same. When the proportions of conv layers inside a model is higher, the CR will be closer to 4.

Table 4.8: Comparison of accuracy of quantized MobileNetV1 and ResNet50. Acc1 represents the Top-1 prediction accuracy of the model on the validation set

Model	Method	Base Acc1	Quantized Acc1	Acc1 Drop↓
MobileNetV1	QAT	70.07%	70.97%	-0.90%
MobileNetV1	PTQ	70.07%	69.40%	0.67%
ResNet50	QAT	75.42%	73.63%	1.79%
ResNet50	PTQ	75.42%	74.65%	0.77%

We present the model accuracy of quantized models in Table 4.8. The model accuracy usually decreases after quantization, but due to the training process in QAT, sometimes the model accuracy will increase instead.

Table 4.9: Comparison of inference speed of quantized MobileNetV1 and ResNet50 with baseline. Performance is measured using PaddleInference, the batch size is set to 1, the speedup is the ratio of baseline and pruned performance.

Model	Method	Base Perf	Quantized Perf	Speedup
MobileNetV1	QAT	0.6553	0.2437	2.69
MobileNetV1	PTQ	0.6553	0.2456	2.67
ResNet50	QAT	2.2217	0.6057	3.67
ResNet50	PTQ	2.2217	0.6179	3.60

As shown in Table 4.9, the inference performance is boosted significantly, especially in model with a large proportion of quantized CNNs, such as the ResNet50-PTQ with speedup of 3.67.

4.2.3 Model Compression Pipeline

Quantizing the pruned model constitutes the model compression pipeline. Using the model compression pipeline, we can get smaller model size and faster inference speed, but it will cause in a greater loss of accuracy. It is important to note that the pruning models we evaluate in this section are all based on FPGM algorithm and the number of filters is multiple of 16.

Table 4.10: Comparison of model size of in model compression pipeline. The unit of model size is megabytes (MB). P represents the pruning and P+PTQ represents the PTQ after pruning.

Model	Pruned Ratio	Base	P	P+PTQ	CR
MobileNetV1	30%	17.0	13.5	6.6	2.58
MobileNetV1	50%	17.0	11.0	6.0	2.83
ResNet50	30%	102.4	84.6	27.7	3.70
ResNet50	50%	102.4	61.9	21.1	4.85

The results of model size, after model compression, is presented in Table 4.10. Because the model size of PTQ and QAT is the same, so we only show the PTQ result in the table. The difference in CR column of MobileNetV1 is relatively small when we use different pruning ratios, so in such cases, we can focus more on the trade-off between accuracy and inference performance.

Table 4.11: Comparison of accuracy in model compression pipeline.

Model	Pruned Ratio	Base	P	P+QAT	P+PTQ
MobileNetV1	30%	70.07%	68.44%	67.79%	65.96%
MobileNetV1	50%	70.07%	63.36%	62.40%	60.60%
ResNet50	30%	75.42%	72.95%	73.43%	71.72%
ResNet50	50%	75.42%	68.22%	68.02%	66.86%

In Table 4.11, we can find that the best choice is to use QAT after pruning, which helps to minimize the loss of accuracy more than PTQ. Specially for ResNet50, QAT even recovers the loss of accuracy from the 30% pruned model, where 73.43% > 72.95%.

Table 4.12: Comparison of inference speed in model compression pipeline. Performance is measured using PaddleInference, the batch size is set to 1.

Model	Pruned Ratio	Base	P	P+QAT	P+PTQ
MobileNetV1	30%	0.6553	0.5856	0.2467	0.2466
MobileNetV1	50%	0.6553	0.4894	0.2441	0.2393
ResNet50	30%	2.2217	2.0574	0.5651	0.5708
ResNet50	50%	2.2217	1.9026	0.5258	0.5274

The model compression pipeline shows us the ultimate performance boost in Table 4.12. After observing such results, we should consider some trade-off problems. Using MobileNetV1 as an example, with similar model size and speedup, we should choose the compression method with low accuracy loss, such as the combination of QAT and 30% pruned ratio model.

In order to better demonstrate how model compression pipeline affects the model size, accuracy and inference speed, we illustrate the data in Figure 4.1, Figure 4.2 and Figure 4.3.

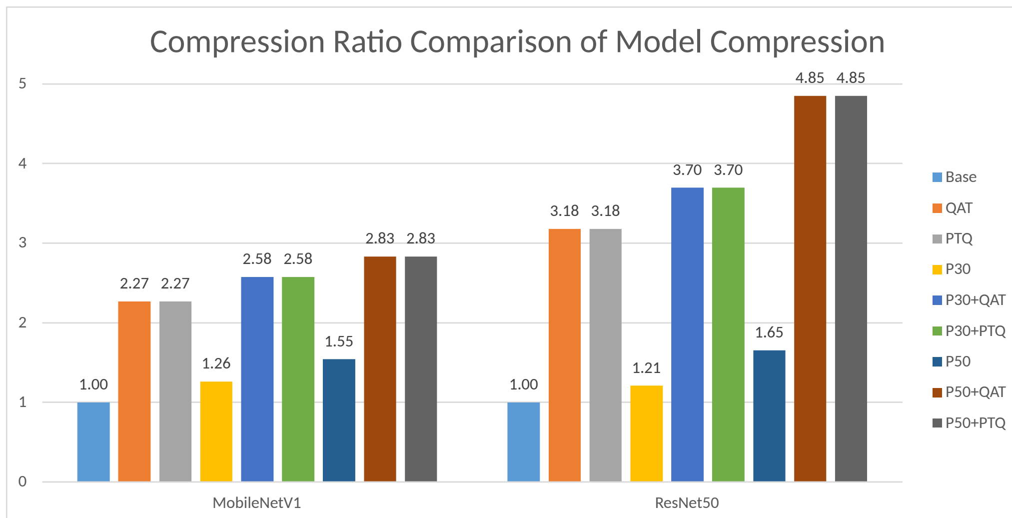


Figure 4.1: The compression ratio comparison from model compression pipeline. The base size of MobileNetV1 and ResNet50 is 17.0 MB and 102.4 MB respectively. The compressed model size can be obtained by dividing base size by compression ratio. The P30 and P50 represents pruning with 30% and 50% pruning ratio respectively.

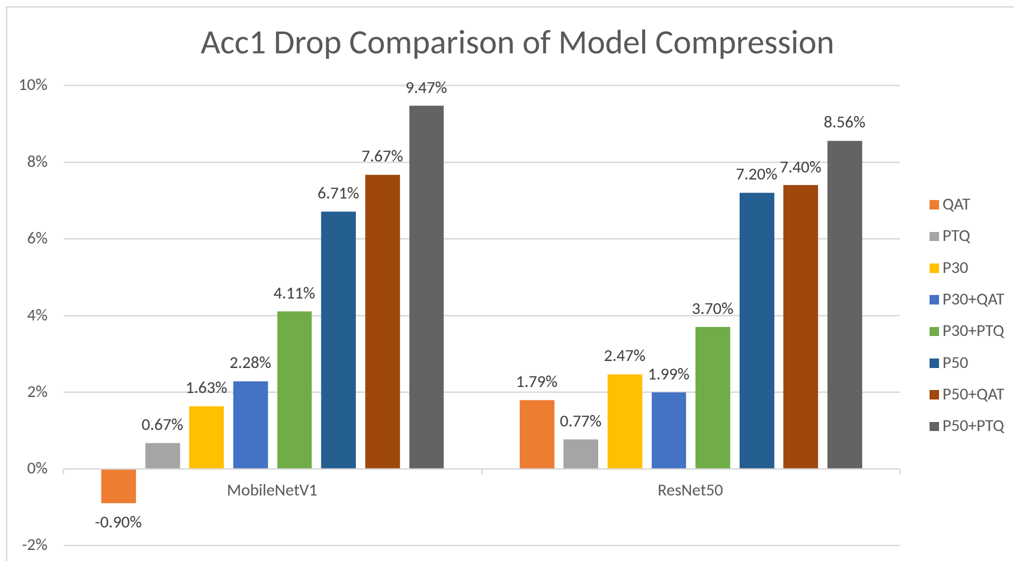


Figure 4.2: The Acc1 Drop comparison from model compression pipeline. The base Acc1 of MobileNetV1 and ResNet50 is 70.07% and 75.42% respectively. The Acc1 of compressed model can be obtained by minus Acc1 Drop from base Acc1.

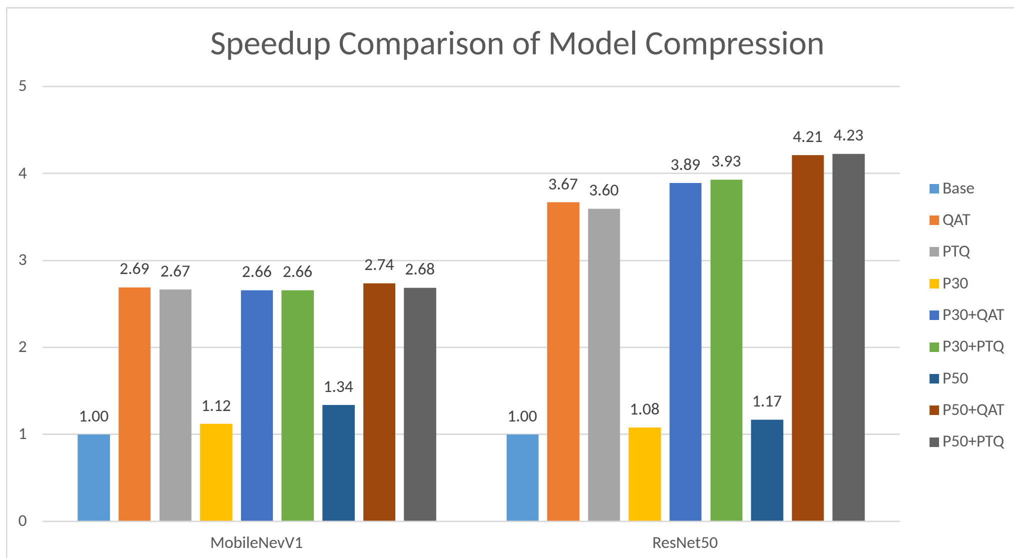


Figure 4.3: The speedup comparison from model compression pipeline. The base inference speed of MobileNetV1 and ResNet50 is 0.6553 ms and 2.2217 ms respectively (batch size = 1). The accelrated speed can be computed by dividing base speed by speedup.

As we can observe from the three bar charts above, the effect of model compression is generally the same: It will decrease model accuracy, but bring smaller model size and faster inference speed. There are many trade-offs among them, and we will advise users on how to choose among them in next chapter.

4.3 Data Compression

In this section, we first use some lossless compression algorithms to compare the compressed size of the original MobileNet model and the PTQ quantized(INT8) MobileNet model.

Table 4.13: Model size(MB) after compression by different compression algorithms

Model	Method	Base	7z	tar	zip	zstd[12](L1)	zstd[12](L19)
MobileNetV1	-	17.0	3.7	3.7	5.2	5.0	4.0
MobileNetV1	PTQ	3.4	3.4	3.4	3.4	3.4	3.4
ResNet50	-	102.4	94.1	94.1	95.2	95.0	94.5
ResNet50	PTQ	32.2	27.6	27.6	28.2	28.1	27.9

In the table 4.13, in addition to the classic 7z, tar, zip lossless compression algorithms, we also use a lossless compression algorithm Zstandard (zstd) developed by Facebook. There are 19 different compression levels in the zstd algorithm[12], where L1 means the least compression and L19 means the most compression.
From the above table 4.13 we can draw two conclusions:

1. After the model is quantized by INT8, the size of the quantized model is smaller than that of other models compressed by lossless compression algorithms.
2. When we directly perform lossless compression on the quantized model, the model size does not change much.

Therefore, in order to obtain a smaller compression ratio, we try to compress the parameters of the quantization model based on the INT8 quantization model. By looking at Table 4.7 in Section 4.2.2 we can know that quantification can already lead to a large compression effect. However, in order to study the generality of this conclusion, we have selected a few additional models for analysis, and the data are as follows:

Table 4.14: Additional Tests: The original size(MB) of the 11 models and the size(MB) after INT8 quantization.

Model name	Original size	INT8 quantized sizes	CR
AlexNet [29]	244.4	237.0	1.03
DarkNet53 [49]	166.7	45.0	3.71
GoogLeNet [59]	28.0	10.1	2.77
InceptionV4 [58]	171.1	47.9	3.57
Xception71 [8]	151.8	45.5	3.34
RedNet152	137.0	41.8	3.28
ResNet34	87.3	23.5	3.71
ResNet50	102.4	32.2	3.19
ResNet152	241.7	67.7	3.57
VGG16 [55]	553.4	509.3	1.09
VGG19	574.7	514.6	1.12

We can see from Table 4.14 that for some models, the compression ratio is very low after INT8 quantization, and the compression effect is very good. But for other models, such as AlexNet, the compression effect after quantization is not good, and the size is basically unchanged.
To explore the reason for this, we first analyze the model parameters after INT8 quantization. We found that the PTQ quantization algorithm does not quantize all the layers in the model, but only quantizes some of them. Because of the particularity of INT8 quantization, we can simply separate the quantized layer and the non-quantized layer according to whether the parameters in the layer are integers. According to this idea, we divided the above model into INT8 part and FP32 part, and calculated the ratio of these two parts.

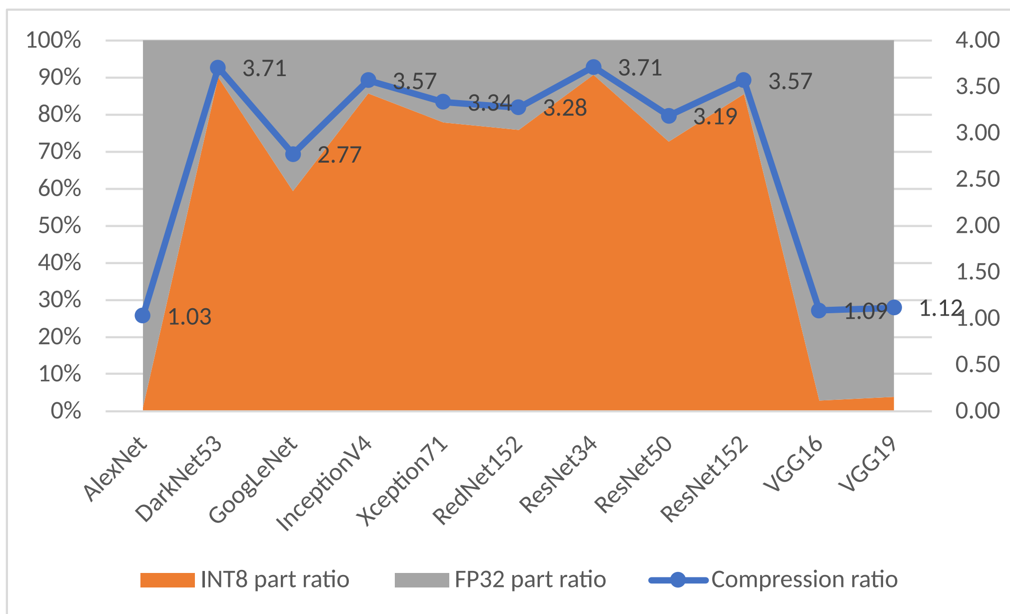
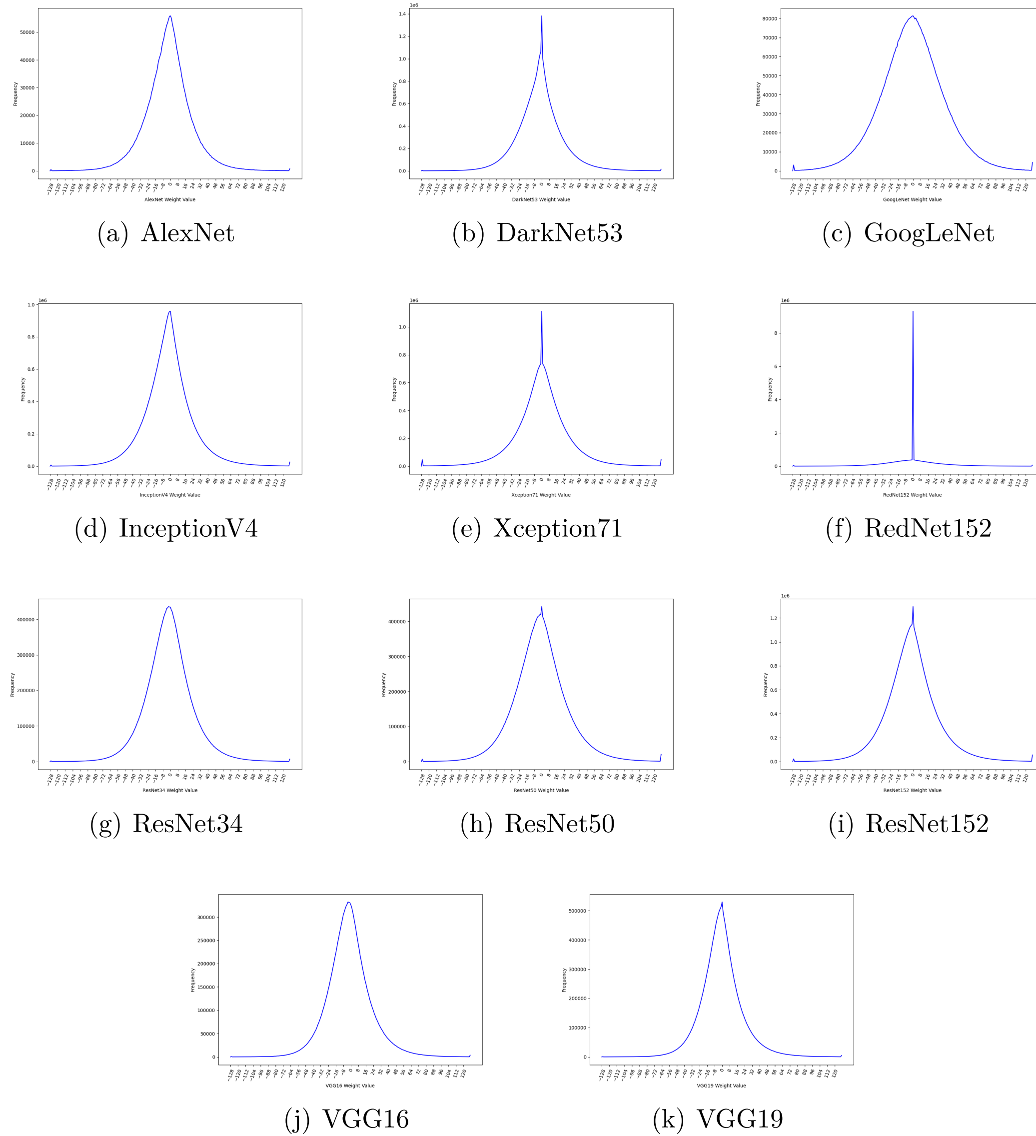


Figure 4.4: The ratio of float weights and integer weights in different models, and the corresponding compression ratio after quantization.

According to Image 4.4 and Table 4.14, it is easy to see that the compression effect of the quantization model is determined according to the number of quantization layers. If the ratio of the parameters of the quantization layer to the total parameters is high, the compression effect is good, otherwise, the compression effect is poor. So if we want to get a generally good compression ratio, we need to compress the quantized INT8 layer and the FP32 layer separately.

4.3.1 INT8 Part Parameter Compression

First, since the quantized integer is INT8, its range is $[-127, 127]$. So we can draw the corresponding graph for the distribution of integer parameters in the model. The model selection is the previous 11 models , and the picture of the integer parameter distribution is as follows:



(a) AlexNet (b) DarkNet53 (c) GoogLeNet
(d) InceptionV4 (e) Xception71 (f) RedNet152
(g) ResNet34 (h) ResNet50 (i) ResNet152
(j) VGG16 (k) VGG19

Figure 4.5: The distribution of integers(INT8) in the model parameters after quantization

We can see that the integer parameter distributions in the above model are all non-uniform. So in the following compression algorithm, the Huffman encoding we choose re-encodes the integer part. The specific results after recoding are as follows:

Table 4.15: The size(MB) of the integer part of the model changes after Huffman coding

Model Name	INT8 part original size	After Huffman encode	CR
AlexNet	2.5	2.1	1.19
DarkNet53	40.6	33.2	1.22
GoogLeNet	6.0	5.3	1.13
InceptionV4	41.1	34.3	1.20

Xception71	35.5	30.0	1.18
RedNet152	31.8	23.2	1.37
ResNet34	21.3	17.7	1.20
ResNet50	23.5	20.1	1.17
ResNet152	58.0	49.2	1.18
VGG16	14.7	12.1	1.21
VGG19	20.0	16.2	1.23

From Table 4.15, we can know that after Huffman coding, the integer part of the model can achieve a compression ratio of about 80%.

4.3.2 FP32 Part Parameter Compression

In Section 4.3, we have used lossless compression algorithms such as 7z and tar for the model as a whole, but since the weights in the Section 4.3 model are a mixture of integer and floating point weights, we will next analyze lossless compression for floating point weights alone. Also from Table 4.14, we can see that AlexNet has the smallest compression ratio after quantization, so the model is chosen as AlexNet. And in addition to the classic 7z, tar, and zip, these lossless compression algorithms also include Zstandard (zstd) developed by Facebook, and zfp algorithm[33][35] developed by Peter Lindstrom at Lawrence Livermore National Laboratory, the specific data are as follows:

Table 4.16: The size(MB) change of AlexNet's FP32 part of data after lossless compression

Lossless compression algorithm	Size	CR
None	234.5	1.00
7z	215.8	1.09
tar	215.9	1.09
zip	217.8	1.08
zstd(L1)[12]	217.3	1.08
zstd(L19)[12]	217.4	1.08
zfp[33]	232.9	1.01

From Table 4.16, we can know that the compression ratio of the lossless compression algorithm for the FP32 part of the model is not very low. So in order to get a lower compression ratio, we try to use lossy compression algorithm of zfp, and compress the FP32 part of the model. The model we also choose AlexNet, the specific data are as follows:

Table 4.17: The size(MB) change of AlexNet's FP32 part of data after lossy compression

zfp(lossy) tolerance setting	Size	CR
$1e-5$	115.9	2.02

$1e-4$	93.9	2.50
$1e-3$	64.6	3.63
$1e-2$	42.6	5.50
$1e-1$	20.4	11.49

We can see that after lossy compression, FP32 data is well compressed. But lossy compression also affects the final model accuracy, so next we will study the relationship between the loss parameter of zfp and the final accuracy. We selected the previous 11 models for testing and plotted the data as follows:

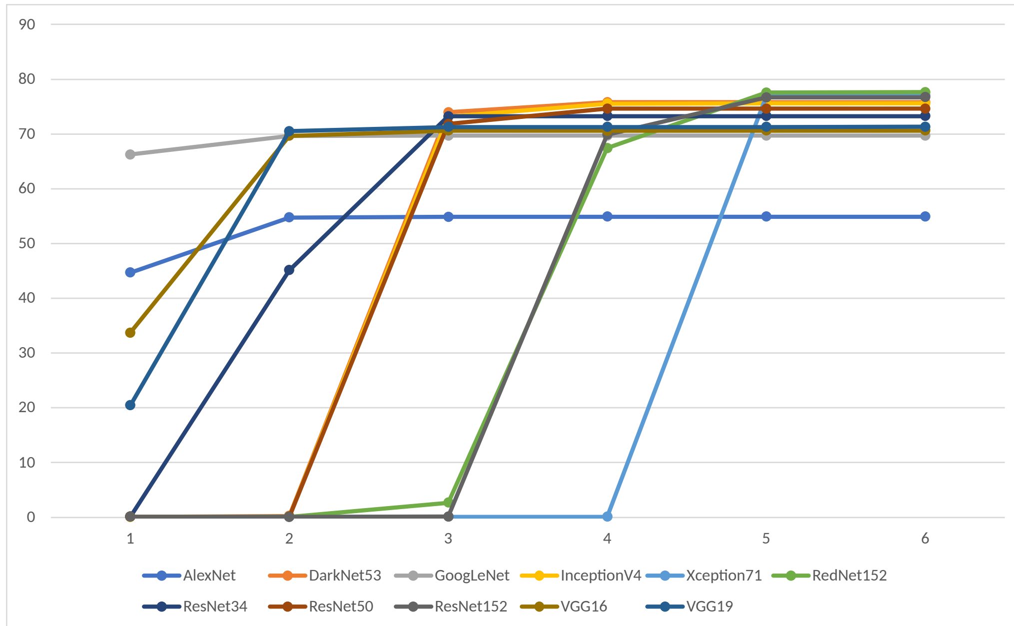


Figure 4.6: The accuracy change of the model after zfp lossy compression.

The ordinate in Figure 4.6 is the accuracy percentage; 1 in the abscissa represents the model accuracy when the zfp loss parameter is tolerance= $1e-1$, 2 represents the model accuracy when the zfp loss parameter is tolerance= $1e-2$, and so on. 6 represents the original model accuracy.
We found that sometimes the accuracy of the model will go to zero under the parameters set globally. Inspired by the previous pruning operation, we perform a separate sensitivity analysis for each layer of the model, so as to set separate parameters for the more sensitive layers in the model.
After performing the sensitivity analysis, we can obtain the model size and accuracy as follows:

Table 4.18: Model FP32 part size(MB), compression ratio of FP32 part and accuracy after lossy compression according to sensitivity analysis result

Model name	Accuracy	Size	CR
AlexNet	54.80%	42.6	5.47

ResNet50	74.37%	8.8	4.19
Xception71	76.79%	10.0	2.97

By treating different types of weights separately, we can achieve a compression rate of 3-6 times with little decrease in accuracy.

4.4 Hybrid Compression

This section will compare the compression of the whole process. The workflow is to prune the original model first, then quantize, and finally perform lossy compression for FP32 part and Huffman coding of the INT8 part. In this part, we select two models for experiments. The specific experimental data are as follows:

Table 4.19: Acc1 and model size comparision after hybrid compression. The DC in Method column denotes Data Compression. The letter B denotes the Base.

Model	Method	B Acc1	Acc1	Drop	B Size	Size	CR
MobileNetV1	P30-PTQ-DC	70.07%	65.73%	4.34%	17.0	3.3	5.15
MobileNetV1	P30-QAT-DC	70.07%	67.69%	2.38%	17.0	3.3	5.15
MobileNetV1	P50-PTQ-DC	70.07%	60.53%	9.47%	17.0	2.7	6.29
MobileNetV1	P50-QAT-DC	70.07%	62.36%	7.71%	17.0	2.7	6.29
ResNet50	P30-PTQ-DC	75.42%	70.91%	4.51%	102.4	18.4	5.57
ResNet50	P30-QAT-DC	75.42%	72.78%	2.64%	102.4	18.4	5.57
ResNet50	P50-PTQ-DC	75.42%	64.90%	10.52%	102.4	13.6	7.53
ResNet50	P50-QAT-DC	75.42%	66.15%	9.27%	102.4	13.6	7.53

As can be seen from Table 4.19, the compression ratio can be 5-6 times while allowing a accuracy drop of about 2 percent. In the case of allowing a dropout of about 4 percent, the compression ratio can be 6-8 times.

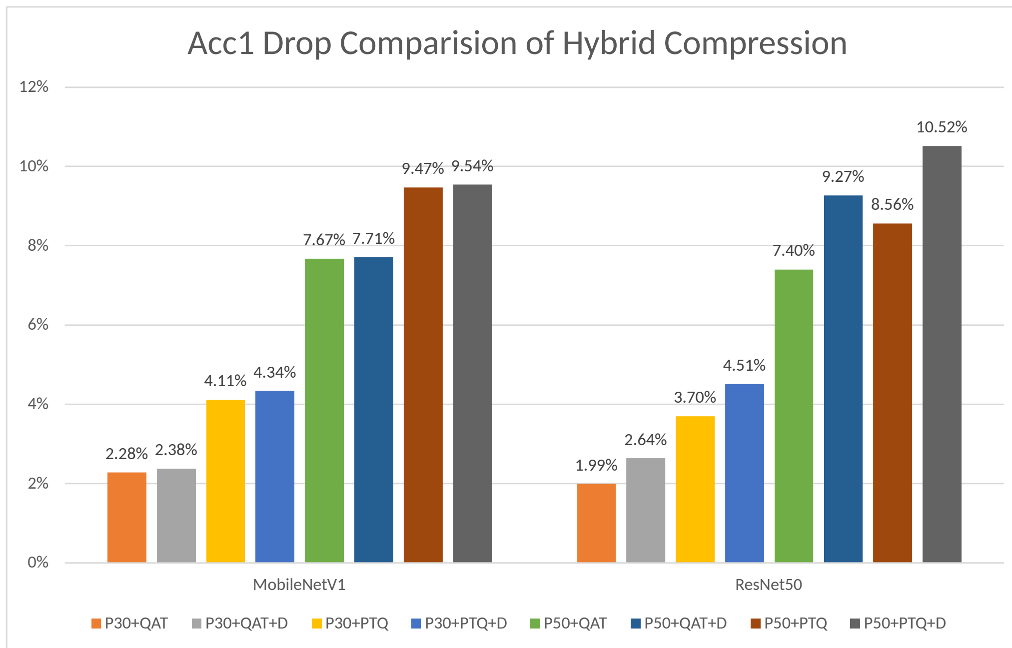


Figure 4.7: The Acc1 drop comparison of hybrid compression. P30+QAT+D represents that we apply the data compression on the P30+QAT model. The base Acc1 of MobileNetV1 and ResNet50 is 70.07% and 75.42% respectively.

As shown in the Figure 4.7, the data compression almost does not introduce any accuracy loss to the compressed MobileNetV1, and cause the loss of accuracy on compressed ResNet50 within 2%.

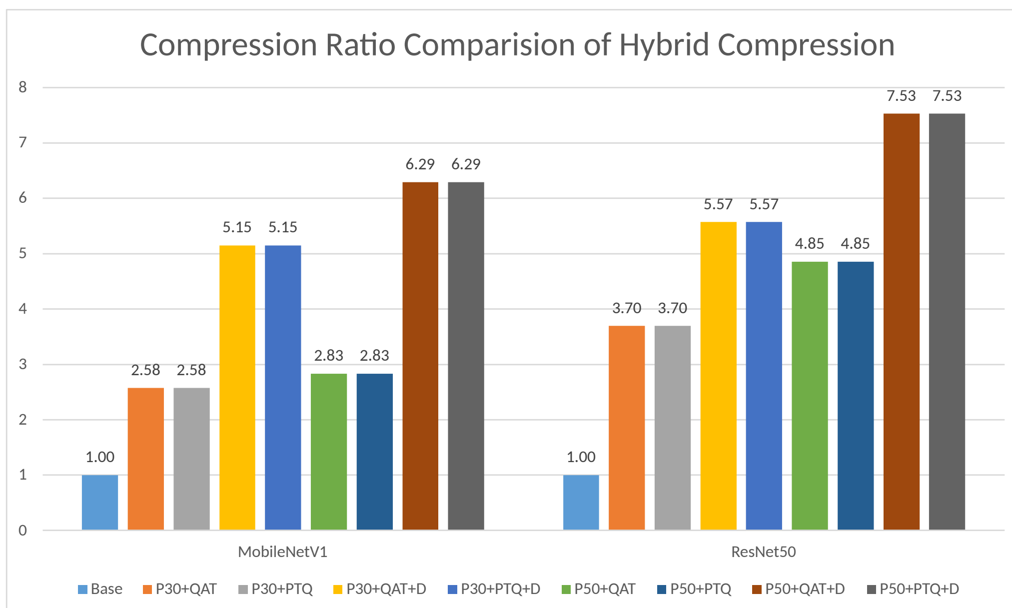


Figure 4.8: The compression ration comparison of hybrid compression. The base size of MobileNetV1 and ResNet50 is 17.0 MB and 102.4 MB respectively.

We can observe that based on model compression, data compression gives us a remarkable compression ratio in the Figure 4.8.

From the perspective of hybrid compression directly, for the MobileNetV1, We obtain a maximum compression ratio of 6.29, a speedup of 2.74, and the accuracy drop of 7.71%. For ResNet50, We achieve a maximum compression ratio of 7.53, a speedup of 4.21, and 9.27% accuracy loss.

4.5 Edge Device

For edge device performance testing, we used two NVIDIA devices: the NVIDIA Jetson Nano and the NVIDIA Jetson Xavier NX. Their basic description and specific experimental results are as follows. It should be noted that the we only evaluate the inference speed on the edge devices, because other results are shown in the previous sections. The data compression will not bring inference speedup on the model, so the results are all measured with model compression methods.

4.5.1 NVIDIA Jetson Nano

4.5.1.1 Hardware Introduction

NVIDIA announced at the 2019 NVIDIA GPU Technology Conference (GTC) the Jetson Nano Development Kit, a $99 computer now available to embedded designers, researchers and DIY manufacturers that delivers the power of modern AI in a compact, easy-to-use platform. Equipped with a quad-core Cortex-A57 processing chip, including 4GB of LPDDR memory and a 128-core Maxwell GPU, it delivers 472 GFLOPS of compute performance and is capable of running multiple algorithms and AI frameworks such as TensorFlow, Keras, PyTorch, Caffe, and more, with support for NVIDIA JetPack supports multiple neural networks running in parallel to achieve image classification, face recognition, speech processing, target detection and object recognition tracking, etc. It is suitable for developing small structure, low cost and low power consumption devices. The specific configuration is shown in the table below.

Table 4.20: NVIDIA Jetson Nano Configuration

Name	Information
GPU	NVIDIA Maxwell architecture, 128 CUDA cores
CPU	Quad-core ARM A57 @ 1.43 GHz
RAM	4GB 64-bit LPDDR4 25.6GB/s
Storage	microSD
INT8 Support	No (Only FP32 and FP16)

4.5.1.2 Results

We chose the same two models as before, ResNet50 and MobileNetV1, for testing on the NVIDIA Jetson Nano. The specific results are as follows.

Table 4.21: Comparison of inference speed on NVIDIA Jetson Nano platform. Performance is measured using PaddleInference, the batch size is set to 1. The values in the table are in milliseconds. The P represents pruning, 30/50 is the pruned ratio.

Model	Inference Precision	Base	P30	P50
MobileNetV1	FP32	20.1459	18.8022	17.7507
MobileNetV1	FP16	25.2085	19.3982	18.4581
ResNet50	FP32	124.7822	90.9884	76.5805
ResNet50	FP16	68.3152	54.1512	45.5273

Since the Nvidia Jetson Nano does not support integer inference, we did not include a quantization algorithm in the experiment. Instead, to exploit the best performance of Jetson Nano, we enabled FP16 half-precision inference when evaluate the models. As can be seen from Table 4.21, the model inference time is decreasing as the pruning ratio increases. However, it is also noticed that the inference time of MobileNetV1 with FP16 precision increases. One reason is that when we use FP16 precision inference, not all the operations in the model are using FP16 data, and if the amount of computations is not large, the speedup of FP16 does not outweigh the overhead of data format conversion(e.g, FP32-FP16-FP32). So, we change the batch size to increase the amount of computation needed for each inference to verify this reason.

Table 4.22: Comparison of inference speed on NVIDIA Jetson Nano platform. The batch size is set to 2.

Model	Inference Precision	Base	P30	P50
MobileNetV1	FP32	47.0476	40.4155	27.6370
MobileNetV1	FP16	37.1609	31.1660	25.3056
ResNet50	FP32	187.8028	150.5707	111.5469
ResNet50	FP16	90.6803	73.1011	56.8960

As shown in Table 4.22, when the batch size is increased to 2, all the results of FP16 precision is less than FP32.

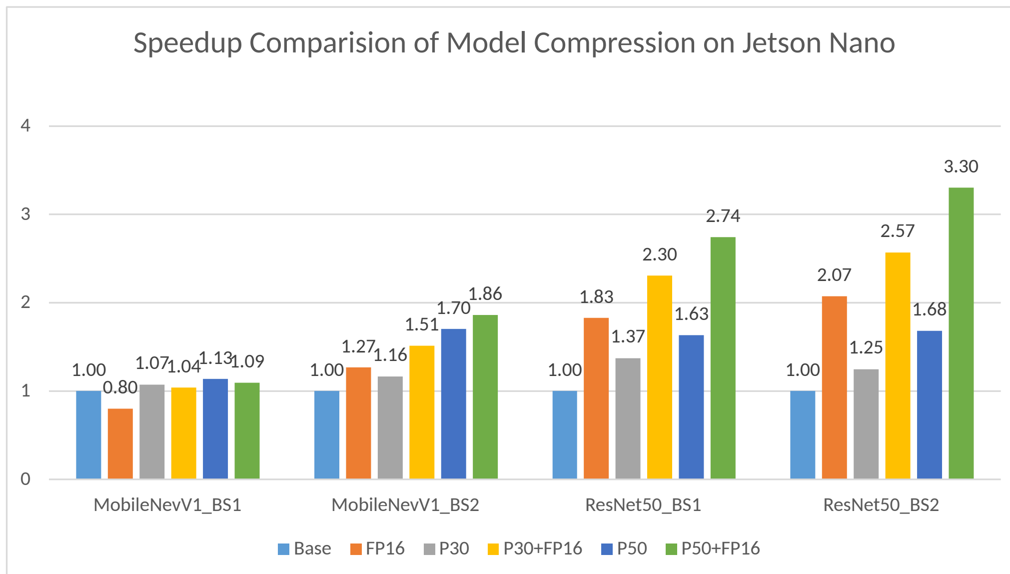


Figure 4.9: The speedup comparison of model compression pipeline on Jetson NANO. BS1 and BS2 represent the cases when the batch size is set to 1 and 2 respectively.

To make it more clear, the speedup is shown in the Figure 4.9. We can observe that in MobileNetV1_BS1, FP16 precision brings negative effect on speedup, where 0.80 $<$ 1.00, 1.04 $<$ 1.07 and 1.09 $<$ 1.13. However, when the batch size is increased to 2, FP16 precision provides a slight acceleration, but the increase of speedup is not as large as that of ResNet50.

4.5.2 Nvidia Jetson Xavier NX

4.5.2.1 Hardware Introduction

Nvidia Jeston Xavier NX is the same size as the Jetson Nano (70mm x 45mm) and brings supercomputer performance to the edge system, making it the world's smallest supercomputer, according to NVIDIA. NX delivers up to 21TOPS of computing power, which is perfect for smart cameras, robots, intelligent industry and other AI edge systems. NX is equipped with 6-core NVIDIA Carmel ARM v8.2 CPU, an NVIDIA Volta GPU with 384 CUDA cores and 48 Tensor cores, 2 deep learning accelerators and 2 vision accelerators. It also has 8 GB of 128-bit LPDDR4x memory with 59.7GB/s bandwidth.

Table 4.23: NVIDIA Jetson Xavier NX Configuration

Name	Information
GPU	NVIDIA Volta architecture, 384 CUDA cores, 48 Tensor cores
CPU	6-core NVIDIA Carmel ARM v8.2 @ 2.26 GHz
RAM	8GB 128-bit LPDDR4x 59.7GB/s
Storage	microSD
INT8 Support	Yes

4.5.2.2 Results

We evaluate the same models on Jetson Xavier NX. In addition to the hardware performance difference, Jetson Xavier NX supports INT8 computing, and we can obviously observe the acceleration empowered by INT8 quantization.

Table 4.24: Comparison of inference speed on NVIDIA Jetson xavier NX. Performance is measured using PaddleInference and the batch size is set to 1. The unit is milliseconds.

Model	Inference Precision	Base	P30	P50
MobileNetV1	FP32	4.1660	3.5346	2.9144
MobileNetV1	INT8-QAT	1.5217	1.3915	1.3329
MobileNetV1	INT8-PTQ	1.5109	1.4187	1.3542
ResNet50	FP32	17.1248	14.8117	12.8865
ResNet50	INT8-QAT	3.9688	3.7815	3.4221
ResNet50	INT8-PTQ	3.9086	3.7846	3.4896

In Table 4.24, We can observe the acceleration from quantization in each column and from pruning in each row for every model. The results of QAT and PTQ are similar because the structure of their inference models is the same. Quantization boosts the inference speed more obviously compared to pruning, but combining pruning and quantization is a better choice if we want to get the ultimate performance.

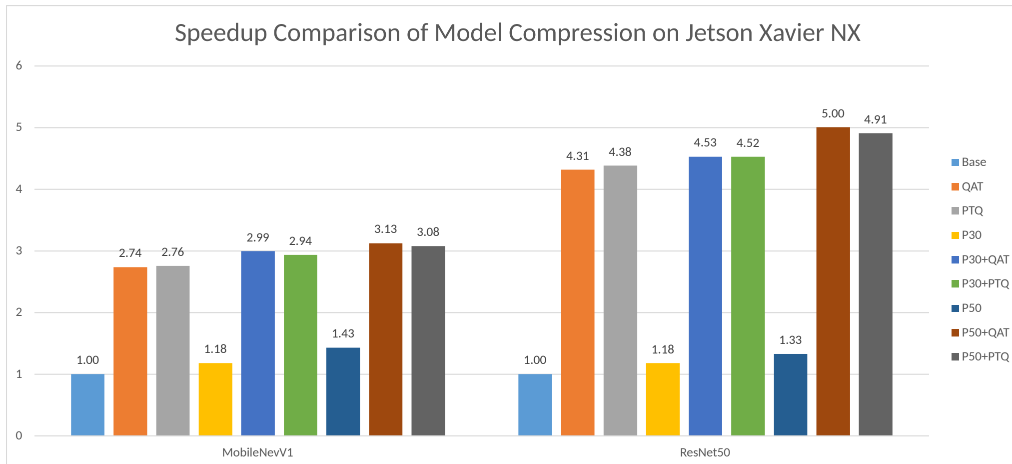


Figure 4.10: The speedup comparison from model compression pipeline on Jetson Xavier NX. The base inference speed of MobileNetV1 and ResNet50 is 4.1660 ms and 17.1248 ms respectively (batch size = 1). The accelerated speed can be computed by dividing base speed by speedup.

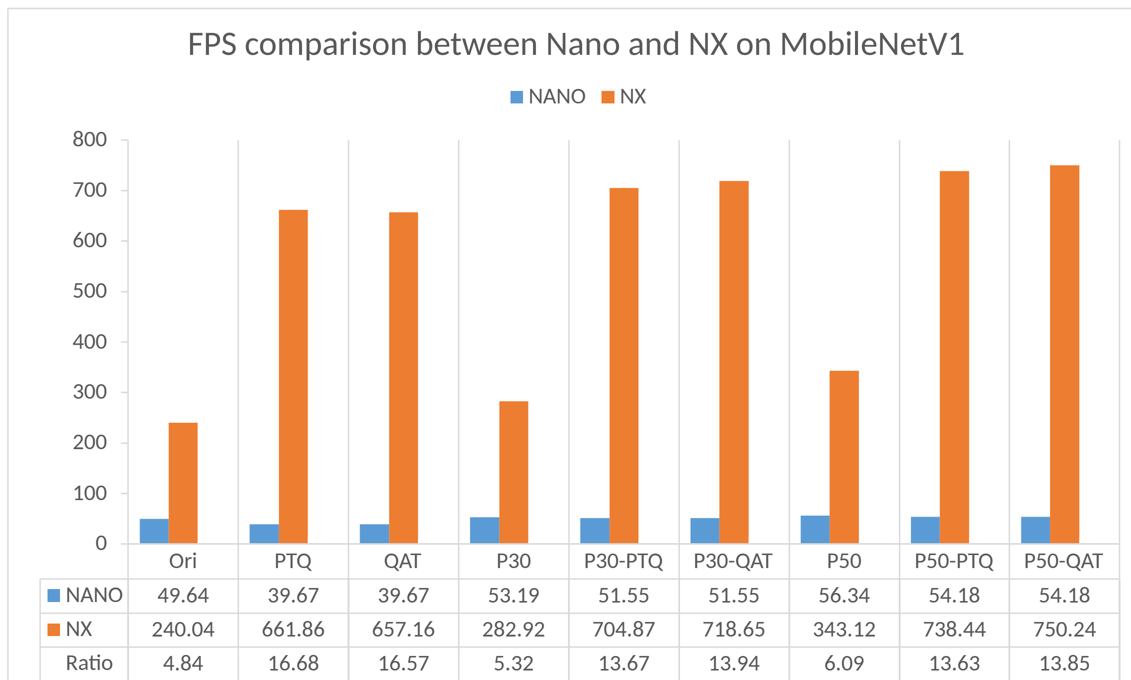
The speedup is shown in Figure 4.10, the result shows the a similar trend of speed up for Jetson Xavier NX and desktop GPU.

4.5.3 Nano Vs NX

In last section, we show the inference time and speedup of the model on Nvidia Jetson Nano and Nvidia Jetson Xavier NX. Now we convert the time to frame per second (FPS), which is the inverse of inference time, to compare the performance of these two edge devices in the Figure 4.11 and Figure 4.12.

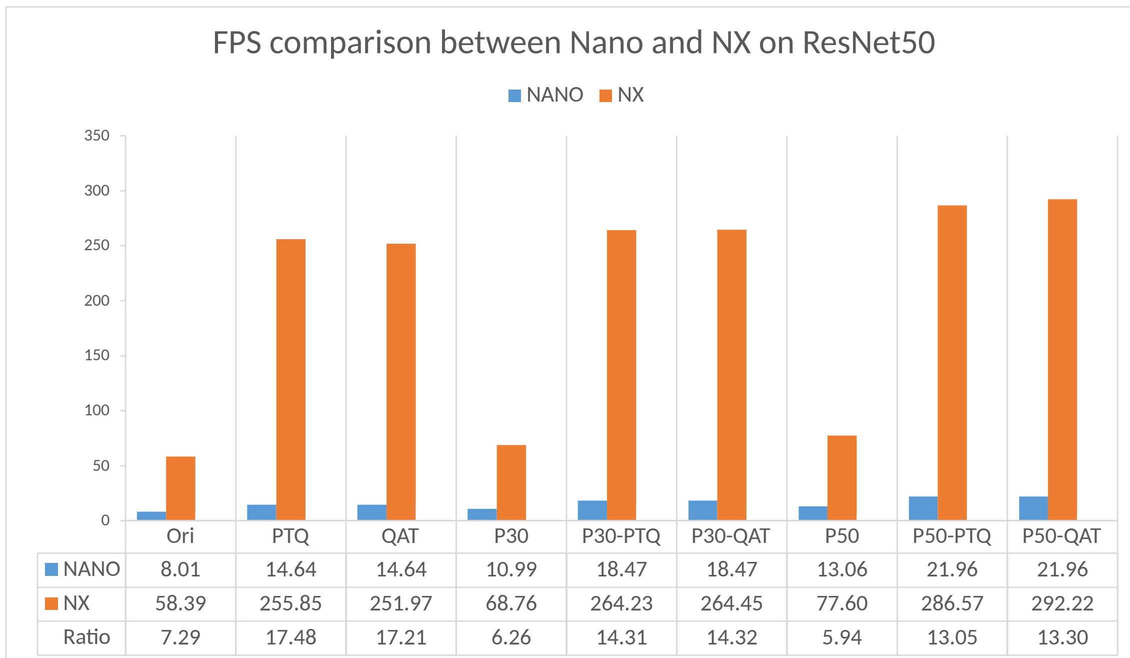
In these two figures, We can observe from the results of Ratio, the FPS of NX is at least 4.84 times higher than Nano for all models with FP32 precision. However, when low precision inference is applied (INT8 quantization in NX and FP16 in Nano), the gap increases dramatically, with NX outperforming Nano by a factor of 13 to 17.5.

When we look at the value of FPS, in the Figure 4.11, Nano can reach about 50 FPS, but compared to the original model, our model compression just brings a slight speedup. What is worse, when we perform FP16 precision inference, FPS decreases instead of increases. In comparison, NX can reach up to 750 FPS in the P50-QAT model with INT8 quantization. In Figure 4.12, although FP16 inference brings some speedup to the original model on Nano, the FPS of all models is less than 22 FPS, which is not able to reach what we usually think of as real-time inference speed, i.e., 30 FPS. However, NX is more powerful, and it can easily infer the quantized ResNet50 model with a maximum FPS of up to 292 FPS.



	Ori	PTQ	QAT	P30	P30-PTQ	P30-QAT	P50	P50-PTQ	P50-QAT
NANO	49.64	39.67	39.67	53.19	51.55	51.55	56.34	54.18	54.18
NX	240.04	661.86	657.16	282.92	704.87	718.65	343.12	738.44	750.24
Ratio	4.84	16.68	16.57	5.32	13.67	13.94	6.09	13.63	13.85

Figure 4.11: FPS comparison between Nano and NX on MobileNetV1. The value above the orange column is the FPS ratio obtained by dividing NX by Nano. The FPS values can be found in the data table in the figure. It should be noted that the FPS value of Nano under the quantization column comes from FP16 precision inference.



	Ori	PTQ	QAT	P30	P30-PTQ	P30-QAT	P50	P50-PTQ	P50-QAT
NANO	8.01	14.64	14.64	10.99	18.47	18.47	13.06	21.96	21.96
NX	58.39	255.85	251.97	68.76	264.23	264.45	77.60	286.57	292.22
Ratio	7.29	17.48	17.21	6.26	14.31	14.32	5.94	13.05	13.30

Figure 4.12: FPS comparison between Nano and NX on ResNet50.

From the figures above we can see that there is a huge difference of performance between the two edge devices, and we will analyze how to choose them in the next chapter.

5

Analysis

5.1 Time Overhead

In this section, we will analyze the time overhead in hybrid compression. The main overhead is spent on model retraining and sensitivity analysis.

5.1.1 Model Retraining

During model pruning and QAT quantization, model retraining is unavoidable. In pruning, we train each model with 90 epochs on the NVIDIA V100 GPU, and in QAT, we train each model with 30 epochs. Take ResNet50 as an example, it takes 160 seconds to train one epoch for a 30% pruned model. It takes $160 * 90 = 14400$ seconds to complete 90 epochs, which is 4 hours. Note that we use a training set of 40K images, while the original ImageNet2012 training set has about 1280K images, which means that 4 hours may turn into $4 * 1280/40 = 128$ hours. However, this is only a single training session, we may need to tune the hyperparameters to get higher accuracy, and the time overhead may increase manifold with the number of training sessions. Compared with retraining of pruning, QAT is easier to converge, and 30 epochs can achieve satisfied results with about 1/3 of the total time spent on one pruning session.

5.1.2 Sensitivity Analysis

In Section 3.1.2, we mention that a number of evaluations is needed to complete the sensitivity analysis. For example, in ResNet50, there are 53 convolutional layers that need sensitivity analysis, if each layer needs to be analyzed with 9 different ratios, then we need to complete $53 * 9 = 477$ evaluations on the validation set. Using the V100 GPU on a 10K validation set, it takes 25 seconds to evaluate one time, so it takes $477 * 25 = 11925$ seconds to complete the sensitivity analysis of ResNet50, which is about 3 hours and 18 minutes. Using the original 50K validation set, a sensitivity analysis could take 990 minutes, or 16.5 hours, to complete.

5.1.3 Others

In addition to model training and sensitivity analysis, there are other parts of our workflow, such as PTQ and data compression. These processes can be completed in just several seconds, and their overhead is negligible compared to model training and sensitivity analysis.

5.2 Network Transmission

In addition to the time required for model preparation, the network transmission time during each model upgrade has a huge impact on the user experience. Therefore, we will analyze the impact of model size on network transmission time before and after hybrid compression.
Generally speaking, people feel the most about the amount of time saved, so we use the number of seconds saved in network transmission after compression than before compression as the evaluation index, which is calculated as follows, assume that the number of seconds saved is t, the size before compression is s_1 (MB), the size after compression is s_2 (MB), and the network bandwidth is b (Mbps).

$$t = \frac{s_1 - s_2}{b/8} \tag{5.1}$$

Using the data from Speedtest by Ookla Global Fixed and Mobile Network Performance Maps [56]. ResNet50 was chosen as the model and a Hybrid Compression method, P30-QAT-DC, was used. From Table 4.19 we can obtain the amount of variation in the model size and thus calculate the following figure based on the local average network transmission bandwidth [56].

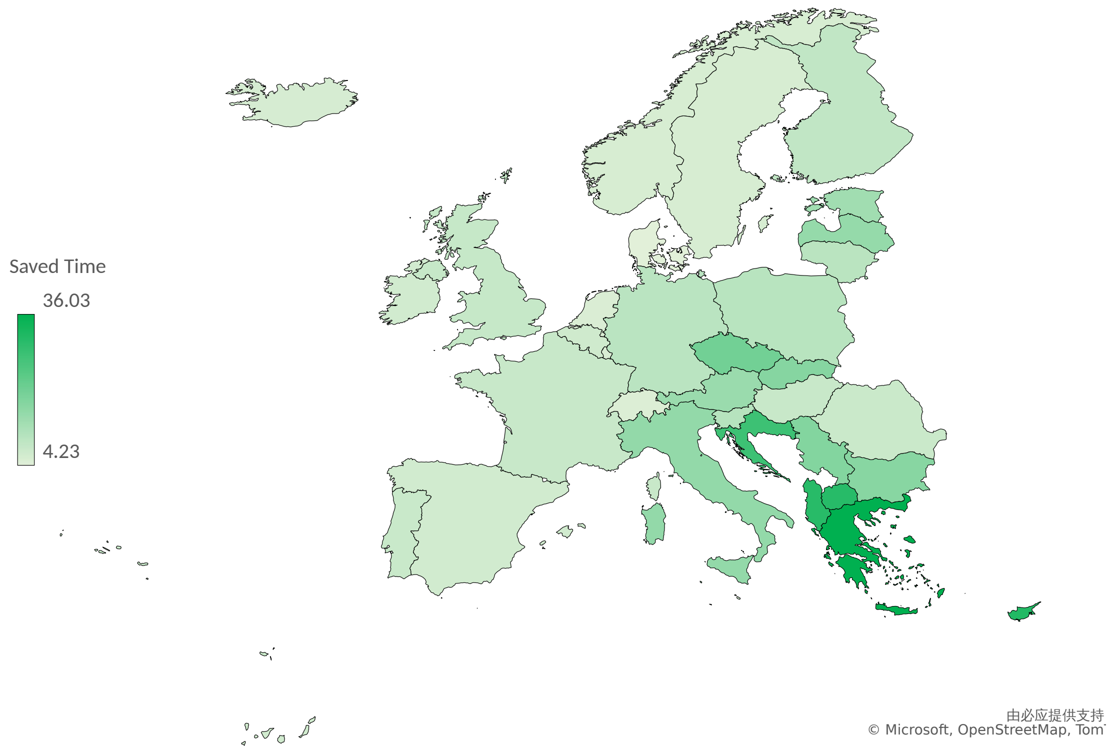


Figure 5.1: Changes in network transmission time before and after compression by the ResNet50 model, taking into account the local network speed.

Figure 5.1 is intended to show the reduction in transfer time, in relation to the region, before and after the use of hybrid compression, when the user transfers the model (e.g. ResNet50). The darker the color is after Hybrid compression, the more

the reduction in network transmission time one perceives. For the European region, some countries are able to reduce the model transmission time by 36 seconds after hybrid compression due to the low local average bandwidth. The reduction in model transfer time of about half a minute is a huge improvement in the user experience for those living in these countries. With this graph, we have a intuitive impression of how the user experience changes with hybrid compression in different countries of Europe.

5.3 Energy

As the computation cost of deep learning models continues to grow, the energy consumption from model training and inference has reached a point where it cannot be ignored. According to estimates within the scientific community, if current trends continue, AI may first become the biggest culprit of the greenhouse effect than providing solutions for climate change.
So we collected some embedded and desktop GPU power consumption and arithmetic data from the nvidia official website, as follows.

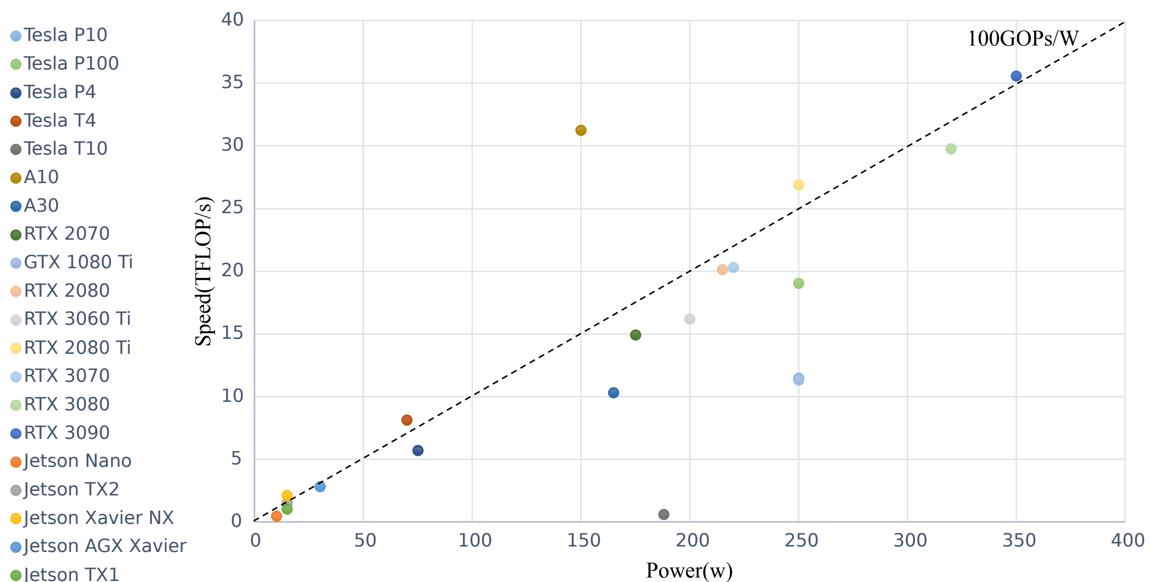


Figure 5.2: Scatter plot of power consumption vs. arithmetic power for different devices.

From the Figure 5.2 we can see that most of the current mainstream devices, the computing power to power consumption ratio is concentrated in 100GOPs/W. The unit GOPs/W can also be converted to GOP/J, where J refers to joules.
From section 5.1, the approximate number of training epochs needed for model preparation is shown. So here, we can use the previous average computing power to power consumption ratio to calculate the power consumption needed for the model preparation. Suppose the number of pictures in the data set is n,the number of GFLOPs of the model is f, the number of training epochs required is e and the final power consumption required is P, then there is the following formula.

$$P \geq \frac{n \times e \times f}{100} \quad (5.2)$$

Table 5.1: Average energy consumption (kw/h) to be consumed by ResNet50 during model preparation, Dataset is ImageNet

Method	Energy consumption (kw/h)
PTQ	0
QAT	0.41
Pruning	1.23
Pruning-PTQ	1.23
Pruning-QAT	1.64
Pruning-Sensitivity analysis	1.51
Pruning-Sensitivity analysis-PTQ	1.51
Pruning-Sensitivity analysis-QAT	1.92

The data in Table 5.1 are only the theoretical power consumption in GPU-only calculations, and if other devices of the computer are added, the power consumption will be much more.

5.4 Deploy On Edge

The purpose of hybrid compression is to enable resource-limited hardware devices to infer models efficiently and to deliver faster network transmission at the edge. In this section, we will analyze the benefits of hybrid compression with respect to edge devices and network transmission.

Nano can infer some lightweight models (e.g. MobileNetV1) in real-time, but there is just a little benefit from pruning when inference the model. When batch size is set to 1, the FP16 precision even slows down the inference speed. When Nano infers more complex models, although model compression can bring some speedup, the inference speed can not reach real-time, which may not meet the requirements of some computer vision tasks.

In contrast, due to the INT8 precision support on hardware, model compression brings significant benefits when the models are deployed on NX. NX can easily be competent in these image classification tasks and maintain high FPS. Moreover, NX shows the potential to perform more complex computer vision tasks (e.g. object detection and semantic segmentation).

In terms of price, the Nano is priced at 99 US dollars, and the NX is priced at 399 US dollars according to NVIDIA. Although the price of NX is 4 times higher than Nano, in terms of performance, NX may be more than 10 times better than Nano.

In summary, hybrid compression is not suitable for all edge devices, and hardware that supports INT8 computing may be more advantageous. In terms of Nano and NX, we believe that Nvidia Jetson Xavier NX is currently the better edge device.

5.5 Discussion

In this section, we will discuss and analyze all the results and give some suggestions about how to select different methods to compress the DNNs model.
First of all, one needs to choose the hardware and the suitable model compression method according to the task requirements, and the model compression in our paper is mainly applicable to computer vision tasks. When we are ready to do model compression on neural networks, we need to find out whether the hardware supports low-bit computation. Because hardware that supports INT8 and lower bit computation may benefit more from model quantization. We suggest that users utilize hardware that supports INT8 precision computation.
We evaluate our hybrid compression in terms of three main metrics, which are accuracy, compression ratio and inference speedup. Before we apply the hybrid compression, we need to consider the trade-offs between these metrics. Suppose the users can not tolerate the accuracy drop after compression. In that case, we recommend that users only use QAT to compress the model because QAT will bring remarkable inference speedup while causing the minimum accuracy loss among all methods. If users want to gain the highest inference speed, we suggest that users can use pruning to reduce the FLOPs firstly, then do the quantization on the pruned model. But in this way, users may spend a lot of time on model retraining after pruning to control the loss of accuracy. To obtain the highest compression ratio, we recommend users perform the whole workflow of hybrid compression. Pruning, quantization, and data compression can reduce the size of the model and save storage, decreasing the network transmission time at the edge and reducing memory cost when inference.
Assuming that we have a real AI model to deploy on the edge application for computer vision task, we recommend that the user deploy the model in the following order. First we need to have hardware that supports INT8 low-precision inference. For fast deployment, we can use the PTQ compression model, since the time overhead of using PTQ is negligible, and the speedup can be significant. After that, we can improve the prediction accuracy of the model by fine-tuning it with QAT to recover some of the accuracy loss due to PTQ. After that, if the we have enough time to train the model, we can using a combination of pruning and QAT to compress the model for the fastest inference speed. When we release or update the model, we can use hybrid compression, which reduces the network transfer time and storage.

6
Conclusion

6.1 Conclusion and Recommendation

In order to bring the powerful capabilities of neural networks to real life and to facilitate deployment of neural network models, we develop the hybrid compression workflow which can reduce the computation cost and size of model, and achieves speedups in model inference and network transmission.

Starting from pruning 30% FLOPs of MobileNetV1 and ResNet50, hybrid compression achieves up to 2.66 and 3.93 of inference speedup, 5.15 and 5.57 model size compression ratio, and keeps accuracy loss to 4.34% and 4.51% for these two models respectively. If the hybrid compression is applied on the MobileNetV1 and ResNet50, both are pruned away 50% FLOPs, it will bring up to 2.74 and 4.21 speedup, 6.29 and 7.53 of compression ratio and suffer the accuracy drop of 7.71% and 9.27% respectively.

When we evaluate the hybrid compression on the edge devices, the speedup on Nvidia Jetson Xavier NX are similar to the results on desktop GPU, while the speedup on Nvidia Jetson Nano is not very impressive. In contrast, NX is a more suitable hardware for deploying compressed models from hybrid compression.

According to previous chapters, we will give some recommendations about how to apply the hybrid compression from following aspects:

(1) **Hardware**. From the point of view of inference speed, energy consumption, and support for quantization, we recommend that users use hardware that supports INT8 or lower bit computation.

(2) **Highest Accuracy** . We recommend QAT for users who cannot tolerate accuracy loss. The QAT shows minimal accuracy loss in our work and sometimes may exceed the accuracy of the original model after retraining.

(3) **Highest Inference Speed**. To obtain the maximum speedup, we recommend using the pruning and quantization pipeline. However, it also takes the most time and energy to train the model to obtain a satisfactory accuracy and fastest inference speed.

(4) **Highest Compression Ratio**. We recommend users perform the whole workflow of hybrid compression, pruning, quantization and data compression.

(5) **In Practical Deployment**. Assuming that the user is ready to use hybrid compression and deploy the model in industry, we recommend the following process. (a) Prepare INT8 supported hardware. (b) Rapid deployment using PTQ. (c) Fine-tune the model using QAT. (d) If can pay for time and energy overhead, use pruning and QAT. (e) To save network transmission time, all

models can use hybrid compression.

6.2 Future Work

Although our hybrid compression can reduce computation cost and model size while boosting the inference speed, there are still limitations for improvement.
Compress other types of Networks. Our hybrid compression is mainly explored for CNN models, and in the future we will explore whether hybrid compression can be applied to models in other application areas (e.g., NLP and recommendation systems).
Other compression method. The retraining of the network in pruning not only takes a considerable amount of time, but also delivers an insignificant speedup. In later work, we will try to use other model compression methods, such as network architecture search, mixed lower bit quantization or matrix factorization, to achieve better effect.
Automatic compression. In our workflow, the pruning and quantization process and the setting of various parameters are set manually. In the future, we consider using reinforcement learning to develop an automatic compression framework to search for the optimal compression solution for every specific model among multiple model compression methods. In terms of data compression, the current sensitivity analysis in data compression only uses a simple iterative selection method, but it is also possible to use reinforcement learning models for compression parameters selection to select the best compression strategy.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Al-Laham, M., and El Emary, I. M. Comparative study between various algorithms of data compression techniques. *IJCSNS 7*, 4 (2007), 281.

[3] Baidu. Baidu AI Studio. `https://aistudio.baidu.com/aistudio/index?lang=en` Accessed January 14, 2022.

[4] Baidu. Paddle-Inference. `https://paddle-inference.readthedocs.io/en/latest/#` Accessed April 2, 2022.

[5] Baidu. PaddleClas. `https://github.com/PaddlePaddle/PaddleClas/blob/release/2.3/README_en.md` Accessed April 2, 2022.

[6] Baidu. PaddleSlim. `https://github.com/PaddlePaddle/PaddleSlim` Accessed April 2, 2022.

[7] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems 33* (2020), 1877–1901.

[8] Chollet, F. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 1251–1258.

[9] Deng, L., Li, G., Han, S., Shi, L., and Xie, Y. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE 108*, 4 (2020), 485–532.

[10] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[11] Diffenderfer, J., Fox, A. L., Hittinger, J. A., Sanders, G., and Lindstrom, P. G. Error analysis of zfp compression for floating-point data. *SIAM Journal on Scientific Computing 41*, 3 (2019), A1867–A1898.

[12] Facebook. Zstandard github project. `https://github.com/facebook/zstd` Accessed January 14, 2022.

[13] Fletcher, P. T., Venkatasubramanian, S., and Joshi, S. Robust statistics on riemannian manifolds via the geometric median. In *2008 IEEE Conference on Computer Vision and Pattern Recognition* (2008), IEEE, pp. 1–8.

[14] Gale, T., Zaharia, M., Young, C., and Elsen, E. Sparse gpu kernels for deep learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), IEEE, pp. 1–14.

[15] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630* (2021).

[16] Goodfellow, I., Bengio, Y., and Courville, A. 6.2. 2.3 softmax units for multinoulli output distributions. *Deep learning, 1* (2016), 180.

[17] Gupta, A., Bansal, A., and Khanduja, V. Modern lossless compression techniques: Review, comparison and analysis. In *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)* (2017), IEEE, pp. 1–8.

[18] Han, J., and Moraga, C. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International workshop on artificial neural networks* (1995), Springer, pp. 195–201.

[19] Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News 44*, 3 (2016), 243–254.

[20] Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[21] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[22] He, Y., Liu, P., Wang, Z., Hu, Z., and Yang, Y. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), pp. 4340–4349.

[23] Hennessy, J. L., and Patterson, D. A. *Computer architecture: a quantitative approach.* Elsevier, 2011.

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

[25] Huffman, D. A. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE 40*, 9 (1952), 1098–1101.

[26] Katz, P. String searcher, and compressor using same, sept. 1991.

[27] Kavitha, P. A survey on lossless and lossy data compression methods. *International Journal of Computer Science & Engineering Technology 7*, 03 (2016), 110–114.

[28] Krishnamoorthi, R. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342* (2018).

[29] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems 25* (2012).

[30] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[31] Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).

[32] Liang, T., Glossner, J., Wang, L., Shi, S., and Zhang, X. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing 461* (2021), 370–403.

[33] Lindstrom, P. ZFP github project. `https://github.com/LLNL/zfp` Accessed January 14, 2022.

[34] Lindstrom, P. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2674–2683.

[35] Lindstrom, P. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics 20*, 12 (2014), 2674–2683.

[36] Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270* (2018).

[37] Loshchilov, I., and Hutter, F. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983* (2016).

[38] Ma, Y., Yu, D., Wu, T., and Wang, H. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Domputing 1*, 1 (2019), 105–115.

[39] Mazumder, A. N., Meng, J., Rashid, H.-A., Kallakuri, U., Zhang, X., Seo, J.-S., and Mohsenin, T. A survey on the optimization of neural network accelerators for micro-ai on-device inference. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems 11*, 4 (2021), 532–547.

[40] Mitra, A. On finite wordlength properties of block-floating-point arithmetic. *International Journal of Electronics and Communication Engineering 2*, 8 (2008), 1709–1714.

[41] Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[42] Nvidia. NVIDIA TensorRT. `https://developer.nvidia.com/tensorrt` Accessed April 2, 2022.

[43] Nvidia. Pytorch Quantization Toolkit. `https://docs.nvidia.com/deeplearning/tensorrt/pytorch-quantization-toolkit/docs/index.html` Accessed April 2, 2022.

[44] ONNX. Open Neural Network Exchange. `https://onnx.ai/` Accessed April 2, 2022.

[45] PaddleSlim. PaddleSlim Official Accuracy Reference. `https://github.com/PaddlePaddle/PaddleSlim/blob/develop/docs/zh_cn/tutorials/pruning/overview.md` Accessed January 22, 2022.

[46] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems 32* (2019).

[47] PyTorch. PyTorch Quantization. `https://pytorch.org/docs/stable/quantization.html` Accessed April 2, 2022.

[48] Ranganathan, N., and Henriques, S. High-speed vlsi designs for lempel-ziv-based data compression. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 40*, 2 (1993), 96–106.

[49] Redmon, J., and Farhadi, A. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[50] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., et al. Imagenet large scale visual recognition challenge. *International journal of computer vision 115*, 3 (2015), 211–252.

[51] Sayood, K. *Introduction to data compression.* Morgan Kaufmann, 2017.

[52] Schwartz, R., Dodge, J., Smith, N. A., and Etzioni, O. Green ai. *Communications of the ACM 63*, 12 (2020), 54–63.

[53] Seward, J. The bzip2 program. *vers. 0.1 pl2, http://www. muraroa. demon. co. uk* (1997).

[54] Shannon, C. E. A mathematical theory of communication. *The Bell System Technical Journal 27*, 3 (1948), 379–423.

[55] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[56] Speedtest. Average internet speed across Europe. `https://datavis.europeandatajournalism.eu/obct/connectivity/` Accessed May 17, 2022.

[57] Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture 15*, 2 (2020), 1–341.

[58] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence* (2017).

[59] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2015), pp. 1–9.

[60] TensorFlow. TensorFlow Lite. `https://www.tensorflow.org/lite` Accessed April 2, 2022.

[61] Wang, F., Zhang, M., Wang, X., Ma, X., and Liu, J. Deep learning for edge computing applications: A state-of-the-art survey. *IEEE Access 8* (2020), 58322–58336.

[62] Wikipedia contributors. Rectifier (neural networks) — Wikipedia, the free encyclopedia, 2022. [Online; accessed 27-May-2022].

[63] Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602* (2020).

[64] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. Dive into deep learning. *arXiv preprint arXiv:2106.11342* (2021).

[65] Zhou, G., Zhu, X., Song, C., Fan, Y., Zhu, H., Ma, X., Yan, Y., Jin, J., Li, H., and Gai, K. Deep interest network for click-through rate

prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining* (2018), pp. 1059–1068.

[66] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on information theory 23*, 3 (1977), 337–343.